

Александр
Жадаев

PHP

ДЛЯ НАЧИНАЮЩИХ

**Создавайте
динамические
веб-сайты
с помощью PHP
и MySQL**

**Изучите основы
объектно-
ориентированного
программирования**

**Применяйте
технология
AJAX**

**Используйте
на практике
примеры
из книги**



ББК 32.988.02-018
УДК 004.738.5
Ж15

Жадаев А. Г.

Ж15 PHP для начинающих. — СПб.: Питер, 2014. — 288 с.: ил.
ISBN 978-5-496-00844-0

Если у вас есть опыт верстки веб-страниц и вы хотите перейти на новый уровень разработки, то эта книга для вас. Вы познакомитесь с наиболее популярным языком программирования для создания веб-приложений — PHP. Благодаря практическим примерам в книге вы научитесь разрабатывать веб-приложения, превращать статические сайты в динамические, использовать веб-технологии AJAX для загрузки больших объемов данных на сайт. Кроме того, узнаете, как управлять базами данных с помощью MySQL, и познакомитесь с концепциями объектно-ориентированного программирования.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018
УДК 004.738.5

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

В оформлении обложки использованы иллюстрации shutterstock.com.

Краткое содержание

От издательства	9
Глава 1. Что такое язык PHP.....	10
Глава 2. Веб-приложения.....	52
Глава 3. Знакомство с MySQL	79
Глава 4. Операторы и функции языка SQL	146
Глава 5. Работа с базами данных и их администрирование из веб-приложений.....	205
Глава 6. Технология AJAX	237
Глава 7. Концепция объектно-ориентированного программирования	261

Оглавление

От издательства	9
Глава 1. Что такое язык PHP.....	10
1.1. Работа сценария PHP.....	11
1.2. Основы синтаксиса языка PHP	14
1.3. Создание и запуск первого сценария PHP	15
1.4. Комментарии внутри кода PHP.....	18
1.5. Переменные	20
1.6. Типы данных PHP	21
boolean	21
string.....	21
integer.....	22
float.....	22
array	22
resource	24
NULL	24
1.7. Константы в PHP	24
1.8. Операторы PHP.....	26
Операторы присвоения.....	26
Математические операторы	27
Комбинированные операторы присвоения	28

Строковые операторы	28
Операторы инкремента и декремента	29
Битовые операторы	30
Операторы сравнения	31
Логические операторы	32
Приоритеты операторов	33
Условные операторы	34
Оператор switch	36
Операторы циклов	37
1.9. Работа с функциями в PHP	42
Синтаксис функций PHP	43
Передача аргументов	44
Значения параметров по умолчанию	46
Видимость переменных	46
1.10. Операторы повторного использования кода	49
1.11. Резюме	51
Глава 2. Веб-приложения	52
2.1. Работа с формами	52
2.2. Загрузка и обработка файлов	60
2.3. Сеансы	64
2.4. Работа с cookie	67
2.5. Работа с FTP	72
Загрузка файлов на сервер FTP	74
Скачивание файла с сервера FTP	76
2.6. Резюме	77
Глава 3. Знакомство с MySQL	79
3.1. Что такое MySQL	79
3.2. Основные сведения о реляционных базах данных	80
Таблицы	80
Первичный ключ	81

Связи между таблицами. Внешний ключ	82
Целостность данных	84
3.3. Проектирование базы данных	86
3.4. Управление базой данных с помощью SQL	89
Выполнение SQL-команд	90
Создание базы данных	92
Работа с таблицами	94
Типы данных в MySQL	99
Свойства столбцов	106
Другие команды работы с таблицами	120
Ввод данных в таблицу	122
Извлечение данных из таблиц	132
Вложенные запросы	138
Объединение результатов запросов	139
Выгрузка данных в файл	140
Изменение данных	142
3.5. Резюме	145
Глава 4. Операторы и функции языка SQL	146
4.1. Операторы и функции проверки условий	146
Операторы сравнения	147
Операторы сравнения с результатами вложенного запроса	161
Логические операторы	166
Операторы и функции, основанные на сравнении	168
4.2. Групповые функции	172
Перечень групповых функций	172
Параметр GROUP BY	180
Параметр HAVING	182
4.3. Числовые операторы и функции	183
Арифметические операторы	183
Алгебраические функции	183
Тригонометрические функции	185

4.4. Функции даты и времени.	186
Функции получения текущей даты и времени.	186
Функции получения компонентов даты и времени	187
Функции сложения и вычитания дат	191
Функции преобразования форматов дат.	195
4.5. Символьные функции	198
4.6. Резюме.	204

Глава 5. Работа с базами данных и их администрирование из веб-приложений.

205

5.1. Интерфейс с PHP	205
Подготовительные действия	205
Выполнение запроса к базе данных	209
Обработка ошибок.	214
Ввод данных в базу.	217
5.2. Администрирование и безопасность баз данных MySQL	225
Учетные записи пользователей.	225
Система привилегий доступа	229
5.3. Резюме.	236

Глава 6. Технология AJAX.

237

6.1. Как работает AJAX.	238
Знакомство с XML.	239
Первое веб-приложение с использованием AJAX	242
6.2. Работа с данными XML.	251
6.3. Работа с MySQL.	255
6.4. Резюме.	260

Глава 7. Концепция объектно-ориентированного программирования.

261

7.1. Классы и объекты.	262
Создание классов, свойств и методов	263
Создание объектов — экземпляров класса.	265
Обращение к свойствам и методам	266

7.2. Реализация наследования в PHP	267
Перекрытие	269
Статические методы класса	270
Константы класса	271
Обращение к элементам классов	271
Проверка типа объекта	273
Клонирование объекта	273
7.3. Абстрактные классы	274
Интерфейсы	275
Предотвращение перекрытия — final	276
Итераторы	277
7.4. Функции для работы с классами и объектами	277
get_class_methods()	278
get_class_vars()	278
get_object_vars()	279
method_exists()	280
get_class()	281
get_parent_class()	282
is_subclass_of()	282
7.5. Обработка исключительных ситуаций	283
7.6. Резюме	287

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты vinitski@minsk.piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.

Глава 1

Что такое язык PHP

Аббревиатурой PHP называют один из наиболее популярных языков, используемых для создания веб-приложений. Первоначально PHP означало *Personal Home Page* — *персональная домашняя страница*, но потом, с расширением области применения языка, аббревиатура PHP стала расшифровываться как *PHP: Hypertext Preprocessor* — *PHP: гипертекстовый препроцессор*. Текущая версия языка PHP предназначена для создания веб-приложений самого широкого назначения, например, на его основе создана небезызвестная социальная сеть Facebook. В настоящее время PHP поддерживается подавляющим большинством хостинг-провайдеров, его используют сотни тысяч программистов.

Язык PHP разрабатывается группой энтузиастов в рамках проекта с открытым кодом. Проект распространяется под собственной лицензией, несовместимой с GNU GPL. Официальный сайт языка находится по адресу www.php.net. На момент написания книги последним стабильным релизом языка являлся PHP 5.4.12 и 5.3.22. Несколько лет назад были предприняты попытки создания версии PHP 6, но сейчас этот проект по некоторым соображениям признан неперспективным. Поэтому при знакомстве с PHP мы будем основываться на версиях 5.3, 5.4, имея в виду, что при практическом применении своих программ, то есть размещении их в Сети, вы всегда будете ограничены возможностями, предлагаемыми хостинг-провайдерами. Опыт же показывает, что провайдерам свойственен вполне понятный консерватизм, и ныне наиболее распространена поддержка версий 5.2–5.4, причем не самых последних релизов. Так что новые и новейшие возможности языка PHP следует использовать осторожно.

1.1. Работа сценария PHP

Как работает приложение, созданное на основе языка PHP? Принцип очень похож на тот, что используют обычные, статические веб-страницы, созданные на основе языка HTML. В последнем случае на веб-сервере создается сайт, состоящий из нескольких страниц с кодом HTML. Посетитель, зашедший на сайт, загружает их в браузере, щелкая на гиперссылках, представленных на этих страницах.

Щелчок на ссылке инициирует цепочку операций взаимодействия между компьютером пользователя и веб-сервером. Когда вы щелкаете на ссылке какой-то веб-страницы, ваш браузер отправляет запрос на веб-сервер, который считывает HTML-код запрашиваемой страницы, формирует ответ и отправляет его обратно браузеру. В результате содержимое страницы, отображаемой в ответ на запрос пользователя, будет всегда одним и тем же, то есть эта веб-страница *статична*. Вы не можете никаким образом изменить ее, уточнить отображаемый контент, передав в запросе серверу определенные данные.

Например, если вы посетите сайт со статическими веб-страницами, предлагающими данные о прогнозах погоды по всей стране, то *не сможете* указать веб-серверу, что вас интересует погода в определенном месте, скажем в Москве, введя название города в специальное поле ввода и щелкнув на ссылке для отображения данных. Чтобы реализовать такую функцию на статическом сайте, вам пришлось бы создать огромный список городов и для каждого из них оформить ссылку на статическую страницу с нужной информацией. Это крайне неудобно и совершенно нереализуемо, когда список выбора достаточно велик или запрашиваемая информация непрерывно изменяется, как на сайтах погоды.

Все это легко решается средствами языка PHP, позволяющего создавать *динамические* сайты! С помощью PHP при создании веб-страниц вы сможете включать в них специальный сценарий и передавать сценарию данные, которые он должен использовать при обработке запроса вашего браузера. Например, для предыдущего примера сайта погоды вы сможете на одной из его веб-страниц с помощью HTML-формы выбрать нужный город и нажать кнопку подтверждения выбора. Браузер автоматически запросит сервер, указав ему адрес той веб-страницы, которая должна отобразиться в ответ на ваш запрос, и передаст ей всю информацию, введенную вами в форму, в данном случае название выбранного города. Далее сценарий, размещенный внутри запрашиваемой страницы, получит эти данные, обработает их и передаст вам в ответ именно то, что требуется, — прогноз по Москве.

Такие веб-страницы, «начиненные» кодом сценария PHP (или другого аналогичного языка), называются динамическими, поскольку могут реагировать на действия посетителя сайта, передавая ему в ответ на сделанный выбор именно ту информацию, которая нужна. В отличие от статического сайта на динамическом она может меняться с течением времени. Это важно, поскольку, например, погода — вещь переменчивая, а сайт с прогнозом погоды должен показывать самую свежую информацию. Для этого сценарий PHP постоянно обновляет свои данные, обращаясь к нужным источникам: базам данных, показаниям приборов и т. д. Все эти функции легко реализуются средствами PHP, достаточно обширными и развитыми для решения огромного круга задач.

Для исполнения функций, реализуемых сценариями PHP, на веб-сервере должны содержаться специальные средства обработки кода сценария. Их и используют средства PHP, которые играют роль *препроцессора*, то есть предварительного обработчика запроса. Именно они исполняют код сценария внутри запрошенной страницы, передают ему данные из запроса и отправляют ответ браузеру, пользуясь возможностями веб-сервера. Таким образом, средства языка PHP представляют собой дополнение веб-сервера, которое делает возможным интерактивное взаимодействие посетителя сайта с его содержимым.

Итак, чтобы ваш PHP-сценарий смог работать, ему нужны следующие компоненты: браузер, веб-сервер и средства самого препроцессора PHP, входящие в состав веб-сервера. Эти средства предназначены для обработки кода сценария внутри страниц динамического сайта, отображаемых по запросам посетителей в ответ на их действия на сайте.

Кроме того, для создания полнофункциональных PHP-сценариев требуется обеспечить сохранение данных на веб-сервере. Это можно реализовать сохранением информации в файле, и для простых сценариев такой метод вполне приемлем. Но для сложных, профессиональных сценариев применяются более развитые средства — базы данных, обеспечивающие хранение и извлечение данных любого типа по специальным запросам. Эти запросы формируются на специальном языке, называемом SQL (*Structured Query Language* — язык структурированных запросов). SQL — это международный стандартный язык для работы с базами данных, и мы познакомимся с ним далее в этой книге.

Для исполнения запросов на веб-сервере с поддержкой PHP должен быть установлен еще один сервер, который будет воспринимать запросы к базе данных, поступающие из сценария, и исполнять их. Таких серверов множество, но в среде разработчиков PHP наиболее популярен MySQL, поддерживаемый и бесплатно распространяемый на сайте mysql.com. Его мы и выберем для изучения.

Подведем итог: для практической работы по созданию и исполнению сценариев PHP нам нужен браузер и веб-сервер, средства самого языка PHP и сервер MySQL. Любой провайдер, обеспечивающий работу с PHP, предоставляет эти инструменты для развертывания сайта на хостинге. Однако для создания своих профессиональных сценариев лучше будет установить инструменты работы с PHP на рабочем компьютере, локально, что даст вам более гибкий и удобный доступ к PHP-сценарию и средствам его отладки. Поэтому для изучения материала этой книги вам нужно обзавестись всеми этими компонентами, установив их на свой компьютер. Это можно сделать следующим образом.

- ❑ Вы можете загрузить на свой компьютер установочный пакет PHP с сайта www.php.net и установить его самостоятельно, пользуясь инструкцией на сайте.
- ❑ Можете установить на свой компьютер сервер MySQL, загрузив с сайта mysql.com соответствующий дистрибутив.
- ❑ Если на вашем компьютере отсутствует веб-сервер (IIS, Apache и др.), поддерживающий PHP, вы должны запустить его. Для системы Windows вам нужно только ввести в действие сервер IIS, воспользовавшись инструментом добавления компонентов Windows на Панели управления. Другой вариант — загрузка установочного пакета сервера Apache с сайта производителя www.apache.org с последующей установкой и настройкой.
- ❑ Воспользоваться специальными средствами для создания среды разработки PHP-приложений — так называемыми интегрированными пакетами работы с PHP. В их состав входит все, что нужно для разработки сценариев PHP: веб-сервер, SQL-сервер, средства PHP, дополнительные полезные инструменты, причем в интегрированном виде. Вам не придется самостоятельно устанавливать веб- и SQL-сервер и подключать к нему PHP (тем более что это требует достаточно глубоких познаний в этой сфере) — за вас все сделает инсталлятор пакета.
- ❑ Прибегнуть к интегрированным средам разработки (IDE), включающим, помимо компонентов среды для работы с PHP, средства отладки сценариев PHP. Это очень полезные возможности, которые значительно облегчают разработку сценариев, позволяя их пошагово исполнять и контролировать результаты работы кода.

Рекомендую использовать две последние возможности как самые доступные для начинающего программиста, чтобы не усложнять себе освоение самого языка установкой и настройкой установочных пакетов PHP и MySQL. В качестве интегрированного пакета средств PHP мы выберем XAMPP, дистрибутив которого можно бесплатно загрузить на сайте производителя www.apachefriends.org. Он включает в себя веб-сервер Apache, сервер MySQL, средства PHP и множество

других полезных вещей, например почтовый сервис, сервер FTP. Пакет XAMPP устанавливается на компьютерах Windows XP/Vista/7 и Linux, в процессе установки инсталлятор потребует от вас лишь ответить на несколько вопросов, уточняющих конфигурацию среды, а в результате вы получите полный набор средств разработки PHP. Это избавит вас от непростой установки и настройки средств PHP из отдельных компонентов.

В качестве среды IDE рекомендую использовать программу PHPEdit, условно бесплатно распространяемую на сайте разработчика www.phpedit.com. Лицензия предоставляется на месяц, но использовать бесплатно программу вы сможете два месяца, после чего решите, нужна ли она вам настолько, что следует купить ее (около \$100). Для сложных сценариев PHP такие средства отладки, которые предлагает PHPEdit, весьма полезны, кроме того, программа помогает вводить код, отображая подсказки и подсвечивая различные компоненты PHP-кода разным цветом.

1.2. Основы синтаксиса языка PHP

Синтаксис языка PHP достаточно прост для изучения, а его корни лежат в языках Perl, Java и C. Однако в отличие от этих языков PHP изначально был разработан как язык программирования, специально предназначенный для написания веб-приложений (сценариев), исполняющихся на веб-сервере. В этом состоит его преимущество, поскольку его средства позволяют быстро и эффективно создавать полнофункциональные веб-приложения.

Другое достоинство PHP — предоставляемая им возможность внедрять свои сценарии в HTML-код веб-страниц, что значительно упрощает задачу создания динамических сайтов. Благодаря PHP разработчики могут динамически изменять HTML-код страниц в зависимости от действий посетителя сайта. Например, при вводе им данных в поля формы, или установке переключателей, флажков, или выборе определенного элемента списка сервер учтет все эти данные и динамически перестроит всю информацию, передаваемую браузеру.

Вместе с тем язык PHP весьма прост и легок для освоения. Даже начинающие программисты смогут быстро научиться создавать на нем достаточно сложные сценарии, реализующие функции и интерфейс на уровне профессиональных сайтов.

Итак, приступим к изучению языка PHP. Надеюсь, что вы прочли предыдущий раздел этой книги и у вас на компьютере установлены серверы Apache и MySQL, чтобы вы могли повторять вслед за изложением примеры конструкции языка PHP.

Если вы этого не сделали — настоятельно рекомендую вернуться назад и выполнить установку этих средств.

Сначала мы займемся изучением синтаксиса языка PHP. Мы познакомимся с типами данных, используемых в PHP, базовыми конструкциями по управлению ходом работы программы, функциями, способами передачи данных в сценарии, работы с базами данных MySQL, открытия сеансов работы со сценарием и записью cookie-файлов. По мере изложения материала применение этих средств будет демонстрироваться на примерах сценариев, исполняемых на локальном сервере пользовательского компьютера. Отсюда видна важность установки этих средств.

1.3. Создание и запуск первого сценария PHP

Первый сценарий будет простым, так как он нужен только для демонстрации общей структуры PHP-кода и его совместного использования с кодом HTML. Откройте какой-либо текстовый редактор, например Блокнот системы Windows, и запишите в него такой PHP-код:

```
<?php
    phpinfo();
?>
```

Далее откройте корневую папку `htdocs` вашего сервера Apache, в котором сохраняются исполняемые сценарии, и создайте там подпапку с названием `examples`. Мы будем сохранять в ней файлы PHP-сценариев, создаваемых в процессе изучения. Назовите файл `start.php` и сохраните его в этой папке.



ПРИМЕЧАНИЕ

Если вы используете пакет XAMPP, то папка `htdocs` находится внутри папки, в которую вы установили пакет. Обычно эта папка называется так: *имя_диска:\XAMPP*. При работе с сервером Apache напрямую этот каталог должен быть указан в директиве `DocumentRoot` в файле `httpd.conf`. Если же вы предпочитаете работать с сервером IIS, то должны помещать свои сценарии в папку `inetpub\wwwroot`.

Теперь все готово для исполнения первого сценария. Откройте свой браузер и в адресной строке введите `http://localhost/examples/start.php`. В окне браузера отобразятся сведения об установленной версии и конфигурации установленного языка PHP (рис. 1.1).

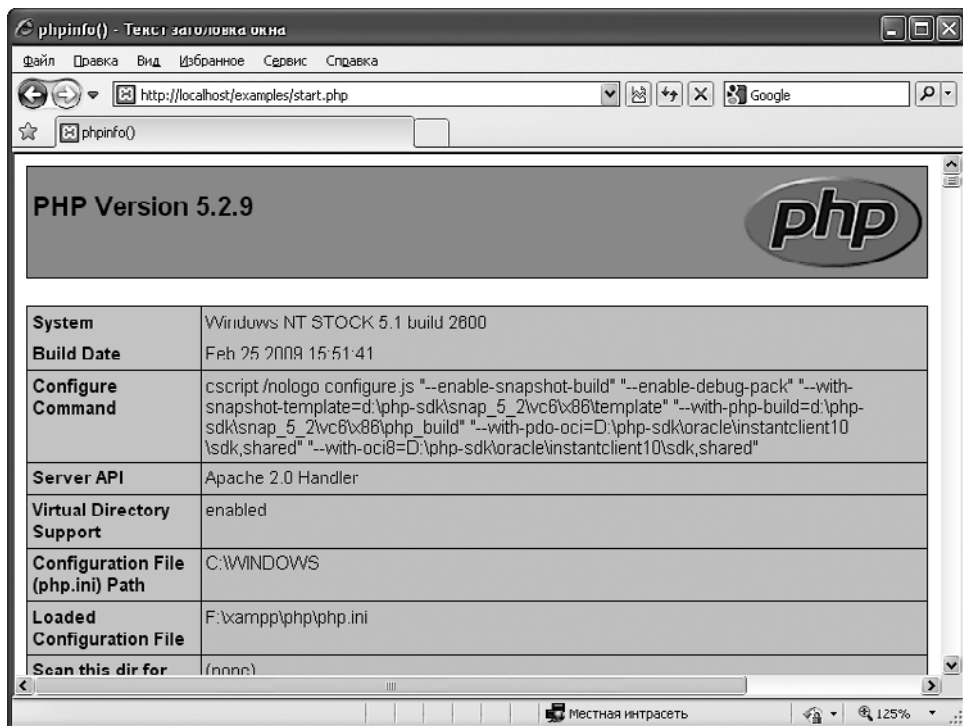


Рис. 1.1. Окно со сведениями об установленной версии языка PHP

Запись `phpinfo()` внутри сценария означает вызов встроенной функции PHP — одной из многих, которые этот язык предлагает разработчикам сценариев. Запомните, что в конце любого оператора языка PHP должна стоять точка с запятой (;). Функция `phpinfo()` отображает сведения об установленных компонентах языка PHP и настройках.

Обратите внимание на остальные записи внутри файла. Строки `<?php` и `?>` соответствуют начальному и конечному тегу PHP-кода, внутри которых помещаются сценарии, внедряемые в код HTML веб-страницы. Посмотрим, как это делается на практике. Откройте новый документ текстового процессора и запишите в него такой код (листинг 1.1).

Листинг 1.1. Документ HTML с кодом PHP

```
<html>
<head>
<title>
```

Эта страничка содержит код PHP


```
</title>
<body>
    <h1>
        Внедрение сценария PHP в код HTML
    </h1>
    <?php
        echo "Привет, Мир! Это PHP-сценарий внутри странички Web!";
    ?>
</body>
</html>
```

Сохраните этот код в файле `hello.php` и поместите его в папку `examples`. Введите в адресную строку браузера вызов сохраненного кода `http://localhost/examples/hello.php`, после чего в окне браузера отобразится результат его исполнения (рис. 1.2).

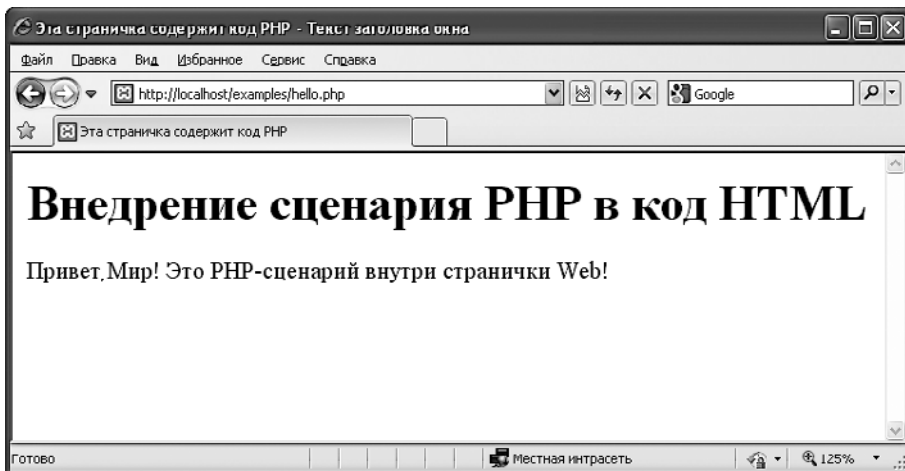


Рис. 1.2. Приветствие PHP-сценария внутри HTML-кода

Рассмотрим содержимое листинга 1.1 внимательнее. Мы видим, что внутри тела документа HTML в содержимое тега `<h1>` помещен заголовок, после которого внутри тегов `<?php` и `?>` расположен оператор `echo` с текстом приветствия. Оператор `echo` — один из основных в языке PHP, он позволяет отображать любой текст внутри веб-страницы. В листинге 1.1 показан простейший пример использования этого оператора:

```
echo "Здесь помещается выводимый текст";
```

Выводимый текст должен быть заключен либо в одинарные кавычки (апострофы), либо в двойные кавычки. Числа могут отображаться без кавычек, например корректно писать так:

```
echo 872976;
```

Вы можете включать в выводимую строку теги HTML, которые будут восприниматься браузером как обычный HTML-код. Например, код:

```
echo "<b>Этот текст</b> выделен полужирным начертанием";
```

приведет к выделению строки `Этот текст` полужирным начертанием.

В операторе `echo` можно выводить несколько строк по отдельности, например запись:

```
echo "Привет", "это", "PHP";
```

приведет к выводу текста «ПриветэтоPHP». Чтобы отделить слова друг от друга, помещайте внутри кавычек пробелы:

```
echo "Привет", "это", "PHP";
```

Если вы хотите отобразить текст в кавычках, пишите так:

```
echo "Петя сказал: \"Я пошел в школу\"";
```

В результате отобразится:

```
Петя сказал: "Я пошел в школу"
```

Иначе говоря, обратный слеш перед кавычками экранирует, предотвращает их интерпретацию языком PHP как служебного символа.

1.4. Комментарии внутри кода PHP

Любой программист при создании сценария оставляет в нем свои замечания, или комментарии, которые должны информировать пользователя программы о функции и назначении определенного фрагмента кода. Не следует пренебрегать этой возможностью, так как без комментариев вам придется все время вспоминать, что и как вы реализовали в своем сценарии, и тратить на это силы и время (а программные коды забываются очень быстро). Так что используйте комментарии, они вам очень пригодятся, если вы захотите вернуться к ранее написанным программам.

Поместить в PHP-сценарий комментарий можно несколькими способами. Во-первых, можно заключить комментарий в знаки `/* Текст комментария */`, например, так:

```
<?php
/* Этот сценарий просто отображает сведения
   о версии и конфигурации языка PHP
   в окне браузера */

    phpinfo();
?>
```

Такие комментарии допустимо располагать в нескольких строках. Учтите, что вложенные сценарии недопустимы. Такого рода запись приведет к ошибке:

```
<?php
/* Этот сценарий просто отображает сведения
/* о версии и конфигурации языка PHP */
   в окне браузера */
    phpinfo();
?>
```

Здесь строка:

```
в окне браузера */
```

не будет распознана как комментарий, поскольку начальная метка комментария `/*` завершится в конце второй строки меткой `*/`.

Есть и еще два способа размещения комментариев, но только находящихся в одной строке. Их следует помещать после символов `//` или `#`, например:

```
<?php
// Этот сценарий просто отображает сведения
// о версии и конфигурации языка PHP
// в окне браузера
    phpinfo(); # Это встроенная функция PHP вывода информации о PHP
?>
```

Писать или нет комментарии? И если да — то сколько? Общее мнение таково — много комментариев не бывает. Настоятельно рекомендую не пренебрегать ими. В идеале каждая функция и значимый оператор сценария должны быть закомментированы.

1.5. Переменные

Все данные, с которыми работает сценарий, хранятся в переменных. Переменные — это контейнеры данных, обозначенные определенным идентификатором. В языке PHP идентификатор каждой переменной начинается со знака доллара (\$). Переменная создается после того, как ей присваивается какое либо значение:

```
$message= "Здравствуй, мир!";  
$counter=10;  
$amount=3.62;
```

Переменные могут хранить данные разных типов — целочисленные, вещественные, логические, текстовые и др., что мы и видим в вышеприведенном примере. В отличие от других языков программирования, в PHP не требуется явно указывать тип данных, поскольку он определяется автоматически. Однако если вы используете в одном выражении переменные разных типов, могут возникнуть трудности. Например, в коде:

```
$temperature="0";  
$temperature=$temperature+2;
```

к текстовой переменной `$temperature`, хранящей символ "0" (ноль), добавляется число 2. В этом случае выполняется неявное преобразование типов. Чтобы явно определить преобразование типов, следует писать:

```
$int_variable=(integer)$variable;
```

то есть указать в скобках нужный тип данных. Вот список допустимых типов данных PHP:

- ☐ `boolean` — логический тип со значениями `true` или `false`;
- ☐ `string` — текстовые данные;
- ☐ `integer` — целые числа;
- ☐ `float` — вещественные числа;
- ☐ `object` — объект;
- ☐ `array` — массив;
- ☐ `resource` — ресурс;
- ☐ `NULL` — пустое значение.

Рассмотрим эти типы данных подробнее (кроме типа `object` — ему целиком отведена глава 7).

1.6. Типы данных PHP

Далее будут перечислены все типы данных и приведены варианты их использования на простых примерах. Пока мы опустим описание типа `object`, поскольку такие данные используются для создания сценариев методами объектно-ориентированного программирования, которое мы подробно обсудим в главе 7.

boolean

Логический тип данных позволяет переменной иметь два значения: `true` — истина и `false` — ложь. Для определения переменной этого типа данных ей следует присвоить одно из указанных значений:

```
<?php
$variable = false; // присвоить $variable значение false
?>
```

Обычно логические типы данных используются для конструкций, в которых операторы или функции проверяют исполнение какого-либо условия и возвращают результат в виде булевой переменной. Например, в такой конструкции:

```
<?php
    if (MyFunction() == true) {
        echo "Функция MyFunction выполнена успешно";
    }
?>
```

Здесь оператор `if` проверяет значение, возвращаемое функцией `MyFunction()`, и, если оно равно `true`, сообщает об успешности выполнения функции.

string

Строковые переменные хранят текстовые данные любой длины. Длина строки ограничена только размером свободной оперативной памяти.

Строки обрабатываются с помощью набора функций, предоставляемых PHP, либо к любому символу строки можно обратиться непосредственно операторами языка. Ниже представлен пример строковой переменной:

```
<?php
$string = "Пример текста, хранимого строковой переменной";
```

```
echo $string;
?>
```

Функция `echo` отображает содержимое строки `string` в окне браузера.

integer

Этот тип предназначен для хранения целочисленных данных разрядностью 32 бита. Значения этих чисел лежат в пределах от $-2\,147\,483\,648$ до $2\,147\,483\,647$. Они могут иметь знак $+$ или $-$ и записываться с использованием десятичной, восьмеричной и шестнадцатеричной системы счисления.

- При использовании восьмеричной системы счисления перед значением пишется символ `0` (ноль), например `$variable=0735`.
- Если вы используете шестнадцатеричную систему, то должны предварять число символами `0x`, например `$variable=0xA37`.

float

Этот тип данных предназначен для хранения вещественных чисел. Максимальное значение вещественного числа зависит от платформы, на которой исполняется сценарий, и обычно ограничено числом с 14 десятичными знаками. Вещественные числа записываются в одном из ниже представленных форматов:

```
<?php
$variable1 = 1.264;
$variable2 = 1.3e3;
$variable3 = 3E-10;
?>
```

array

В любом языке программирования массивами называются упорядоченные наборы данных, в которых для каждого хранимого значения указан соответствующий индекс, или ключ. Индекс используется для идентификации элемента, и между значениями индекса и элементом массива установлено взаимно однозначное соответствие.

Массивы могут быть по-разному организованы, в зависимости от того, как организовано соответствие между ключом и индексом. Например, в простейшем массиве каждый индекс — это целое число от `0` и выше:

```
<?php
```

```
$array[0]="Диван";  
$array[1]="Кресло";  
$array[2]="Стол";  
$array[3]="Стул";  
?>
```

Здесь `$array` обозначает имя массива, а 0, 1, 2, 3 — его индексы.

Массивы могут быть ассоциативными, в которых индексами служат строки, например:

```
<?php  
$array["Имя"]="Иван";  
$array["Отчество"]="Николаевич";  
$array["Фамилия"]="Семенов";  
?>
```

Здесь ключами массива служат названия пунктов анкетных данных, а значениями — соответствующие сведения.

Массивы могут быть многомерными. Синтаксис элементов многомерного простого массива таков:

```
$имя_массива[индекс_1][индекс_2]...[индекс_N];
```

Например, вы можете определить многомерный массив так:

```
<?php  
$array[0][0]="Диван";  
$array[0][1]="Кровать";  
$array[1][0]="Кресло";  
$array[1][1]="Тумбочка";  
$array[1][2]="Стол";  
$array[2][0]="Стул";  
$array[2][1]="Шкаф";  
$array[2][2]="Комод";  
?>
```

Чтобы создать массив, можно воспользоваться PHP-функцией под названием `array()`. Вот пример:

```
<?php  
$array["Семенов"] = array("Возраст"=>"34", "Имя"=>"Игорь", "Отчество"=>"Онуфриевич");
```

```
$array["Козлов"] = array("Возраст"=>"44", "Имя"=>"Егор", "Отчество"=>"Тимо  
феевич");  
$array["Федотов"] = array("Возраст"=>"32", "Имя"=>"Степан",  
"Отчество"=>"Лукич");  
?>
```

Здесь с каждой фамилией сопоставлены данные анкеты по трем пунктам: Возраст, Имя, Отчество. Метка `=>` в вышеприведенном примере как раз и задает соответствие между индексом массива и значением элемента.

resource

Тип данных `resource` содержит ссылку на внешний ресурс, который будет далее использован в сценарии. Например, это может быть ссылка на файл или запись в базе данных. Для создания данных этого типа используются специальные функции, которые мы еще обсудим в дальнейшем.

NULL

Тип `NULL` также называется пустым типом данных и говорит о том, что переменной еще не установлено значение. Переменная будет иметь значение `NULL`, если оно было ей присвоено, либо при исполнении сценария переменной еще не было присвоено какое-либо значение, либо значение переменной было удалено с помощью специальной функции `unset()`.

1.7. Константы в PHP

Часто для работы сценария необходимо использовать какой-либо параметр, с помощью которого понадобится выполнять расчеты или который нужно будет отображать в каких-то информационных сообщениях и т. д. Классический пример — число π , с помощью которого вычисляется длина окружности. Можно, конечно, везде использовать его значение в виде вещественного числа, но это крайне неудобно, особенно если требуется обеспечить точность и нужно все время писать много знаков после десятичной точки: 3.1415926535. В этом случае можно просто определить константу `pi` с помощью такого оператора:

```
define("pi", 3.1415926535);
```


После этого можно во всех выражениях использовать `pi`. Ее значение будет автоматически подставляться в код программы. Константы могут хранить данные только логического, вещественного, целого и строкового типа.

Общий формат функции `define()` таков:

```
define (имя_константы, значение константы, $case_sensitive);
```

Здесь `$case_sensitive` — это переменная логического типа, определяющая чувствительность имени константы к регистру: значение `true` делает ее чувствительной, `false` — нет.

В листинге 1.2 представлен сценарий подсчета длины окружности и площади круга с помощью определенной константы `pi`.

Листинг 1.2. Расчет длины окружности и площади круга

```
<?php
define("pi", 3.1415926535, true);
$r=10;
$lenght=2*pi*$r;
echo "Длина окружности радиуса ", $r, " равна ", $lenght;
$square=pi*pow($r, 2);
echo "<br>Площадь круга радиуса   ", $r, " равна   ", $square;
?>
```

Результат исполнения сценария представлен на рис. 1.3.

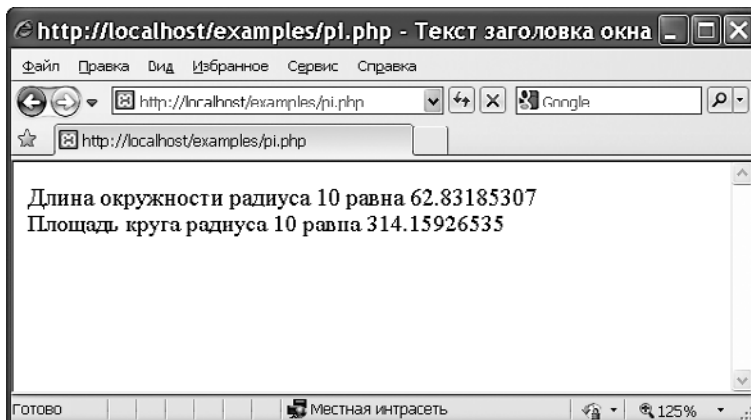


Рис. 1.3. Вычисление длины окружности и площади круга

При работе с константами следует учесть, что они определяются только функцией `define ()`, могут быть определены и будут доступны в любом месте сценария.

Язык PHP предоставляет множество предопределенных констант, которые можно использовать в своих сценариях. Набор этих констант обусловлен конфигурацией той среды, которую вы используете для работы. Для проверки наличия константы существует функция `defined ()`, возвращающая `true`, если константа объявлена.

1.8. Операторы PHP

Язык PHP содержит множество операторов, то есть средств, с помощью которых обрабатываются данные в программе. Ниже мы обсудим основные операторы языка, выполняющие различные функции при построении программного кода: присвоение значений различным переменным, математические операции с числами, обработку текстовых и битовых данных, операторы, управляющие ходом выполнения программы.

Операторы присвоения

Основной оператор присвоения данных любой переменной представляет собой знак равенства `=` и используется так:

Левый_операнд=Правый_операнд;

Здесь оператор присвоения получает значение правого операнда и устанавливает его в левый операнд. Таким образом, результатом исполнения кода:

```
<php?
    $variable1=5;
    $variable2=variable1;
?>
```

будет присвоение переменным `$variable1` и `$variable2` значения 5.

Операцию присвоения можно использовать для присвоения значения сразу нескольким переменным:

```
$var1=$var2=$var3=$var4=2;
```

Это очень удобно и значительно упрощает код программы.

Математические операторы

Математические операторы позволяют выполнять различные арифметические вычисления с переменными и числами. К ним относятся операторы:

- ☐ суммы двух чисел $+$;
- ☐ разности двух чисел $-$;
- ☐ произведения двух чисел $*$;
- ☐ частного от деления двух чисел $/$;
- ☐ остатка от деления двух чисел $\%$.

Примеры применения этих операторов представлены в листинге 1.3.

Листинг 1.3. Применение математических операторов

```
<?php  
echo "7 + 3 = ", 7 + 3, "<br>";  
echo "7 - 3 = ", 7 - 3, "<br>";  
echo "7 * 3 = ", 7 * 3, "<br>";  
echo "7 / 3 = ", 7 / 3, "<br>";  
echo "7 / 3 = ", 7 % 3, "<br>";  
?>
```

Результат исполнения кода представлен на рис. 1.4.

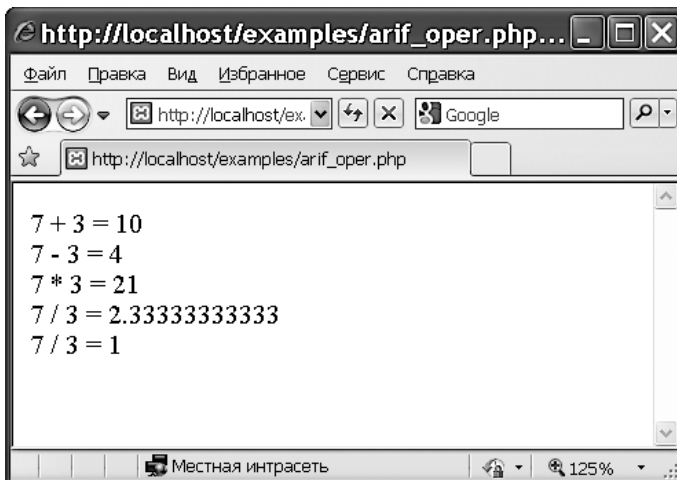


Рис. 1.4. Пример исполнения арифметических операторов PHP

Обратите внимание на операцию деления. Ее результатом всегда будет значение вещественного типа, даже если оба операнда были целочисленными. При использовании арифметических операций возможно использование скобок, как в обычных арифметических выражениях. Приоритет выполнения арифметических операций над другими соответствует обычным математическим правилам.

Комбинированные операторы присвоения

При работе с арифметическими операторами в PHP можно использовать комбинированные операторы присвоения, в которых значение левого операнда применяются в арифметической операции, а затем результат сохраняется в той же переменной. Например, операция:

```
$variable += 21;
```

означает добавление к переменной `$variable` числа 21 и сохранение результата в самой переменной `$variable`, то есть вышеприведенная операция эквивалентна такой:

```
$variable = $variable + 21;
```

Ниже приведены примеры комбинирования оператора присвоения с различными арифметическими операторами:

```
$variable-=24;    // Вычитание из $variable числа 24
```

```
$variable*=5;     // Умножение $variable на 5
```

```
$variable/=3;     // Деление $variable на 3
```

Строковые операторы

Для работы со строковыми типами данных язык PHP предоставляет оператор конкатенации (`.`), объединяющий в одну строку свои операнды, и оператор присвоения с конкатенацией, присоединяющий правый операнд к левому. Вот примеры их использования.

```
<?php
$variable1 = "Здравствуй ";
$variable2 = $variable1 . "мир!";
?>
```

В этом примере переменной `$variable2` присваивается строка "Здравствуй, мир!". Это же самое можно сделать с использованием комбинированной операции присвоения и конкатенации.

```
<?php
$variable1 = "Здравствуй, ";
$variable1 .= "мир!";
?>
```

Здесь к переменной `$variable1` присоединяется строка "мир!". Результат будет одинаков в обоих случаях.



ПРИМЕЧАНИЕ

Для работы со строковыми переменными PHP предоставляет множество встроенных функций, которые значительно упрощают обработку полученных текстовых данных.

Операторы инкремента и декремента

Эти операции увеличивают или уменьшают на 1 значение переменной. Например, выражение `++$variable` увеличивает значение `$variable` на 1 с последующим присвоением результата переменной `$variable`, а выражение `--$variable` уменьшает на 1 значение `$variable`. Это очень удобно при организации счетчиков, применяемых, например, для создания циклических операций (см. подраздел «Операторы циклов» раздела 1.8).

Операции инкремента и декремента бывают *префиксные* и *постфиксные*. Префиксные операции записываются в начале переменной, а постфиксные — в конце. Разница между ними проявляется при использовании инкрементируемой или декрементируемой переменной. При префиксной операции сначала выполняется инкрементация или декрементация переменной, а затем эта переменная используется в соответствующем выражении. В постфиксной операции в выражении используется первоначальное значение переменной, а ее инкрементация или декрементация выполняется после.

В листинге 1.4 показаны способы применения операций инкрементации и декрементации в обоих вариантах.

Листинг 1.4. Постфиксное и префиксное выполнение инкремента и декремента

```
<?php
$var1=$var2=$var3=$var4=2;
echo "\$var1++ = ", $var1++, "<br>";
```

```

echo "++\${var2} = ", ++${var2}, "<br>";
echo "\${var3}-- = ", ${var3}--, "<br>";
echo "--\${var4} = " , --${var4}, "<br>";
?>

```

Результат выполнения сценария листинга 1.4 представлен на рис. 1.5.

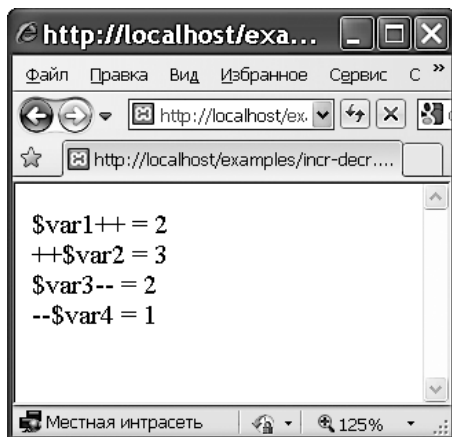


Рис. 1.5. Префиксная и постфиксная инкрементация и декрементация

Учтите, что булевы типы данных не могут быть инкрементированы или декрементированы.

Битовые операторы

Битовые операторы позволяют устанавливать значения 1 или 0 бит в двоичном представлении целочисленных переменных. Целые числа в PHP представлены 32-разрядными переменными. Вот пример такого числа:

```
0000 0000 0000 0000 0000 0000 0000 0101
```

В десятичном представлении это число 5. Иногда возникает необходимость установить в 1 или же сбросить в 0 бит в каком-то двоичном разряде или группе разрядов. Для этого и служат двоичные переменные, которые позволяют сравнивать биты в двоичных разрядах двух переменных, скажем `$var1` и `$var2`, и по результатам этого сравнения устанавливать соответствующий бит в возвращаемом значении. Рассмотрим эти операторы.

- `&` — побитовое «И», устанавливает в результате `var1 & var2` только те биты, которые установлены и в `$var1`, и в `$var2`.

- ❑ `|` — побитовое «ИЛИ», устанавливает в `var1 | var2` биты, которые установлены или в `$var1`, или в `$var2`.
- ❑ `^` — исключающее «ИЛИ», устанавливает в `var1 ^ var2` биты, которые установлены или только в `$var1`, или только в `$var2`.
- ❑ `~` — отрицание, устанавливает в `~$var1` биты, которые в `$var1` не установлены, и наоборот.
- ❑ `<<` — выражение `$var1 << $var2` сдвигает влево все биты переменной `$var1` на `$var2` позиций влево.
- ❑ `>>` — выражение `$var1 >> $var2` сдвигает вправо все биты переменной `$var1` на `$var2` позиций вправо.

Эти операции на практике редко применяются.

Операторы сравнения

Операторы сравнения представляют собой множество логических операторов, предназначенных для использования в других операторах для управления их работой. Самым популярным оператором такого рода является условный оператор `if`, используемый для разветвления процесса исполнения программы в зависимости от различных условий. Вот его общий вид:

```
if (логическое_выражение)
    оператор
```

В этом примере логическое выражение может принимать значение `true` или `false`, в зависимости от текущих значений входящих в него переменных. Если логическое выражение внутри `if` истинно, то исполняется оператор, стоящий сразу после `if`.

В логическое выражение могут входить многие операторы, включая функции, возвращающие логические значения. Но чаще всего в нем используются операторы сравнения. Например, выражение `$x < $y` истинно, если значение `$x` меньше `$y`. Вот какие операторы сравнения имеются в PHP:

- ❑ `==` (равенство), выражение `$x == $y` истинно, если значение `$x` равно `$y`;
- ❑ `===` (идентичность, или тождество), выражение `$x === $y` истинно, если значение `$x` равно `$y` и они одного типа;
- ❑ `!=` (неравенство), выражение `$x != $y` истинно, если `$x` не равно `$y`;

- ❑ `<>` (неравенство), выражение `$x != $y` истинно, если `$x` не равно `$y`;
- ❑ `!==` (неидентичность), выражение `$x !== $y` истинно, если или `$x` не равно `$y` или они не одного типа;
- ❑ `<` (меньше), выражение `$x < $y` истинно, если `$x` меньше `$y`;
- ❑ `>` (больше), выражение `$x > $y` истинно, если `$x` больше `$y`;
- ❑ `<=` (меньше или равно), выражение `$x <= $y` истинно, если `$x` меньше или равно `$y`;
- ❑ `>=` (больше или равно), выражение `$x >= $y` истинно, если `$x` больше или равно `$y`.

Приведу пример использования операторов сравнения:

```
<?php
    $temperature=26;
    if($temperature>=24)
        echo "На улице тепло"
?>
```

Операторы сравнения разрешается применять только для скалярных переменных, но не для массивов и объектов.

Логические операторы

Эти операторы также позволяют управлять ходом вычислений в программе, делая его максимально гибким и интуитивно понятным. Например, они разрешают объединять вместе два условных оператора:

```
if(логическое_выражение_1&&логическое_выражение_2)
    оператор
```

Здесь оператор после `if` выполняется, если истинно И логическое выражение_1, И логическое выражение_2. Таким образом, комбинируя логические выражения и операторы, можно строить весьма сложные конструкции, реализующие любую, самую изощренную, логику работы программы.

Вот какие логические операторы имеются в PHP:

- ❑ **AND** (логическое «И»), выражение `$x AND $y` истинное, если и `$x`, и `$y` истинны;

- ❑ OR (логическое «ИЛИ»), выражение $\$x \text{ OR } \y истинно, если или $\$x$, или $\$y$ истинно;
- ❑ XOR (исключающее «ИЛИ»), выражение $\$x \text{ XOR } \y истинно, если истинно или $\$x$, или $\$y$, но не оба;
- ❑ ! (отрицание), выражение $!\$x$ истинно, если $\$x$ ложно;
- ❑ && (логическое «И»), выражение $\$x \text{ \&\& } \y истинно, если и $\$x$, и $\$y$ истинны;
- ❑ || (логическое «ИЛИ»), выражение $\$x \text{ OR } \y истинно, если или $\$x$, или $\$y$ истинно.

Возникает вопрос, зачем нужны совпадающие по функциям операторы AND и && или OR и ||. Дело в том, что операторы && и || имеют более высокий приоритет операций. Обсудим этот вопрос подробнее, он очень важен при создании в программе сложных управляющих конструкций.

Приоритеты операторов

Приоритет операторов языка PHP представлен в табл. 1.1. При знакомстве с ней помните, что операторы с более высоким приоритетом выполняются в первую очередь.

Таблица 1.1. Приоритет операторов

Уровень приоритета	Оператор
1	=, +, -, *, /, %, >>=, <<=, &=, ^=, =
2	
3	&&
4	
5	^
6	&
7	!=
8	<<=, >>=
9	<<, >>
10	+—
11	*, /, %
12	++, --

В любом случае использование скобок позволяет решить все проблемы — в PHP правила приоритета операторов вполне соответствуют школьной арифметике.

Условные операторы

Условные операторы в языке PHP применяются для управления ходом исполнения программы. Выше, при описании операторов сравнения, я упомянул самый популярный оператор `if`, но, кроме него, PHP предлагает и другие условные операторы.

Оператор `if`

Синтаксис оператора `if` в общем случае таков:

```
if (логическое_выражение)
{
    оператор1;
    ...
    оператор2;
}
```

Если логическое выражение в операторе `if` истинно, то выполняются операторы, следующие за конструкцией `if`. Если таких операторов должно быть несколько, то их помещают в фигурные скобки. Приведу пример использования оператора.

```
<?php
if ($x < $y) {
    echo "x меньше y";
    $z=$y;
    $y = $x;
    $x=$z;
}
?>
```

Здесь выполняется сравнение двух переменных — `$x` и `$y`, и если `$x` меньше `$y`, переменной `$x` присваивается значение `$y`, а `$y` — значение `$x`.

Конструкция `if...else`

Если необходимо, как это часто бывает, совершить в программе некоторые действия в случае как выполнения, так и НЕвыполнения определенных условий, используется конструкция `if...else`. Общий вид этой конструкции таков:

```
if (логическое_выражение)
    оператор_1;
```

```
else  
    оператор_2;
```

Здесь при истинности логического выражения в операторе `if` выполняется оператор_1, а иначе — оператор_2. Если исполняемых операторов должно быть несколько, то их заключают в фигурные скобки:

```
<?php  
    if ($x > $y) {  
        echo "x меньше y";  
        $z=$y;  
        $y=$x;  
        $x=$z;  
    } else  
        echo "y больше или равно x";  
?>
```

Конструкцию `if...else` можно записать и так:

```
if (логическое_выражение_1):  
    набор_операторов_1;  
elseif(логическое_выражение_2):  
    набор_операторов_2;  
else:  
    набор_операторов_3;  
endif
```

В этой конструкции после каждого блока `if...else` вместо фигурных скобок ставится двоеточие, после которого располагаются наборы операторов. Обратите внимание на новую конструкцию `elseif`, расширяющую возможности оператора `if`. Общий синтаксис конструкции `elseif` таков.

```
if (логическое_выражение_1) {  
    набор_операторов_1;  
} elseif (логическое_выражение_2) {  
    набор_операторов_2;  
} elseif(логическое_выражение_3) {  
    набор_операторов_3;  
}
```

Описанные выше конструкции позволяют эффективно разветвлять процесс исполнения кода программы в зависимости от условий, но ими возможности PHP не ограничиваются.

Оператор switch

Если программе следует выбрать варианты действий в зависимости от значений какого-либо выражения, не обязательно логического, то вместо оператора `if` удобнее использовать конструкцию `switch-case`. Синтаксис конструкции `switch-case` таков:

```
switch(выражение) {
case значение_выражения_1:
    набор_операторов_1;
    [break;]
case значение_выражения_2:
    набор_операторов_2;
    [break;]
. . .
case значение_выражения_N:
    набор_операторов_N;
    [break;]
[default: операторы_выполняемые_по_умолчанию;
[break]]
}
```

В приведенной выше конструкции в операторе `switch` вычисляется выражение и полученный результат сравнивается с параметрами значения_выражения каждого блока `case`. Если значения совпадают, то выполняется набор операторов данного блока `case`. Если ни одно значение_выражения в блоках `case` не совпадет с результатом подсчета выражения, выполняются операторы блока `default`.

Нужно учесть одну тонкость: после исполнения операторов блока `case` выполнение сценария переходит на следующий блок, потом на следующий и так далее до конца оператора `switch` либо пока не встретится оператор `break`. Приведу пример конструкции `switch-case`.

```
<?php
switch ($x) {
case 0:
```

```
case 1:
case 2:
    echo "x меньше 3<br>";
    break;
case 3:
    echo "x=3<br>";
    break;
case 4:
    echo "x=4<br>";
    break;
case 5:
    echo "x=5<br>";
    break;
case 6:
    echo "x больше 5<br>";
}
?>
```

Как видим, здесь набор операторов `case` для значений 0, 1 пуст, но если значение `$x` равно 0 или 1, управление перейдет соответственно на блок `case 0:` или `case 1:`, после чего будет выполнен оператор третьего блока `case 2:` и отобразится сообщение `x меньше 3`.

Операторы циклов

Циклы — это следующая по значимости категория PHP-операторов после условных. Они предназначены для повторения исполнения определенного набора операторов по заданному условию. Повторяемые операторы образуют тело оператора цикла, а повторяемые проходы по этим операторам и представляют собой цикл.

В PHP имеются четыре вида операторов циклов.

- ❑ `for` — цикл со счетчиком.
- ❑ `while` — цикл с предусловием.
- ❑ `do..while` — цикл с постусловием.
- ❑ `foreach` — цикл перебора массивов.

Рассмотрим эти операторы по порядку.

for

Оператор `for` позволяет выполнить операторы тела цикла заданное количество раз. Синтаксис оператора цикла `for` таков:

```
for (операторы_инициации_цикла; условие_цикла; операторы_после_выполнения_цикла) {
    тело_цикла;
}
```

Вот пример такого оператора:

```
<?php
    for(x=2, $i=1; $i<12; i++) {
        x +=2;
        echo "<br> i = ", $i, " x = ", $x;
    }
?>
```

В этом примере в начале цикла инициализируется значение счетчика `$i` и переменной `$x`, после чего начинается выполнение цикла (обратите внимание, что операторы инициализации цикла разделяются запятыми). Порядок действий после инициализации таков.

1. Проверяется условие цикла `$i<12`, то есть НЕдостижение переменной цикла `$i` значения 12.
2. Если условие цикла истинно, выполняется тело цикла, иначе итерации завершаются.
3. Далее выполняются операторы после исполнения цикла `$i++`.
4. Переходим на шаг 1.

Если при задании цикла требуется ввести несколько операторов инициализации цикла, проверки условий цикла или операторов после цикла, то их можно разделить запятыми, как в приведенном выше примере.

do...while

Оператор цикла `while` также проверяет значение выражения *после* каждого прохода, поэтому, в отличие от `for`, тело цикла `while` выполняется хотя бы один раз. Синтаксис цикла `while` таков:

```
do {
```

```
        тело_цикла;
    }
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. Вот пример применения цикла с постусловием `do...while`:

```
<?php
$i = 1;
do {
    echo "<br> i = ", $i;
} while ($i++< 10);
?>
```

Здесь перед выполнением цикла счетчику `$i` присваивается начальное значение, после чего цикл выполняется десять раз.

while

Синтаксис оператора цикла `while` таков:

```
while (логическое_выражение) {
    набор_операторов
}
```

Оператор цикла `while` называется циклом *с предусловием*, поскольку работает следующим образом.

1. Вычисляется значение логического выражения.
2. Если значение истинно, выполняется тело цикла, в противном случае переходим на следующий за циклом оператор.

Иными словами, в отличие от ранее описанного оператора `do...while` проверка логического условия проводится в начале цикла. Приведу пример цикла с предусловием `while`:

```
<?php
$i=0;
while ($i++<12) {
    echo "<br> i = ", $i;
}
?>
```

Важно учитывать последовательность выполнения операций условия `$i++<12` с постфиксной инкрементацией. Сначала проверяется условие, и только потом увеличивается значение переменной. Если операция инкремента префиксная, то есть записывается как `++$i<12`, то сначала будет выполнено увеличение переменной, а только затем сравнение.

foreach

Оператор `foreach` предназначен специально для перебора массивов. Синтаксис оператора `foreach` таков:

```
foreach (массив as $ключ=>$значение) {
    набор_операторов
}
```

Вот пример работы цикла `foreach`:

```
<?php
$fruits["бананы"] = "30 руб.";
$fruits["ананасы"] = "50 руб.";
$fruits["яблоки"] = "20 руб.";
$fruits["апельсины"] = "30 руб.";
foreach ($fruits as $key => $value) {
    echo "<br> Фрукт: ", $key, " Цена: ", $value;
}
?>
```

В примере циклически просматривается массив `$fruits` и для каждого элемента выводится пара `$key` и `$value`, содержащая соответственно название фрукта и его цену:

```
Фрукт: бананы Цена: 30 руб.
Фрукт: ананасы Цена: 50 руб.
Фрукт: яблоки Цена: 20 руб.
Фрукт: апельсины Цена: 30 руб.
```

break

Чтобы прервать выполнение цикла в ходе любой итерации, не дожидаясь ее завершения, используется оператор `break`. С этой целью его помещают в нужное место внутри тела цикла, например:

```
<?php
$i=1;
```



```
while ($i++<9) {  
    if ($i==3) break;  
    echo "<b> Итерация", $i";  
}  
?>
```

Здесь при достижении `$i` значения 3 цикл прерывается.

Если используются вложенные циклы, то можно указать номер прерываемого цикла. Нумерация циклов выглядит следующим образом:

```
for (...) // Третий цикл  
{  
    for (...) // Второй цикл  
    {  
        for (...) // Первый цикл  
        {  
        }  
    }  
}
```

Чтобы прервать выполнение какого-то вложенного цикла, следует использовать такую конструкцию:

```
break(номер_цикла);
```

По умолчанию для номера цикла используется 1, то есть выполняется выход из текущего цикла, но иногда применяются и другие значения.

continue

Оператор `continue` немедленно завершает текущую итерацию цикла и при выполнении условия цикла переходит к новой итерации. Для `continue` можно указать уровень вложенности цикла, который будет продолжен после возврата 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 определенное количество объектов и выбрать из них только те, которые удовлетворяют заданным условиям. Приведу пример использования конструкции `continue`:

```
<?php
    $i=-3;
    while ($i++< 5) {
        if ($i==0) continue;
        echo " 1/ $i = ", 1/$i, "<br>";
    }
?>
```

Здесь в цикле отображается обратное значение переменной цикла `1/$i` и оператор `continue` позволяет отбросить деление на нуль. В результате отобразится следующее:

```
1/ -2 = -0.5
1/ -1 = -1
1/ 1 = 1
1/ 2 = 0.5
1/ 3 = 0.333333333333
1/ 4 = 0.25
1/ 5 = 0.2
```

Как видим, некорректная арифметическая операция отменена.

1.9. Работа с функциями в PHP

В PHP, как и в любом языке программирования, существуют подпрограммы, то есть фрагменты кода PHP, оформленные так, что к ним можно обратиться из любого места программы. Это позволяет значительно упростить программу за счет исключения повторяющихся кодов, улучшить ее читабельность, а также сократить объем памяти, необходимый для хранения и исполнения программы.

В PHP подпрограммы называются пользовательскими функциями и обладают некоторыми особенностями, отличающими их от подобного рода средств в других языках программирования.

- ❑ Функции PHP возвращают любой тип данных.
- ❑ Имеется возможность изменять параметры, переданные функции.
- ❑ Поддерживаются параметры по умолчанию, вы можете вызывать одну и ту же функцию с переменным количеством параметров.

Кроме того, имеются отличия в видимости, то есть доступности переменных в различных частях программного кода. Этот вопрос мы подробно обсудим чуть позже, а сейчас посмотрим, как в PHP задаются пользовательские функции.

Синтаксис функций PHP

Синтаксис объявления функций таков:

```
function имя_функции (аргумент1[=значение1], ..., аргумент1[=значение1])  
{  
    тело_функции  
}
```

В начале объявления функции стоит служебное слово `function`, после указано имя функции и далее в скобках через запятую перечислены аргументы функции. Каждому аргументу может быть присвоено значение по умолчанию. Далее находится тело функции, содержащее список операторов, заключенный в фигурные скобки.

Функцию можно объявить в любом месте программы, но обязательно до момента ее первого использования. Имена функций в общем случае могут содержать русские буквы, но этого делать не рекомендуется — возможны проблемы совместимости. Имена функций не должны содержать пробелов и должны быть уникальными. Регистр символов при этом не учитывается.

Пользовательские функции могут возвращать значения любого типа. Для этого используется оператор `return`, который может вернуть что угодно, включая массивы. Если этот оператор опустить, то функция не вернет никакого значения.

Приведу пример функции.

```
<?php  
function user_function() {  
    $return_value = 666;  
    return $return_value;  
}  
$x = user_function();  
echo "<br>x=   ", $x;  
?>
```

Здесь функция `user_function` с помощью оператора `return` возвратит число 666.

Передача аргументов

Если при объявлении функции был указан список аргументов, то этой функции при вызове можно передавать параметры. Например, пусть функция объявлена так:

```
<?php
function user_function ($x, $y, $z) {
    ...
}
?>
```

Тогда при вызове функции `user_function()` обязательно нужно указать все передаваемые параметры.

Аргументы можно передавать по ссылке. Для этого перед именем аргумента в определении функции следует поместить амперсанд (&).

```
<?php
function user_function(&$string_argument) {
    $string_argument = 'ПОСЛЕ ВЫЗОВА';
}
$string = 'ДО ВЫЗОВА ';
echo "До вызова функции: ", $string;
user_function($string);
echo "<br> После вызова: ", $string;
?>
```

В результате будет выведено:

```
До вызова функции: ДО ВЫЗОВА
После вызова: ПОСЛЕ ВЫЗОВА
```

Передав функции ссылку на аргумент, мы позволили ей изменить значение переменной `$string` внутри функции. Сравните, что получилось бы, если бы в определении функции мы убрали передачу значения по ссылке (удалили амперсанд).

```
function user_function($string_argument) {
```

```
$string_argument = 'ПОСЛЕ ВЫЗОВА';  
}
```

В результате отобразится:

До вызова функции: ДО ВЫЗОВА

После вызова: ДО ВЫЗОВА

Получается, функция не смогла изменить значение переданного параметра! Такого рода ошибки — частое явление при написании программ, приводящее к проблемам при отладке кода. Дело в том, что все переменные, объявленные внутри функции, существуют только во время ее исполнения и их видимость ограничена телом функции. Изменив значение такой переменной внутри функции, мы никак не повлияли на значение одноименной переменной вне функции. Передача параметра по ссылке решает эту проблему — изменение параметра, переданного по ссылке, внутри функции, приводит к изменению соответствующей переменной вне функции.

Функции также могут возвращать ссылки как результат своей работы. В определении такой функции в начале ее названия ставится амперсанд (&). Рассмотрим пример такой конструкции:

```
function &reference_ret(&$reference)  
{  
    return $reference;  
}
```

Здесь функция `&reference_ret()` просто возвращает ссылку на переданный ей параметр. Вот пример ее использования:

```
$var=7;  
echo "Начальное значение: ", $var;  
$reference = &reference_ret($var);  
$reference -=3;  
echo "<br>Новое значение: ", $var;
```

В результате отобразится:

Начальное значение: 7

Новое значение: 4

Таким образом, мы получили из функции `&reference_ret()` ссылку `$reference` на переменную `$var` и с помощью этой ссылки изменили значение самой переменной (вычли 3). Ссылки указывают на места памяти, хранящие переменную,

и работать с ними следует осторожно — получив такую ссылку, мы можем изменять значение соответствующей переменной из любого места программы.

Значения параметров по умолчанию

Если при задании функции указать для аргументов значения по умолчанию, то можно опускать передачу параметров. В этом случае будут использованы значения по умолчанию. Приведу пример.

```
<?php
    function sale_fruit($argument = "банан") {
        return "Дайте мне $argument";
    }
echo "<br>". sale_fruit();
echo "<br>". sale_fruit("яблоко");
?>
```

При исполнении программы отобразятся две строки:

```
Дайте мне банан
Дайте мне яблоко
```

Первая строка отображает значение аргумента по умолчанию, а вторая — переданное значение. Учтите, что значение по умолчанию должно быть константой.

Видимость переменных

Видимость переменной определяет, в каком месте программы мы можем обратиться к ней для выполнения своих операций. В PHP с точки зрения видимости переменные подразделяются на *глобальные* и *локальные*.

- ❑ Глобальные переменные доступны из любого места программы, включая функции.
- ❑ Локальные переменные определяются внутри функции и доступны только из нее.

В зависимости от способа и места своего определения для каждой переменной задается ее видимость. Для определения видимости переменных, то есть их доступности из различных частей сценария, в PHP используются следующие правила.

- ❑ В сценариях PHP применяются суперглобальные переменные, видимые в любом месте сценария.

- ❑ После своего объявления константы видны везде, как внутри, так и вне функций.
- ❑ Переменные, объявленные в сценарии, по умолчанию являются глобальными, которые будут видны везде, но не внутри функций.
- ❑ Переменные, объявленные внутри функций, являются локальными для своих функций и прекращают свое существование после завершения работы функции.
- ❑ Переменные, объявленные глобальными внутри функций, ссылаются на глобальные переменные с теми же именами.
- ❑ Переменные, объявленные статическими внутри функций, невидимы за пределами функций, но сохраняют значения между вызовами функций.

Поясню вышесказанное примерами. По умолчанию все используемые внутри функции переменные локальны и вы не можете изменить их вне тела функции. Даже если ее имя совпадает с именем глобальной переменной, объявленной где-то в другом месте программы, вне функции, то никакие изменения локальной переменной не скажутся на глобальной.

Например, рассмотрим такую программу:

```
<?php
$var = 45;
function myfunction() {
    $var = 54;
    echo "<br> внутри функции var = ", $var;
}
myfunction();
echo "<br> извне функции var = ", $var;
?>
```

Она отобразит следующее:

```
внутри функции var = 54
извне функции var = 45
```

Чтобы переменная была доступна из любого места программы, ее следует объявить глобальной с помощью такой конструкции:

```
global $variable1, $variable 2... ;
```

Рассмотрим пример использования глобальной переменной.

```

<?php
$a = 6;
$b = 8;
function myfunc() {
    global $a, $b, $c;
    $c = $a + $b;
}
myfunc();
echo "<br> c = ", $c;
?>

```

Программа этого примера выведет `c = 14`. После определения внутри функции `myfunc()` переменных `$a`, `$b` и `$c` глобальными все обращения к этим переменным будут указывать на глобальную версию. Не существует никаких ограничений на количество глобальных переменных, которые могут обрабатываться пользовательскими функциями.

В сценариях PHP, кроме локальных и глобальных переменных, можно использовать *статические* переменные. Для этого переменную внутри функции с помощью ключевого слова `static` объявляют статической, в результате чего ее значения не будут изменяться после вызова функции. Ниже приведен пример работы функции, реализующей счетчик, в котором статическая переменная накапливает количество обращений к функции.

```

function mycounter() {
    static $counter;
    $counter = $counter + 1;
    echo $counter;
}
for ($i = 5; $i<12; $i++) mycounter();

```

Программа в данном примере выведет строку:

```
1234567
```

При каждом вызове функции статическая переменная `$counter` сохраняет значение, полученное при предыдущем вызове. Если удалить объявление статической переменной, то отобразится строка:

```
1111111
```


Причина в том, что в этом случае при каждом новом вызове функции `mycounter()` переменные внутри функции будут инициализироваться, а переменным присваивается значение нуль. Далее в нашем примере это значение инкрементируется и выводится оператором `echo`.

Суперглобальные переменные введены в PHP как место хранения данных системного характера, определяющих работу всего сценария на системном уровне. Перечислю эти переменные, так как мы будем постоянно сталкиваться с ними в дальнейшем.

- ❑ `$GLOBAL` — массив всех глобальных переменных. С его помощью можно получить доступ к любой глобальной переменной внутри функции, например `$GLOBAL['myglobalvar']`.
- ❑ `$_SERVER` — массив параметров работы сервера.
- ❑ `$_GET`, `$_POST` — массивы параметров, переданных в сценарий, например, при обработке формы.
- ❑ `$_COOKIE` — массив данных, сохраняемых в файлах cookie.
- ❑ `$_REQUEST` — массив введенных пользователем данных, включая информацию в массивах `$_GET`, `$_POST`, `$_COOKIE`.
- ❑ `$_FILES` — массив данных о загруженном файле.
- ❑ `$_ENV` — массив переменных окружения.
- ❑ `$_SESSION` — массив переменных сеанса работы пользователя со сценарием.

Вышеперечисленные суперглобальные переменные будут встречаться на протяжении всей книги, и мы подробно рассмотрим их назначение и применение. А пока запомните, что все переменные, объявленные в сценарии вне функций, по умолчанию являются глобальными, внутри функций — локальными, а суперглобальные переменные доступны в любой части сценария.

1.10. Операторы повторного использования кода

Кроме функций, язык PHP предлагает еще одно полезное средство повторного использования кода — операторы `include()` и `require()`. С их помощью вы можете включить в сценарий любой код, который будет исполняться так, как если бы он был помещен в исходный код сценария обычным образом. Поясню примером.

Допустим, у вас имеется простейший сценарий:

```
<?php
echo '<p>Это исходный код сценария</p>';
?>
```

При исполнении отобразится текст `Это исходный код сценария`. Теперь предположим, что нам требуется включить в свой сценарий несколько констант, которые предполагается использовать во многих местах, например число π . Мы можем просто определить константу:

```
define("pi", 3.14159);
```

и далее использовать ее в коде сценария. Но можно сделать и так — записать в отдельный файл, скажем `constants.inc`, такой сценарий:

```
<?php
define("pi", 3.14159);
?>
```

Далее в наш исходный код сценария добавляем следующее:

```
<?php
echo '<p>Это исходный код сценария</p>';
include('constants.inc');
?>
```

Тогда наш сценарий будет исполнен так, как если бы он содержал такой код:

```
<?php
echo '<p>Это исходный код сценария</p>';
define("pi", 3.14159);
?>
```

Таким образом, мы получаем возможность включить в свои сценарии код, хранящийся в различных файлах. Для чего это нужно? Конечно, в вышеприведенном примере это не играет особой роли, но если такой повторно используемый код велик по объему, например содержит множество констант, библиотечных функций и пр., а веб-приложение состоит из множества файлов (что чаще всего и бывает), то использование такой техники весьма полезно. Вы можете в одном файле собрать какой-то общий для всех фрагмент кода и далее включить его во все эти файлы одним оператором. Это улучшает читаемость кода и упрощает внесение исправлений в повторно используемые фрагменты — это можно делать в одном месте.

Для того чтобы можно было использовать такой добавляемый код, он должен быть заключен в дескрипторы `<?PHP ... ?>`, иначе PHP не воспримет его как код сценария и исполнять не будет. Расширение имени включаемого файла для PHP не имеет значения, но лучше присваивать таким файлам однообразное расширение, например `.inc`.

В PHP имеется четыре оператора для включения в сценарий повторно используемых фрагментов кода: `include()`, с которым мы только что познакомились, `require()`, `include_once()` и `require_once()`. Отличие `require()` от `include()` состоит только в том, что при сбое исполнения `require()` выдает неисправимую ошибку, а `include()` лишь сообщает о нештатной ситуации. Операторы `require_once()` и `include_once()` отличаются от `require()` и `include()` только тем, что исполняются всего один раз. Это очень важно, поскольку позволяет избежать повторной загрузки кода, что чревато ошибкой в случае, например, повторной загрузки библиотечных функций.

1.11. Резюме

Итак, вы познакомились с основными синтаксическими конструкциями языка PHP. Конечно, многое осталось вне нашего обсуждения, но теперь вы сможете реализовать практически все возможности, предоставляемые PHP, по обработке данных любого типа. Теперь вы знаете, как объявляются переменные, организуются циклы и ветвления при выполнении сценария, создаются и используются функции.

В следующей главе мы продолжим рассматривать средства языка PHP и научимся создавать настоящие веб-приложения, реализующие различные сервисы Интернета. Мы обсудим организацию сеансов работы со сценарием, создание файлов cookie, передачу данных между сценариями и получение данных из форм, размещенных на страничках сайтов, работу с сервером FTP.

Глава 2

Веб-приложения

В предыдущей главе вы познакомились с синтаксисом языка PHP и освоили основные конструкции из операторов, позволяющие выполнять обработку данных в сценариях. Теперь настала пора применить полученные сведения на практике и на основе полученных знаний построить собственное веб-приложение.

В этой главе вы научитесь передавать в сценарии PHP данные, вводимые пользователем в формы на веб-страницах, обрабатывать эти данные и отображать результаты в окне браузера. Таким образом, вы научитесь превращать статические сайты в динамические, интерактивные веб-приложения, наиболее широко используемые при создании профессиональных сайтов Интернета.

Наконец, вы освоите работу с файлами cookie и научитесь организовывать сеансы работы пользователя с сайтом, что позволит устанавливать авторизованный доступ к его ресурсам с момента регистрации на сайте и открытия сеанса до момента завершения работы, то есть выхода из сеанса.

Таким образом, вы пройдете все этапы построения динамического сайта, который, быть может, будет слишком простым для настоящего профессионального веб-приложения, но научит вас всем принципам программирования соответствующих сценариев PHP.

Начнем с обсуждения работы с формами на веб-страницах.

2.1. Работа с формами

Рассмотрим простейшую веб-страницу, содержащую форму с двумя элементами — полем ввода и кнопкой отправки данных (листинг 2.1).

Листинг 2.1. Простая веб-форма

```
<html>
  <head>
    <title>
      Простая форма
    </title>
  </head>
  <body>
    <center>
      <h1> Пример формы </h1>
      <form method="post" action="simple_form.php">
        Укажите свое имя
        <input name="name" type="text">
        <br>
        <input type="submit" value="OK">
      </form>
    </center>
  </body>
</html>
```

Сохраните код этого листинга в файл `simple_form.html` в папке `examples` и откройте страницу в окне браузера, введя в адресную строку `http://localhost/examples/simple_form.html` (рис. 2.1).

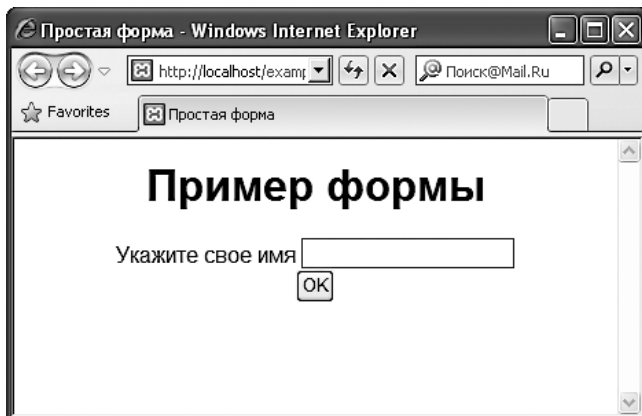


Рис. 2.1. Простая форма для ввода имени

В этом примере при нажатии кнопки ОК формы будет выполнен запрос сценария `simple_form.php`, указанного атрибутом `ACTION` формы, и данные, введенные пользователем, будут переданы сценарию PHP для обработки. Как же сценарий получит эти данные? Для этого в языке PHP предусмотрены так называемые суперглобальные массивы, к которым вы можете обратиться из любого места сценария. В данном случае используются массивы `$_GET` и `$_POST`, содержащие передаваемые данные при использовании соответственно методов `GET` и `POST` (этот метод указывается атрибутом `METHOD`). Вы можете также использовать суперглобальный массив `$_REQUEST`, включающий данные из обоих массивов. Вот как мы используем его в сценарии обработки нашего запроса (листинг 2.2).

Листинг 2.2. Сценарий обработки простой формы

```
<html>
  <head>
    <title>
      Сценарий обработки простой формы
    </title>
  </head>
  <body>
    <center>
      <h1> Чтение данных, введенных пользователем </h1>
      Вы ввели имя:
      <?php
        echo $_POST["name"];
      ?>
    </center>
  </body>
</html>
```

Опять-таки сохраните код сценария в файле `simple_form.php` в папке `examples`, откройте в браузере файл `simple_form.html`, введите имя в поле формы и запустите сценарий, нажав кнопку ОК (см. рис. 2.1). Результат представлен на рис. 2.2.

Как видите, введенное имя благополучно передалось в сценарий обработки.

Таким же образом можно передать в сценарий данные формы любой сложности. Сейчас мы займемся этим, но сначала одно замечание. Как вы могли заметить из приведенного выше примера, мы построили веб-приложение из двух файлов — один содержал форму, другой — сценарий ее обработки. Однако это не единствен-

ный и не самый распространенный способ написания веб-приложений. Общепринятый способ состоит в объединении таких файлов, то есть во включении в единый файл HTML-кода для ввода данных и сценария для их последующей обработки. В листинге 2.3 представлен сценарий, объединяющий код из листингов 2.1 и 2.2.

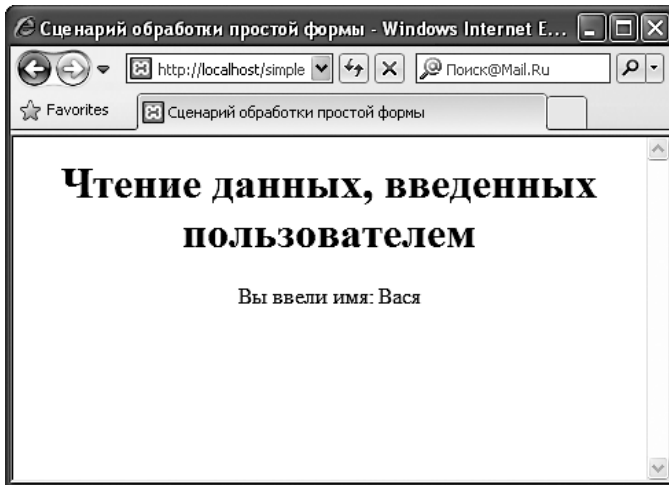


Рис. 2.2. Результат обработки формы

Листинг 2.3. Веб-приложение в едином файле

```
<html>
  <head>
    <title>
      Сценарий обработки простой формы
    </title>
  </head>
  <body>
    <center>
      <?php
// Этот оператор if управляет выполнением нашего веб-приложения
        if(isset($_REQUEST["name"]))
        {
            ?>
                <h1> Чтение данных, введенных пользователем </h1>
                Вы ввели имя:
```

```

    <?php
// Это оператор из листинга 2.2 для отображения введенных в форму данных
        echo $_POST["name"];
    }
    else
    {
        ?>
// Это форма из листинга 2.1 для ввода данных
    <h1> Пример формы </h1>
    <form method="post" action="simple_form_script.php">
        Укажите свое имя
        <input name="name" type="text">
        <br>
        <input type="submit" value="OK">
    </form>
    <?php
    }
    ?>
    </center>
</body>
</html>

```

Несмотря на свой устрашающий вид, сценарий весьма прост и содержит тот же самый HTML-код из листинга 2.1, в который с помощью тегов `<?php` и `?>` включены операторы PHP, обрабатывающие введенные данные. А чтобы браузер знал, когда ему следует отображать форму ввода данных, а когда — результаты обработки введенных данных, в сценарий включен условный оператор:

```
if(isset($_REQUEST["name"]))
```

Функция `isset()` в условном выражении оператора `if` определяет существование переданной веб-приложению переменной: если переменная отсутствует, она возвращает значение `false`, иначе — `true`.

На первом шаге работы веб-приложения, до ввода данных пользователем своего имени, элемент массива `$_POST["name"]` не определен и функция возвращает `false`. При этом исполнение сценария переходит на блок `else`, содержащий форму ввода. После ввода данных в поля формы и нажатия кнопки **OK** вызывается повторное исполнение веб-приложения, но теперь переменная `$_POST["name"]`

определена и исполняются операторы, стоящие после `if`, которые просто отображают введенное имя.

Как видите, все просто и эффективно и мы избежали повторного ввода HTML-кода в обоих файлах — документе HTML и сценарии PHP. Вы можете сохранить этот код в файле `simple_form_script.php` в папке `examples` и после открыть в браузере вводом адреса `http://localhost/example/simple_form_script.php`. Результат будет аналогичным представленному на рис. 2.1, 2.2.

А теперь приведу пример более сложной формы, включающей в себя все определенные в языке HTML элементы (листинг 2.4).

Листинг 2.4. Веб-приложение в едином файле

```
<html>
  <head>
    <title>
      Первое веб-приложение
    </title>
  </head>
  <body>
    <center>
      <?php
// Этот оператор if управляет выполнением нашего веб-приложения
        if(isset($_POST["login"]))
        {
          ?>
            <h1> Вы ввели: </h1>
          <?php
// Это операторы для отображения введенных в форму данных
            echo "<br> Логин: ", $_POST["login"],
              "<br> Пароль: ", $_POST["password"],
              "<br> Ваши увлечения: ", $_POST["preferences"],
              "<br> Ваш любимый вид спорта: ", $_POST["sport"],
              "<br> Вы занимаетесь спортом: ", $_
POST["sportsmen"],
              "<br> Вы хотите получать рассылку: ", $_
POST["message"];
```

```

        }
    else
    {
        ?>
<! Это форма для ввода данных !>
<h1> Пример формы </h1>

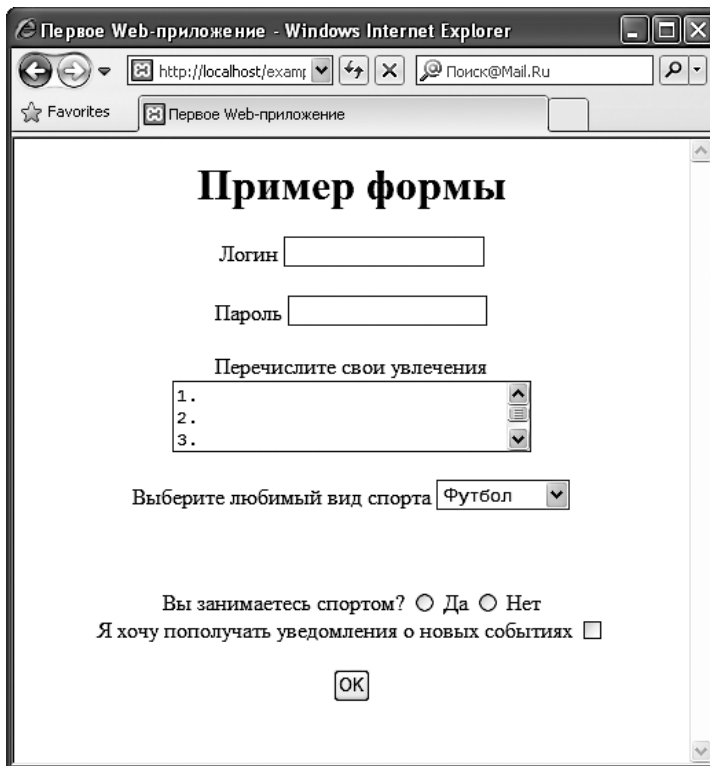
<form method="post" action="complex_form.php">
    Login
    <input name="login" type="text">
    <br><br>Пароль
    <input name="password" type="password">
    <br><br>Перечислите свои увлечения <br>
    <textarea name="preferences" rows="3" cols="30">

1.
2.
3.

    </textarea>
    <br><br>Выберите любимый вид спорта
    <Select name="sport">
        <option>Футбол</option>
        <option>Баскетбол</option>
        <option>Волейбол</option>
    </select>
    <br><br><br><br>Вы занимаетесь спортом?
    <input name="sportsmen" type="radio" value="Да"> Да
    <input name="sportsmen" type="radio" value="Нет"> Нет
    <br> Я хочу получать уведомления о новых событиях
    <input name="message" type="checkbox" value="Да">
    <br><br>
    <input type="submit" value="OK">
</form>
<?php
}
```

```
?>
</center>
</body>
</html>
```

Сохраним, как и ранее, этот сценарий в файл `complex_form.php` в папке `examples` и откроем в браузере вводом адреса `http://localhost/examples/complex_script.php`. Результат представлен на рис. 2.3.



Первое Web-приложение - Windows Internet Explorer

http://localhost/exam... Поиск@Mail.Ru

Первое Web-приложение

Пример формы

Логин

Пароль

Перечислите свои увлечения

1.

2.

3.

Выберите любимый вид спорта

Вы занимаетесь спортом? ☐ Да ☐ Нет

Я хочу получать уведомления о новых событиях ☐

Рис. 2.3. Форма ввода данных в наше первое веб-приложение

Введя нужные данные, нажмите кнопку `OK` и запустите наше веб-приложение. Сценарий будет исполнен, и в окне браузера отобразятся введенные данные (рис. 2.4).

Вы сами сможете разобраться в работе сценария. По сути, он делает одну и ту же операцию со всеми данными, полученными из формы: извлекает из массива `$_POST` значения для элементов формы и отображает их оператором `echo`, как установлено в листинге 2.2.

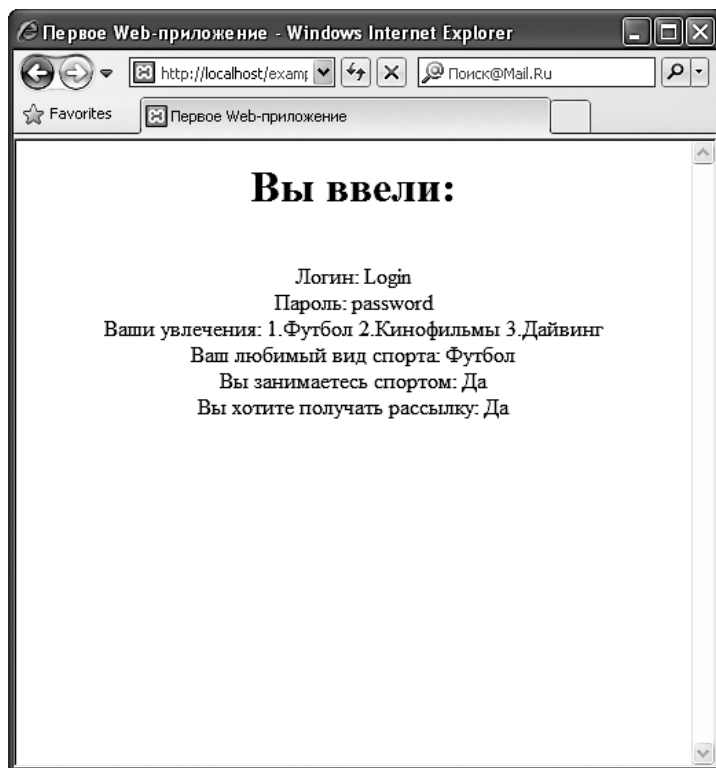


Рис. 2.4. Сценарий отобразил полученные из формы данные

Немного сложнее в веб-приложения загружаются файлы. Остановимся на этом вопросе отдельно.

2.2. Загрузка и обработка файлов

Для загрузки в веб-приложение файлов используется следующий тег HTML:

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

Он помещается в форму для особого типа данных, определяемого атрибутом `ENCTYPE="multipart/form-data"`, как это сделано в листинге 2.5.

Листинг 2.5. Форма для загрузки файлов

```
<html>
  <head>
    <title>
```

```
    Форма загрузки файлов
</title>
</head>
<body>
    <center>
        <h1> Загрузка файла </h1>
        <form
            enctype="multipart/form-data"
            method="post"
            action="form_file.php">
            Укажите загружаемый файл
            <input name="myfile" type="file">
            <br>
            <input type="submit" value="Загрузить">
        </form>
    </center>
</body>
</html>
```

Сохраним его в файле `form_file.html` в папке `examples` и откроем в браузере (рис. 2.5).

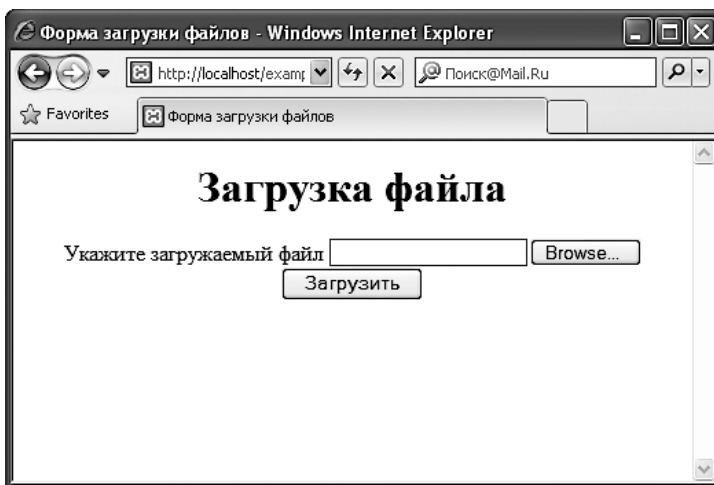


Рис. 2.5. Форма загрузки файла

Как видим, форма принимает имя загружаемого файла и при нажатии кнопки подтверждения обращается к сценарию `form_file.php` (листинг 2.6).

Листинг 2.6. Сценарий обработки загруженного файла

```
<html>
  <head>
    <title>
      Сценарий обработки загруженного файла
    </title>
  </head>
  <body>
    <center>
      <h1> Отображение данных файла </h1>
      Загруженный файл содержит такие данные <br>
      <?php
        $handle=fopen($_FILES["myfile"]["tmp_name"], "r");
        while (feof($handle)==0)
        {
          $file_content=fgets($handle);
          echo $file_content, "<br>";
        }
        fclose($handle);
      ?>
    </center>
  </body>
</html>
```

Вот как он работает. После успешной загрузки файла он сохраняется в каталог `tmp_name` сервера, а имя файла сохраняется в суперглобальном массиве `$_FILES["имя_поля_формы_с_именем_файла"]["tmp_name"]`. Поле формы для ввода имени файла в листинге 2.5 таково: `<input name="myfile" type="file">`. Поэтому имя нашего файла хранится в ячейке массива `$_FILES["myfile"]["tmp_name"]`.

Чтобы получить содержимое файла, сначала его нужно открыть, то есть определить переменную для доступа к этому ресурсу. Эта операция делается с помощью функции `fopen()` с такими основными (обязательными) параметрами:

```
resource fopen ( string filename, string mode)
```

Здесь `filename` — это имя файла, а `mode` — режим работы с файлом. Заданный в нашем случае режим `"r"` разрешает доступ к файлу, значение `"w"` устанавливает режим записи в файл. Функция `fopen()` возвращает так называемый дескриптор файла, имеющий тип данных `resource` и позволяющий получить доступ к данным в файле.

Итак, создадим в Блокноте текстовый файл `MyFile.txt`, сохраним в папке `examples`, выберем его в форме загрузки файла и нажмем кнопку **Загрузить**. В ответ наше веб-приложение загрузит файл и приступит к его обработке. Вот какой код для этого используется в листинге 2.6:

```
$handle=fopen($_FILES["myfile"]["tmp_name"], "r");
while (feof($handle)==0)
{
    $file_content=fgets($handle);
    echo $file_content, "<br>";
}
fclose($handle);
```

Первый оператор вызывает функцию `fopen()`, которая открывает файл, а в переменную `$handle` сохраняется дескриптор файла. Далее с помощью цикла `while` мы читаем содержимое файла порцию за порцией, вызывая на каждой итерации функцию `fgets()`. Общий синтаксис этой функции таков:

```
string fgets ( resource handle [, int length])
```

Здесь `handle` — дескриптор файла, а `length` равно количеству байтов, читаемых при вызове функции `fgets()`. Таким образом, функция `fgets()` возвращает строку длиной `"length-1"` (считая первый разряд нулевым); по умолчанию `length` равно 1024. Чтобы завершить цикл по прочтении файла, используется функция `feof()`, равная `true` при чтении конца файла и `false` в противном случае.

В результате веб-приложение отобразит такой результат (рис. 2.6).

Таким образом, наше приложение заработало — мы прочли файл и сумели отобразить его содержимое в окне браузера. С помощью других функций РНР мы могли бы сохранить полученные данные на сервере, чтобы впоследствии обращаться к ним по мере надобности. Но сохранение данных в файле — не лучший, хотя и приемлемый, выбор. Гораздо эффективнее использовать для этого базы данных, которые мы рассмотрим в следующей главе.

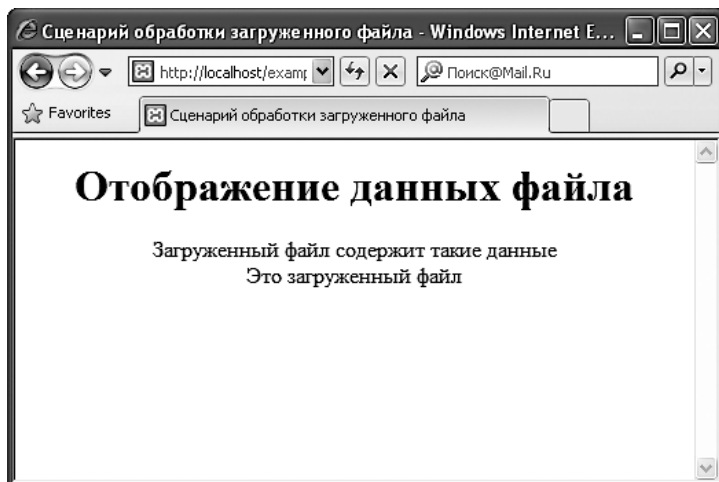


Рис. 2.6. Результат чтения файла MyFile.txt

2.3. Сеансы

Веб-приложение, как правило, состоит из нескольких страниц, содержащих сценарии. При загрузке каждой страницы переменные в этих сценариях очищаются, и для обмена данными между страницами можно использовать механизм передачи параметров, или базу данных, или запись информации в файлах cookie. Однако наиболее удобный метод — использование сеансов.

При обращении посетителя сайта к веб-приложению происходит открытие сеанса и для его работы на сервере формируется определенная среда, заданная совокупностью параметров, специфических для конкретного пользователя. Например, там могут сохраняться какие-то настройки интерфейса, конфигурации отдельных сервисов и т. п. Соответствующие данные хранятся в суперглобальном массиве `$_SESSION`, например, в нем можно сохранить выбранный цвет фона страницы:

```
$_SESSION['background'] = "зеленый";
```

В дальнейшем при переходе от одной страницы к другой вы сможете использовать этот параметр для установки нужного цвета фона.

Каждому сеансу присваивается уникальный идентификатор, который может быть использован при повторном открытии сеанса. Этот идентификатор можно, например, сохранить в файле cookie, после чего при повторном посещении сайта вам не потребуется заново настраивать работу с его интерфейсом и сервисами.

Для обращения к данным сеанса используется функция `session_start()`, не имеющая параметров. После ее исполнения становится доступным массив `$_SESSION`, и с его помощью вы можете сохранять и извлекать свои данные, которые будут доступны из любой части веб-приложения до завершения сеанса. В листинге 2.7 приведен пример использования сеансов для создания счетчиков посещений страницы. Вы не сможете сделать этого с помощью переменных сценария, поскольку они при каждом новом обращении обнуляются, но сохранение значения счетчика в массиве `$_SESSION` решает эту проблему.

Листинг 2.7. Счетчик показов

```
<?php
    session_start();
?>
<html>
    <head>
        <title> Использование сеансов </title>
    </head>
    <body>
        <center>
            <h1> Счетчик показов страницы </h1>
        <?php
            if(isset($_SESSION['counter'])==false)
                $_SESSION['counter']=0;
            $_SESSION['counter']++;
            echo "Эту страницу открывали ", $_SESSION['counter'], "
раз";
        ?>
        </center>
    </body>
</html>
```

Сценарий в листинге достаточно прост. При каждом обращении к странице (скажем, нажатием клавиши F5) выполняется функция инициализации сеанса `session_start()`, после чего происходит либо инициализация счетчика, если это первое открытие страницы, либо инкрементация. На рис. 2.7 показан результат исполнения этого сценария.

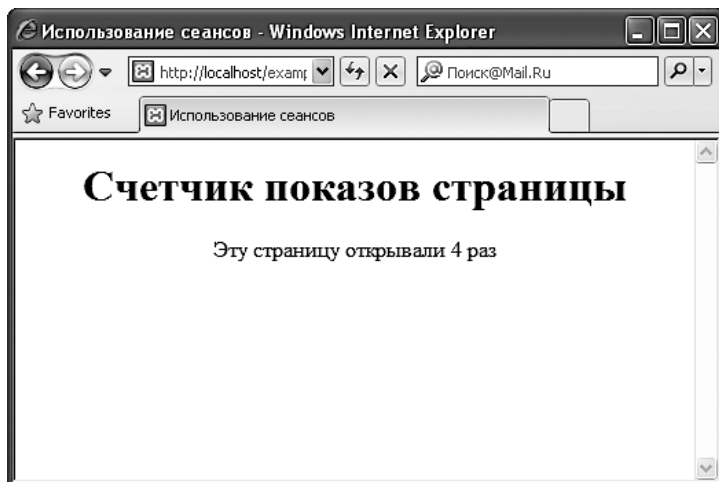


Рис. 2.7. При каждом показе страницы срабатывает счетчик

Иногда возникает потребность удалить данные, сохраненные в сеансе, то есть не просто обнулить или очистить их, а убрать соответствующий параметр из массива `$_SESSION`. Для этого используется функция `unset()`. Например, использованный в предыдущем примере счетчик мы можем удалить так:

```
session_start();  
unset($_SESSION['counter']);
```

Мы уже говорили, что с каждым сеансом сопоставляется идентификатор, позволяющий обратиться к данным, сохраненным во время этого сеанса. Чтобы получить или установить идентификатор сеанса, можно использовать функцию `session_id()`, имеющую такой синтаксис:

```
string session_id ([string id])
```

Эта функция возвращает идентификатор текущей сессии, а если задан параметр `id`, она замещает его значением идентификатора текущего сеанса. В последнем случае функцию `session_id()` следует вызывать до инициализации сеанса, то есть до вызова функции `session_start()`. В строке идентификатора сеанса следует использовать только символы `a-z`, `A-Z` и `0-9`.

Идентификатор сеанса обычно хранится в файле cookie. Однако можно сконфигурировать PHP на передачу его в строке вызова сценария или использовать передачу через скрытое поле формы, имеющее имя `PHPSESSID`. Последний способ полезен в тех случаях, если в клиентском браузере отключено сохранение cookie, а PHP сконфигурирован без поддержки передачи идентификатора через строку вызова.

Но мы не будем отвлекаться на эти методы, а сконцентрируемся на использовании файлов cookie как на наиболее удобном и широко применяемом методе.

2.4. Работа с cookie

Файлы cookie содержат текстовые строки, которые сохраняются на клиентском компьютере и могут быть переданы на сервер во время обращения. Тем самым вы можете сохранить, а потом передать информацию о текущем сеансе работы с веб-сервером из одного сценария в другой, не прибегая к упомянутым выше средствам передачи через формы или в адресной строке. Имейте в виду, что данные в cookie отправляются на сервер в виде заголовков HTTP-запросов, посылаемых сервером на клиентский компьютер. Поэтому вызовы функции установки cookie необходимо помещать до тегов HTML, ведь заголовки HTTP отправляются клиенту до отправки HTML-кода страницы.

Для установления данных в файле cookie используется функция `setcookie()`, имеющая следующий синтаксис:

```
int setcookie (string name [, string value [, int expire [, string path [, string domain [, int secure]]]])
```

Аргумент `name` задает имя cookie, а `value` — его значение. Необязательные аргументы могут быть пропущены, но их придется заменить пустой строкой `""`. Они определяют следующие параметры:

- ❑ `expire` — срок действия cookie;
- ❑ `path` — путь на сервере, в пределах которого cookie доступен;
- ❑ `domain` — домен, на котором доступен cookie;
- ❑ `secure` — указывает на использование cookie только при безопасном соединении.

Покажу на примере, как работает механизм cookie. Создадим страницу с таким кодом записи cookie (листинг 2.8).

Листинг 2.8. Сценарий установки cookie

```
<?php
    setcookie("color", "green");
?>
<html>
    <head>
```

```
<title> Использование cookie </title>
</head>
<body>
  <center>
    <h1> Задание cookie</h1>
    cookie установлен. Содержимое можно посмотреть <a href="read_cookie.php">
здесь </a>
  </center>
</body>
</html>
```

В сценарии устанавливается cookie с названием "color" и значением "green", а значения читаются в сценарии `read_cookie.php`, к которому можно обратиться щелчком на ссылке страницы. Сохраним сценарий в файле под именем `cookie.php` в папке `examples`. Теперь создадим сценарий чтения cookie (листинг 2.9).

Листинг 2.9. Чтение cookie

```
<html>
<head>
  <title> Чтение cookie </title>
</head>
<body>
  <center>
    <h1> Чтение cookie </h1>
    <?php
      if(isset($_COOKIE['color'])!=false)
        echo "cookie имеет значение ", $_COOKIE['color'];
      else
        echo "cookie не установлено";
    ?>
  </center>
</body>
</html>
```

Здесь значения cookie извлекаются из суперглобального массива `$_COOKIE` и отображаются пользователю. Сохраним этот сценарий в файле под именем `read_cookie.php` в папке `examples` и запустим первый из них вводом в адресную строку браузера `localhost/examples/cookie.php`. На отобразившейся странице

(рис. 2.8, *слева*) щелчком на ссылке и исполним сценарий из листинга 2.9. Результат представлен на рис. 2.8, *справа*.

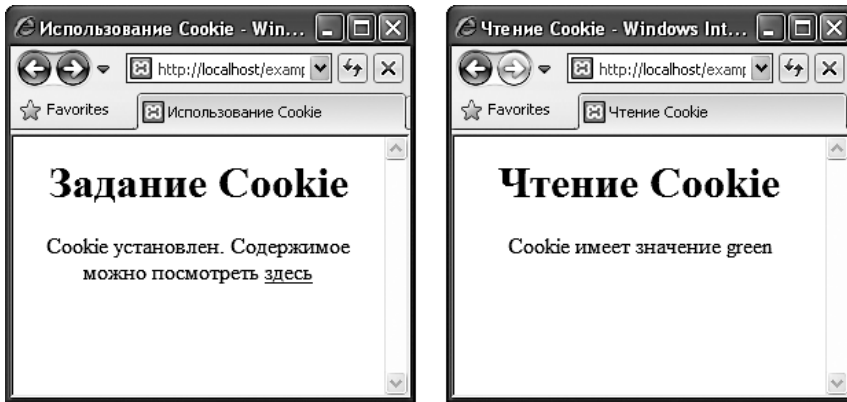


Рис. 2.8. Страницы установки и чтения cookie

Как видим из кода в листинге 2.9, установленное значение cookie извлекается из суперглобального массива `$_COOKIE`. Почему мы не извлекли его прямо в файле `cookie.php`? Да потому, что эти значения передаются в браузер при обращении к веб-серверу, то есть требуют перезагрузки страницы с сервера.

Чтобы удалить установленный cookie, следует вызвать функцию `setcookie()` с теми же параметрами, что и при установке, но с пустой строкой `""` в качестве аргумента `value`. Например, чтобы удалить установленное в листинге 2.8 значение cookie, следует вызвать такую функцию:

```
setcookie("color", "");
```

Cookie можно организовывать в массивы, например, так:

```
setcookie("cookie[1]", "color");  
setcookie("cookie[2]", "data");  
setcookie("cookie[3]", "time");
```

Будет сформирован массив `$_COOKIE["cookie"]`, с которым можно работать обычным образом.

Итак, вы научились работать с cookie и теперь вам ничто не мешает организовывать авторизованные сеансы работы с веб-сервером, передавая данные сеанса через cookie. Вы сами, наверное, убедились в их эффективности, ведь браузер автоматически регистрирует вас на сайтах и устанавливает нужные вам параметры — все это результат применения cookie.

Напоследок обсудим еще одну возможность, обеспечиваемую PHP, — отправку почты. Этот инструмент используется на очень многих сайтах, как на профессиональных, так и на любительских домашних страницах.

Отправка почты. Для отправки почтовых сообщений необходимо включить эту возможность для сервера. Это делается в файле `php.ini`, в котором следует отредактировать раздел `php.ini`:

```
[mail function]
;For Win32 only.
smtp=localhost
; For Win32 only.
sendmail_from=me@example.com
```

Здесь в параметре `smtp` следует указать адрес сервера SMTP, а в `sendmail_from` — обратный адрес отправителя писем. Если вы будете размещать свои сценарии на удаленном сервере, эти настройки вам предоставит ваш провайдер интернет-услуг. Если же вы захотите протестировать работу почтовой службы локально, вам придется развернуть у себя на компьютере серверы SMTP и POP3. Если вы решите воспользоваться пакетом XAMPP, это будет сделано по умолчанию во время его установки, а вам только потребуется запустить работу сервера Mercury с панели управления XAMPP. Далее вы сможете либо создать новый почтовый ящик в клиенте, либо воспользоваться встроенными по умолчанию ящиками, например `postmaster@localhost`.

Отправку писем в сценарии PHP выполняет функция `mail()`, имеющая такой синтаксис:

```
bool mail (string to, string subject, string message [, string additional_
headers [, string additional_parameters]])
```

Здесь параметр `to` содержит адрес получателя почты, `subject` — тему письма, а `message` — само сообщение. Остальные параметры необязательны и применяются для отправки дополнительных заголовков послания (параметр `additional_headers`) и параметров (`additional_parameters`).

Функция `mail()` возвращает значение `true` при успешном принятии письма для доставки и `false` в ином случае (отмечу, что это не означает успешной *доставки* письма конечному получателю).

В листинге 2.10 представлено веб-приложение, выполняющее отправку почты со страницы сайта, руководствуясь введенными в поля формы сведениями.

Листинг 2.10. Отправка почты с веб-страницы

```

<html>
<head>
    <title>
        Веб-приложение отправки писем
    </title>
</head>
<body>
    <center>
        <?php
// Этот оператор if управляет выполнением нашего веб-приложения
        if(isset($_POST["lb"]))
        {
            ?>
                <h1> Отправка почты </h1>
            <?php
                if(!mail($_POST["to"], $_POST["subject"], $_
POST["message"]))
                    echo "Ошибка отправки письма";
                else
                    echo "Письмо отправлено";
            }
            else
            {
                ?>
                <h1> Форма для ввода письма </h1>
                <form method="post" action="mail.php">
                    Кому: <input name="to" type="text"> <br>
                    Тема: <input name="subject" type="text"> <br>
                    <textarea name="preferences" rows="5" cols="40"> </textarea>
                <br>
                    <input name="lb" type="hidden" value="Yes"><br>
                    <input type="submit" value="Отправить">
                </form>
            <?php

```

```

    }
    ?>
</center>
</body>
</html>

```

Сценарий достаточно прост, но обратите внимание, что для управления его исполнением приложением мы использовали скрытое поле формы:

```
<input name="lb" type="hidden" value="Yes">
```

Это поле имеет имя `lb`, и в самом начале исполнения сценария проверяется его значение:

```
if(isset($_POST["lb"]))
...

```

При первом открытии страницы значение `$lb` не определено и будет выполнен блок `else` оператора `if`, который содержит код формы ввода почтового адреса и сообщения. После подтверждения отправки письма сценарию передается значение скрытого поля, теперь равное `"Yes"`, и выполняется функция `mail()`, параметры которой извлекаются из суперглобального массива `$_POST`.

Если вы правильно определите все данные формы, то после нажатия кнопки **Отправить** отобразится сообщение об успешной отправке письма. Подчеркиваю: для этого страница должна быть размещена на удаленном сервере с поддержкой почтового сервиса либо вы должны воспользоваться локальными почтовыми службами, развернутыми на вашем компьютере.

2.5. Работа с FTP

Для работы с файлами, сохраняемыми на сервере, язык PHP поддерживает протокол FTP (*File Transfer Protocol — протокол передачи файлов*). С помощью встроенных в PHP средств вы можете подключиться к серверу FTP, чтобы загрузить/скопировать файлы, удалить или изменить их.

Для подключения к серверу FTP используется такая функция:

```
ftp_connect(string ftp_host [, int port [, int timeout]])
```

Она открывает подключение по протоколу FTP с сервером, адрес которого указан в параметре `ftp_host`. Этот адрес представляет собой доменное имя сервера FTP,

например, `ftp.0fees.net`, не должен включать в себя завершающий слеш и название протокола `ftp://`. Необязательный параметр `port` задает номер порта FTP для данного сервера, по умолчанию используется 21, а параметр `timeout` — это время ожидания исполнения всех обменов данными по сети (умолчание — 90 секунд). При успешном завершении `ftp_connect()` возвращает дескриптор FTP-соединения, иначе — `false`.

После успешного подсоединения к серверу FTP в нем следует авторизоваться, введя входное имя (логин) и пароль. Это выполняет другая функция:

```
ftp_login(resource ftp_stream, string username, string password)
```

Здесь `ftp_stream` означает дескриптор FTP-соединения, полученный от функции `ftp_connect()`, а `username` и `password` — имя и пароль для аутентификации на FTP-сервере. В случае успеха функция возвращает значение `true`, иначе — `false`.

Вот пример подключения к серверу FTP:

```
<?php
    $ftp_stream = ftp_connect("ftp.0fees.net");
    if(ftp_stream) {
        $username = "Vasya";
        $password = "123456";
        $result = ftp_login($ftp_stream, $username, $password);
    }
?>
```

Для работы с ресурсами FTP язык PHP предоставляет множество функций. Перечислю часть из них:

- ❑ `ftp_chdir` — выполняет переход в другой каталог сервера;
- ❑ `ftp_alloc` — определяет объем свободного места на сервере;
- ❑ `ftp_mkdir` — создает каталог на сервере FTP;
- ❑ `ftp_rmdir` — удаляет каталог на сервере FTP;
- ❑ `ftp_nlist` — возвращает список файлов в текущем каталоге;
- ❑ `ftp_chmod` — устанавливает права доступа к файлу по протоколу FTP;
- ❑ `ftp_delete` — удаляет файл на сервере FTP;
- ❑ `ftp_exec` — запускает исполнение программы на сервере FTP;

- ❑ `ftp_fput` — загружает файл на сервер FTP;
- ❑ `ftp_fget` — скачивает файл с сервера FTP;
- ❑ `ftp_close` — закрывает FTP-соединение;
- ❑ `ftp_nb_fput` — загружает данные из открытого файла на FTP-сервер.

Это лишь самые популярные функции для работы с сервером FTP. Рассмотрим примеры их использования на практике.

Загрузка файлов на сервер FTP

Сначала рассмотрим, как загружаются файлы на сервер FTP с помощью указанной выше функции `ftp_put()`.

```
ftp_put(resource ftp_stream, string remote_file, string local_file, int
mode [, int startpos])
```

Здесь `ftp_stream` означает дескриптор FTP-соединения, `remote_file` — путь к месту хранения выгружаемого файла на сервере, `local_file` — полный путь к локальному файлу, `mode` задает символьный или бинарный режим закачки и имеет значение `FTP_ASCII` или `FTP_BINARY` соответственно. В первом случае при передаче файла между разнотипными операционными системами (например, UNIX и Windows) автоматически преобразуются символы завершения конца строки. Параметр `startpos` задает позицию в файле, с которой следует начать загрузку, что требуется при возобновлении загрузки при обрыве связи. В листинге 2.11 приведен пример использования этой функции для загрузки файла на FTP-сервер.

Листинг 2.11. Пример загрузки файла `local_file.txt`

```
<html>
  <head>
    <title>
      Загрузка файла на сервер FTP
    </title>
  </head>
  <body>
    <H1>Загрузка файла на сервер FTP</H1>
    <?php
      $ftp_stream = ftp_connect("ftp.0fees.net ");
      $result = ftp_login($ftp_stream, "Vasya", "123456");
      if(!$result) {
```

```

        echo "Ошибка подключения!";
        exit;
    }
    $result = ftp_put($connect, "remote_file.txt", "local_file.txt",
FTP_ASCII);

    if($result){
        echo "Файл успешно загружен";
    }
    else {
        echo "Ошибка загрузки файла!";
    }
    ftp_close($ftp_stream);
?>
</body>
</html>

```

Здесь локальный текстовый файл `local_file.txt` загружается на сервер FTP и сохраняется под именем `remote_file.txt` в корневой папке. Загрузка выполняется в режиме `FTP_ASCII`, что позволяет сохранить структуру строк текста без изменения и без проблем просматривать файл в операционной системе сервера.

Рассмотрим подробно код сценария.

Сначала мы создали соединение с сервером FTP:

```

$ftp_stream = ftp_connect("ftp.0fees.net ");
$result = ftp_login($ftp_stream, "Vasya", "123456");

```

Далее мы проверяем результат исполнения аутентификации на сервере и в случае неудачи выводим сообщение "Ошибка подключения!":

```

if(!$result){
    echo "Ошибка подключения!";
    exit;
}

```

Если все прошло успешно, выполняется функция:

```

$result = ftp_put($connect, "remote_file.txt", "local_file.txt", FTP_
ASCII);

```

Далее проверяется результат загрузки и выводятся сообщения о результатах проверки. Если все завершилось нормально, отображается сообщение "Файл успешно загружен", иначе — "Ошибка загрузки файла!".

```
if($result){
    echo " Файл успешно загружен";
}
else {
    echo " Ошибка загрузки файла!";
}
```

В завершение соединение FTP закрывается:

```
ftp_close($connect);
```

Загруженный файл теперь доступен на сервере.

Скачивание файла с сервера FTP

Для скачивания файла с сервера FTP на локальный компьютер используется следующая функция:

```
ftp_get(resource ftp_stream, string local_file, string remote_file, int
mode [, int resumepos])
```

Все ее параметры вам знакомы и совпадают с вышеописанной функцией `ftp_put()` загрузки файла, изменилось только направление действия. Функция `ftp_get()` считывает файл `remote_file` на сервере FTP и сохраняет на локальном компьютере под именем и в месте, которые указаны в параметре `local_file` в виде путевого имени.

Приведем пример (листинг 2.12), в котором только что загруженный на FTP-сервер файл `remote_file.txt` будет скачан на локальный компьютер и сохранен под именем `local_file.txt`.

Листинг 2.12. Пример скачивания файла `remote_file.txt`

```
<html>
    <head>
        <title>
            Скачивание файла с сервера FTP
        </title>
    </head>
    <body>
```

```
<H1>Скачивание файла с сервера FTP</H1>
<?php
    $ftp_stream = ftp_connect("ftp.0fees.net ");
    $result = ftp_login($ftp_stream, "Vasya", "123456");
    if(!$result) {
        echo "Ошибка подключения!";
        exit;
    }
    $result = ftp_get($connect, "local_file.txt", "remote_file.txt",
FTP_ASCII);
    if($result){
        echo "Файл успешно скачан";
    }
    else {
        echo "Ошибка скачивания файла!";
    }
    ftp_close($ftp_stream);
?>
</body>
</html>
```

Этот сценарий отличается от приведенного в листинге 2.11 только тем, что вместо функции загрузки файла `ftp_put()` мы использовали функцию скачивания `ftp_get()`.

Аналогичным образом работают и остальные средства FTP — сначала вы устанавливаете соединение с сервером, а потом вызываете нужную функцию, реализующую требуемое действие. По завершении следует закрыть FTP-соединение. Пользуясь набором функций FTP, можно без труда построить собственное хранилище файлов или даже файлообменник и снабдить его всеми инструментами, необходимыми для просмотра и редактирования информации.

2.6. Резюме

Итак, в этой главе мы закрепили теоретические сведения о языке PHP, полученные в главе 1. Теперь вы умеете строить веб-приложения, содержащие формы ввода

данных, запоминать переданную информацию в файле и работать с ней средствами PHP, организовывать сеансы работы с сайтом и сохранять параметры сеанса в cookie, отправлять почту с веб-страницы и загружать файлы по протоколу FTP. Этого багажа вполне достаточно для построения собственных почтовых и файло-обменных сервисов на своем сайте.

Однако для создания настоящего динамического сайта нам не хватает одного важнейшего компонента, без которого не реализуется ни одно профессиональное веб-приложение, — базы данных. Без такого хранилища данных невозможно создать сайт, предоставляющий посетителям личные кабинеты и современные интернет-сервисы, а также отображающий на своих страницах сложноструктурированную и актуальную информацию. Для этого чаще всего используются базы данных MySQL, и в следующей главе мы познакомимся со средствами PHP для работы с ними. Мы рассмотрим, что такое база данных, из чего она состоит и какие основные операции с базами данных выполняются с помощью PHP.

Глава 3

Знакомство с MySQL

В этой главе вы познакомитесь с основными понятиями, необходимыми для построения базы данных MySQL. Для всех операций, которые вам предстоит выполнить, вы получите подробные пошаговые инструкции. Кроме того, все основные действия будут пояснены на примере учебной базы данных, содержащей информацию о клиентах, товарах и заказах некоей торговой компании.

3.1. Что такое MySQL

MySQL — свободно распространяемая система управления базами данных (СУБД), разработанная компанией MySQL AB (www.mysql.com). MySQL имеет клиент-серверную архитектуру: к серверу MySQL могут обращаться различные клиентские приложения, в том числе с удаленных компьютеров. Перечислю важнейшие особенности MySQL, благодаря которым эта программа приобрела популярность.

- ❑ MySQL — СУБД с открытым кодом. Любой желающий может бесплатно скачать программу на сайте разработчика (<http://dev.mysql.com/downloads/>) и при необходимости доработать ее. Благодаря этому существует множество приложений MySQL, созданных и свободно распространяемых сторонними разработчиками. Однако для применения MySQL в коммерческом приложении необходимо приобрести у компании MySQL AB коммерческую лицензированную версию программы.

- ❑ MySQL — кросс-платформенная система. Ее можно использовать практически во всех современных операционных системах, в том числе Windows, Linux, Mac OS, Solaris, HP-UX и др. В этой книге мы рассмотрим работу с MySQL только в Windows.
- ❑ MySQL имеет множество программных интерфейсов (API), благодаря которым к базе данных этой системы могут подключаться приложения, созданные с помощью C/C++, Eiffel, Java, Perl, PHP, Python, Tcl, ODBC, .NET и Visual Studio. Из главы 4 вы узнаете, как обращаться к базе данных MySQL из PHP-, Perl- и Java-приложений.
- ❑ MySQL имеет отличные технические характеристики: многопоточность, многопользовательский доступ, быстродействие, масштабируемость (компания-разработчик приводит пример MySQL-сервера, который работает с 60 тысячами таблиц, содержащими приблизительно 5 млрд строк).
- ❑ MySQL имеет развитую систему обеспечения безопасности и разграничения доступа на основе системы привилегий (см. главу 5).

MySQL представляет собой реляционную СУБД, то есть систему управления *реляционными* базами данных. Поэтому для построения базы данных в MySQL нам потребуются базовые понятия теории реляционных баз данных. Им посвящен следующий раздел.

3.2. Основные сведения о реляционных базах данных

Из этого раздела вы узнаете, как устроена реляционная база данных. Сначала мы рассмотрим таблицы, затем ключевые столбцы, связи между таблицами и, наконец, целостность данных в базе.

Таблицы

Итак, все данные, хранящиеся в реляционной базе данных, представлены в виде таблиц. Каждая таблица имеет имя и состоит из строк и столбцов. На пересечении каждого столбца и строки располагается одно значение.

В качестве примера рассмотрим таблицу, содержащую сведения о клиентах компании (табл. 3.1).

Таблица 3.1. Таблица Customers (Клиенты)

id (идентификатор)	name (имя)	phone (телефон)	address (адрес)	rating (рейтинг)
533	ООО "Кускус"	313-48-48	ул. Смольная, д. 7	1000
534	Петров	7 (929) 112-14-15	ул. Рокотова, д. 8	1500
536	Крылов	444-78-90	Зеленый пр-т, д. 22	1000

Строки таблицы не упорядочены и могут храниться в произвольной последовательности. В таблице не должно быть повторяющихся строк.

Каждый столбец таблицы имеет имя и *тип данных* — этому типу соответствуют все значения в столбце. Так, в нашем примере столбец с именем `id` и столбец с именем `rating` — числовые, а поля `name`, `phone` и `address` — символьные.

По существу, таблица реляционной базы данных представляет собой набор информации об однотипных объектах. При этом каждая строка содержит сведения об одном объекте, а в каждом столбце хранятся значения некоторого атрибута этих объектов. Например, строка с идентификационным номером (`id`) 533 содержит информацию об объекте, у которого атрибут `name` имеет значение 'ООО "Кускус"', атрибут `phone` (телефон) — значение '313-48-48' и т. д.

Первичный ключ

Поскольку строки таблицы не упорядочены, то у них нет порядковых номеров и отличить строки одну от другой можно только по содержащимся в них значениям. В связи с этим возникает понятие *первичного ключа* (*primary key*). Первичным ключом таблицы называется минимальный набор столбцов, совокупность значений которых однозначно определяет строку. Это означает, что в таблице не должно быть строк, у которых значения во всех столбцах первичного ключа совпадают, при этом ни один столбец нельзя исключить из первичного ключа, иначе это условие будет нарушено.

На практике первичным ключом служит специальный столбец, значения которого автоматически задает СУБД. Например, в таблице `Customers` (см. табл. 3.1) первичным ключом является столбец `id`. Использовать такой искусственный первичный ключ значительно проще, чем естественный (основанный на атрибутах объекта). Например, в таблице `Customers` столбец `name` не может быть первичным ключом, так как имена клиентов могут совпадать; а первичный ключ из столбцов

name и phone был бы слишком громоздким. Дополнительными преимуществами искусственного ключа являются:

- ❑ гарантированная уникальность значений — ее обеспечивает СУБД;
- ❑ постоянство значений — значение практически любого атрибута может измениться, но значение искусственного ключа при этом менять не придется;
- ❑ числовой тип данных — поиск по числовым значениям выполняется намного быстрее, чем по символьным.

Еще одно применение первичного ключа — организация *связей* между таблицами.

Связи между таблицами. Внешний ключ

Реляционная база данных — это не просто набор таблиц. Объединить разрозненные фрагменты информации в единую, целостную структуру данных позволяют *связи* между таблицами, посредством которых строка одной таблицы сопоставляется со строкой (строками) другой таблицы. Благодаря связям можно извлекать информацию одновременно из нескольких таблиц (например, выводить с помощью одного запроса и сведения о клиенте, и сведения о заказах этого клиента), избежать дублирования информации (например, не требуется в каждом заказе хранить адрес клиента), поддерживать полноту информации (например, не допустить сохранения данных о заказе, если в базе данных отсутствует описание заказанного товара) и многое другое.

Поясню, что такое связь между таблицами, на примере. Допустим, у нас имеется таблица А и таблица В и мы хотим связать одну с другой. Для этого в каждую строку таблицы А мы должны поместить некую информацию, позволяющую идентифицировать связанную с ней строку таблицы В. Эта информация называется *ссылкой*, а поля таблицы А, содержащие эту ссылку, называются *внешними ключами*. Наверное, вы уже сами догадались, что в качестве ссылки используется первичный ключ таблицы В, поскольку именно его значения позволяют однозначно идентифицировать нужную строку таблицы В. После того как мы во все строки таблицы А поместим ссылки на строки таблицы В, таблица А будет связана с таблицей В. При этом таблица А будет называться *дочерней*, а таблица В — *родительской*.

Связи, устанавливаемые между таблицами в базе данных, могут быть трех типов.

- ❑ Связь «один ко многим».

Этот тип связи используется чаще всего. В этом случае каждая строка таблицы А ссылается на одну из строк таблицы В, причем на одну строку таблицы В могут ссылаться несколько строк таблицы А.

Для установки связи между таблицами в дочернюю таблицу добавляется *внешний ключ* (*foreign key*) — столбец или набор столбцов, содержащий значения первичного ключа родительской таблицы (иными словами, во внешнем ключе хранятся ссылки на строки родительской таблицы).

Рассмотрим пример таблицы (табл. 3.2), которая содержит сведения о заказах, сделанных клиентами, и является дочерней по отношению к таблице Customers.

Таблица 3.2. Таблица Orders (Заказы)

id (идентификатор)	date (дата)	product_id (товар)	qty (количество)	amount (сумма)	customer_id (клиент)
1012	12.12.2007	5	8	4500	533
1013	12.12.2007	2	14	22000	536
1014	21.01.2008	5	12	5750	533

В таблице Orders внешним ключом является столбец `customer_id`, в котором содержатся номера клиентов из таблицы Customers. Таким образом, каждая строка таблицы Orders ссылается на одну из строк таблицы Customers. Например, строка с идентификационным номером 1012 содержит в столбце `customer_id` значение 533: это говорит о том, что заказ № 1012 сделан клиентом ООО «Кускус».

Столбец `product_id` таблицы Orders также является внешним ключом — он содержит номера товаров из столбца `id` таблицы Products (Товары). Таким образом, таблица Orders одновременно является дочерней по отношению к таблицам Customers и Products.

❑ Связь «один к одному».

Такая связь между таблицами означает, что каждой строке одной таблицы соответствует одна строка другой таблицы и наоборот. Например, если требуется хранить паспортные данные клиентов, то можно создать таблицу Passports (Паспорта), связанную отношением «один к одному» с таблицей Customers.

Таблицы, соединенные связью «один к одному», можно объединить в одну. Две таблицы вместо одной используют по соображениям конфиденциальности (например, можно ограничить доступ пользователей к таблице Passports), для удобства (если в единой таблице слишком много столбцов), для экономии дискового пространства (в дополнительную таблицу выносят те столбцы, которые часто бывают пустыми, тогда дополнительная таблица содержит значительно меньше строк, чем основная, и обе занимают меньше места, чем единая таблица).

Связь «один к одному» может быть организована так же, как связь «один ко многим», — с помощью первичного ключа родительской таблицы и внешнего ключа дочерней таблицы. Другой вариант — связь посредством первичных ключей обеих таблиц, при этом связанные строки имеют одинаковое значение первичного ключа.

❑ Связь «многие ко многим».

Этот тип связи в реляционной базе данных может быть реализован только с помощью вспомогательной таблицы. Например, если потребуется включать в заказ несколько наименований товаров, то связь «многие ко многим» между таблицами `Orders` и `Products` можно организовать с помощью вспомогательной таблицы `Items` (Позиции заказа), содержащей столбцы `product_id` (номер товара из таблицы `Products`), `qty` (количество товаров данного наименования в заказе) и `order_id` (номер заказа из таблицы `Orders`). При этом столбцы `product_id` и `qty` из таблицы `Orders` исключаются. Таким образом, таблица `Items` будет дочерней по отношению к таблицам `Orders` и `Products` и каждая строка таблицы `Items` будет соответствовать одному наименованию товара в заказе.

Как видите, реляционная база данных представляет собой весьма запутанную структуру, в которой все части (то есть записи) ссылаются на другие самым произвольным образом. А поскольку структура сложная, то неизбежны ее нарушения, происходящие по различным причинам, включая сбои программы, ошибки оператора и пр. Последствия такого нарушения могут быть просто катастрофическими: скажем, что будет, если таблица станет неверно ссылаться на таблицу товаров? Вся деятельность фирмы будет дезорганизована — вместо заказанного товара, допустим лопат, заказчику доставят топоры, а то и вовсе ничего — если ссылка на заказанный товар указывает на несуществующую строку таблицы товаров.

Таким образом, важнейшим понятием теории реляционных баз данных является *целостность данных*. Рассмотрим его подробнее.

Целостность данных

Целостностью данных называется корректность и непротиворечивость данных, хранимых в СУБД.

Базовыми требованиями целостности, которые должны выполняться в любой реляционной базе данных, являются *целостность сущностей* и *целостность связей* (ссылочная целостность).

Целостность сущностей означает, что в каждой таблице есть первичный ключ — уникальный идентификатор строки. Первичный ключ не должен содержать противоречащих и неопределенных значений. Например, если в таблицу `Customers`

добавить еще одну строку с идентификатором 533 (притом что одна строка с таким идентификатором уже существует в таблице), то целостность сущностей будет нарушена и невозможно будет определить, кому из этих двух клиентов с одинаковыми идентификаторами принадлежат заказы № 1012 и 1014.

Целостность связей означает, что внешний ключ в дочерней таблице не содержит значений, отсутствующих в первичном ключе родительской таблицы. Иными словами, строка дочерней таблицы не должна ссылаться на несуществующую строку родительской таблицы. В отличие от первичного ключа внешний ключ может содержать неопределенные значения (NULL), это не считается нарушением целостности. Например, если в таблицу `Orders` добавить строку, содержащую в столбце `customer_id` значение 999, то будет нарушена целостность связи между таблицами `Customers` и `Orders`: с одной стороны, заказ кем-то сделан, так как в ином случае в столбце `customer_id` было бы установлено значение NULL, с другой стороны, невозможно выяснить имя и адрес клиента, сделавшего этот заказ.

Как видно из приведенных примеров, если целостность данных нарушена, то с ними невозможно нормально работать. По этой причине поддержание целостности данных — одна из основных функций любой СУБД.

Для поддержания целостности сущностей СУБД проверяет корректность значения первичного ключа при добавлении и изменении строк. Механизм поддержания ссылочной целостности более сложный. Помимо проверки корректности значения внешнего ключа при добавлении и изменении строк дочерней таблицы, необходимо предотвратить нарушение ссылочной целостности при удалении и изменении строк родительской таблицы. Для этого используется один из следующих способов.

- ❑ **Запрет (RESTRICT)** — если на строку родительской таблицы ссылается хотя бы одна строка дочерней таблицы, то удаление родительской строки и изменение значения первичного ключа в такой строке запрещаются. Например, не допускается удаление информации о клиенте из таблицы `Customers`, если у этого клиента есть зарегистрированные заказы, то есть строки в таблице `Orders`, которые ссылаются на строку со сведениями об этом клиенте.
- ❑ **Каскадное удаление/обновление (CASCADE)** — при удалении строки из родительской таблицы автоматически удаляются все ссылающиеся на нее строки дочерней таблицы; при изменении значения первичного ключа в строке родительской таблицы автоматически обновляется значение внешнего ключа в ссылающихся на нее строках дочерней таблицы.

Например, при удалении записи о клиенте из таблицы `Customers` автоматически удаляются сведения о заказах этого клиента, то есть соответствующие строки в таблице `Orders`.

- ❑ Обнуление (`SET NULL`) — при удалении строки и изменении значения первичного ключа в строке значение внешнего ключа во всех строках, ссылающихся на данную, автоматически становится неопределенным (`NULL`).

Например, при удалении записи о клиенте из таблицы `Customers` заказы этого клиента автоматически аннулируются, то есть в соответствующих строках таблицы `Orders` в столбце `customer_id` устанавливается значение `NULL`.

В СУБД MySQL способ поддержания целостности связи указывается при создании или изменении структуры дочерней таблицы.

С понятием целостности данных тесно связано понятие *транзакции*. Транзакцией называется группа связанных операций, которые должны быть либо все выполнены, либо все отменены. Если при выполнении одной из операций происходит ошибка или сбой, то транзакция отменяется, все уже внесенные другими операциями изменения автоматически аннулируются и восстанавливается исходное состояние база данных. Важнейшее применение транзакций — это объединение тех операций, которые, будучи выполненными по отдельности, могут нарушить целостность данных. Например, рассмотренная выше операция каскадного удаления проводится как единая транзакция: строка родительской таблицы должна быть удалена вместе со всеми ссылающимися на нее строками дочерней таблицы, а если по каким-либо причинам одну из этих строк удалить невозможно, то не будет удалена ни одна из строк.

Теперь, когда мы рассмотрели основные понятия теории реляционных баз данных, можно приступить к разработке собственной базы.

3.3. Проектирование базы данных

Построение базы данных (как и любой информационной системы, любого программного продукта) начинается с проектирования. В процессе проектирования мы определяем задачи, для решения которых предназначена база данных, и создаем представление данных и связей между ними, необходимое для решения этих задач.

Проектирование включает в себя следующие основные этапы.

1. **Определение требований к базе данных.** В первую очередь необходимо составить перечень требований, которым должна соответствовать проектируемая база данных. В этом разделе мы рассматриваем только функциональные требования: хотя другие требования (производительность, масштабируемость, надежность) также нужно учитывать, их выполнение во многом зависит от используемой СУБД.

Например, при проектировании базы данных для торговой компании может выясниться, что отделу по работе с клиентами необходимо знать номера телефонов всех клиентов, отделу доставки нужен отчет, содержащий адрес клиента и список заказанных им товаров, отделу логистики требуется информация о том, какие товары в каком количестве были заказаны в прошлом месяце, и т. п. Эти запросы и будут положены в основу проекта базы данных.

2. **Создание модели данных, соответствующей всем предъявленным требованиям.** Для разработки модели данных на основе сформулированных требований можно использовать одну из двух противоположных стратегий.

- Проектирование «снизу вверх», от элемента к структуре: сначала определяется, какие именно атрибуты должны храниться в базе данных, затем группы атрибутов объединяются в объекты. Этот метод годится для небольших баз данных, в которых количество атрибутов невелико.
- Проектирование «сверху вниз» начинается с выделения высокоуровневых объектов и связей между ними, затем осуществляется декомпозиция объектов и последовательная детализация модели до уровня атрибутов. Для сложных баз данных с большим количеством атрибутов такой метод значительно эффективнее, чем метод «снизу вверх».

В результате мы получим предварительную структуру базы данных: список объектов (таблиц) и список атрибутов каждого объекта (столбцов таблицы). Например, на основе требований, приведенных в п. 1, можно построить модель данных, содержащую сведения о таких объектах, как клиенты, заказы и товары:

- для клиентов предполагается хранить следующие атрибуты: идентификатор, имя (или название организации), номер контактного телефона, адрес, а также рейтинг, используемый для расчета скидки;
- для товаров: идентификатор, наименование, описание, название склада, где хранится этот вид товара, адрес склада;
- для заказов: дату заказа, идентификатор заказанного товара, количество товаров этого наименования, общую стоимость заказа с учетом скидки, идентификатор клиента, сделавшего заказ, и адрес клиента, куда нужно доставить заказ (здесь мы предполагаем, что каждый заказ может включать только одно наименование товара, для каждого заказанного наименования регистрируется новый заказ).

3. **Нормализация базы данных.** Заключается в минимизации избыточности данных. Нормализация позволяет уменьшить объем базы данных и устранить

потенциальную противоречивость данных (например, если в базе данных одна и та же информация дублируется в нескольких местах, то при ее обновлении есть риск появления разночтений).

Результат нормализации — приведение таблиц базы данных к одной из *нормальных форм*. На практике чаще всего используются три перечисленные ниже нормальные формы.

- Таблица находится в *первой нормальной форме*, если все атрибуты атомарны, то есть на пересечении любого столбца и строки находится значение, части которого не будут использоваться по отдельности.

Ответ на вопрос, является ли атрибут атомарным, зависит от функциональных требований к базе данных. Рассмотрим, например, столбец `address` из таблицы `Customers` (см. табл. 3.1). Если адрес клиента будет использоваться только целиком, то этот столбец является атомарным. Если же потребуется получать из базы отдельно название города, улицы и т. п., то для приведения таблицы `Customers` к первой нормальной форме столбец `address` следует разбить на столбцы `city` (город), `street` (улица), `building` (здание) и т. д.

- Таблица находится во *второй нормальной форме*, если она находится в первой нормальной форме и ни один из ее неключевых атрибутов не находится в функциональной зависимости от *части* первичного ключа.

Получается, что в таблице, в которой есть составной первичный ключ, значения остальных столбцов таблицы должны зависеть от значений *всех* столбцов первичного ключа. Если же есть столбцы, которые зависят только от *некоторых* столбцов первичного ключа, то для приведения таблицы во вторую нормальную форму необходимо перенести все эти столбцы в другую таблицу.

Например, в нашей модели, построенной в п. 1, в таблице заказов первичным ключом может служить набор столбцов, содержащих дату заказа, идентификатор товара и идентификатор клиента (если мы допустим, что клиент не может сделать повторный заказ того же товара в тот же день, а может только изменить ранее сделанный заказ). Таким образом, для приведения таблицы заказов ко второй нормальной форме нужно исключить из таблицы адрес клиента, так как он зависит от идентификатора клиента, который является частью возможного первичного ключа. В противном случае адрес клиента будет повторяться в каждом заказе, что может привести к несогласованности данных. В частности, при изменении адреса клиента потребуется изменить адрес во всех заказах этого клиента. Если

при выполнении такого массового обновления данных произойдет ошибка, то возможна ситуация, когда в некоторых заказах адрес будет изменен, а в некоторых останется прежним и будет неясно, какой из адресов правильный. Нормализация таблицы позволяет избежать такой несогласованности.



ПРИМЕЧАНИЕ

Атрибут А функционально зависит от группы атрибутов В, если значение атрибута А однозначно определяется набором значений группы атрибутов В, иными словами, в строках с одинаковым набором значений атрибутов группы В значение атрибута А также одинаково.

- Таблица находится в *третьей нормальной форме*, если она находится во второй нормальной форме и любой неключевой атрибут функционально зависит *только* от первичного ключа.

Например, в модели данных из п. 1 таблица, содержащая сведения о товарах, не находится в третьей нормальной форме, поскольку в ней имеется функциональная зависимость адреса склада от его названия. Таким образом, вам придется всякий раз при упоминании склада писать и его адрес, что приведет к многократному дублированию данных. Чтобы привести таблицу к третьей нормальной форме, нужно все данные о складе вынести в отдельную таблицу, которая будет родительской по отношению к таблице товаров.

Когда все таблицы базы данных приведены в третью нормальную форму, мы можем считать, что наша база данных нормализована и информация о каждом факте хранится только в одном месте.

Итак, мы разработали логическую структуру базы данных и можно переходить к созданию базы данных в СУБД MySQL. Будем считать, что на вашем компьютере установлена среда XAMPP с поддержкой сервера MySQL. Если вы этого еще не сделали, придется вернуться к установке и настройке, чтобы иметь возможность повторять действия с базой данных, описываемые далее.

3.4. Управление базой данных с помощью SQL

Из этого раздела вы узнаете, как работать с данными в СУБД MySQL: определять структуру данных, добавлять, изменять, удалять и искать данные. Для выполнения

всех этих операций используется SQL — универсальный язык структурированных запросов, являющийся стандартным средством доступа к реляционным базам данных.

Для выполнения SQL-команд вы можете использовать любое из многочисленных клиентских приложений сервера MySQL. Мы будем использовать графическую утилиту phpMyAdmin, входящую в пакет XAMPP. Преимущество такого подхода еще и в том, что эта утилита — очень популярное средство, предоставляемое многими хостинг-провайдерами для управления базами данных.

Утилита phpMyAdmin предоставляет множество вспомогательных средств для более удобной работы с базой данных, в том числе наглядное представление компонентов базы, непосредственное редактирование данных (без использования SQL-оператора UPDATE), дополнительные возможности работы с запросами, такие как построение запросов с помощью специального инструмента (при этом не нужно вручную вводить названия таблиц и столбцов), сохранение запросов в файле, экспорт результатов запросов и многое другое.

Сначала поясню, как выполнять SQL-команды в командной строке и в phpMyAdmin, а в дальнейшем будем рассматривать только синтаксис SQL-команд.

Выполнение SQL-команд

Прежде чем выполнять SQL-команды, необходимо подключиться к работающему серверу MySQL (как это сделать, рассказывалось в главе 1). Из этого раздела вы узнаете, как создавать SQL-команды и передавать их серверу для выполнения.

Мы будем использовать для этой цели панель управления phpMyAdmin. Откроем ее на панели управления XAMPP нажатием кнопки Admin (Администратор) и перейдем на вкладку SQL, предназначенную для передачи SQL-запросов серверу (рис. 3.1).

В поле Выполнить SQL-запрос(ы) на сервере "localhost" следует ввести команду на языке SQL и нажать кнопку OK для передачи ее на сервер MySQL. Если команд несколько, то каждая должна завершаться символом, указанным в поле Разделитель (по умолчанию используется точка с запятой).

Например, введем команду `SHOW DATABASES`, которая указывает серверу, что следует отобразить все базы данных, подключенные к серверу. Сообщение о результате выполнения команды, а также запрошенные данные выводятся в этом же окне (рис. 3.2).

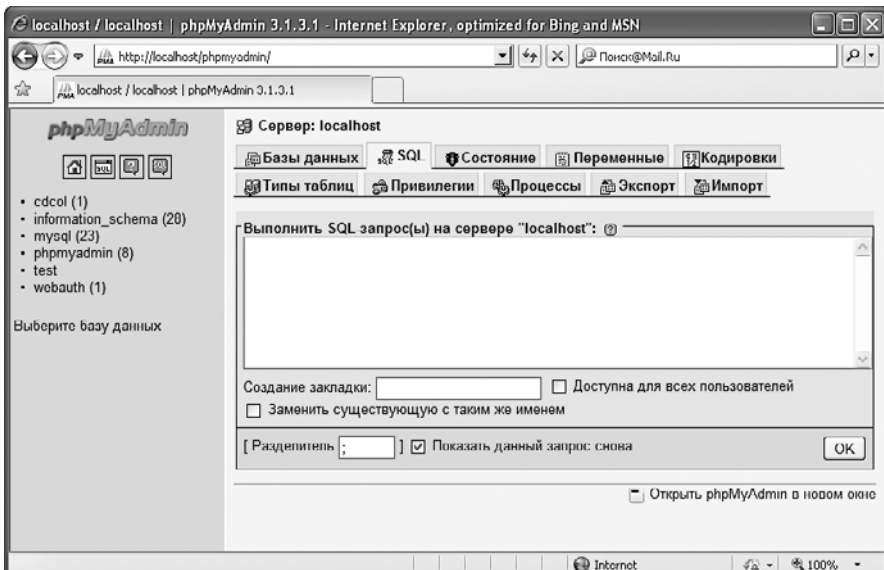


Рис. 3.1. Вкладка для работы с SQL-запросами

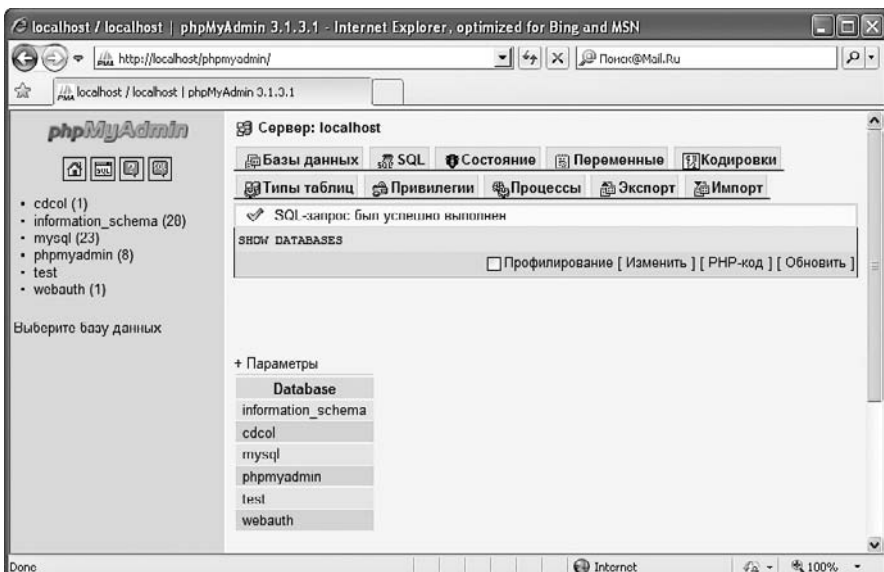


Рис. 3.2. Выполнение SQL-запроса в окне phpMyAdmin

Видим, что в ответ отобразилось несколько баз данных, создаваемых при установке пакета XAMPP.

Теперь, когда вы научились вводить SQL-команды, приступим к управлению данными с помощью этих команд. В первую очередь мы рассмотрим команды, предназначенные для работы с базой данных в целом.

Создание базы данных

Из этого раздела вы узнаете, как создать базу данных, удалить ее, изменить кодировку по умолчанию, выбрать текущую базу, а также просмотреть список всех баз данных на сервере MySQL.

Чтобы создать базу данных, выполним команду:

```
CREATE DATABASE <Имя базы данных>;
```

Например, команда:

```
CREATE DATABASE SalesDept;
```

создает базу данных с именем SalesDept (Отдел продаж).

Если вам по каким-либо причинам нужно установить для новой базы данных кодировку по умолчанию, отличную от кодировки, указанной при настройке MySQL, то при создании базы вы можете указать нужную кодировку (CHARACTER SET) и/или правило сравнения (сортировки) символьных значений:

```
CREATE DATABASE <Имя базы данных>  
CHARACTER SET <Имя кодировки>  
COLLATE <Имя правила сравнения>;
```

Например, если вы будете импортировать в новую базу данные, которые находятся в кодировке CP-1251, то укажите эту кодировку при создании базы таким образом:

```
CREATE DATABASE SalesDept  
CHARACTER SET cp1251 COLLATE cp1251_general_ci;
```

СОВЕТ



Чтобы просмотреть список используемых в MySQL кодировок, выполните команду SHOW CHARACTER SET;, а чтобы увидеть список правил сравнения символьных значений — команду SHOW COLLATION;. При этом можно использовать оператор LIKE. Например, чтобы увидеть все правила сравнения для кодировки CP-1251, выполним команду SHOW COLLATION LIKE '%1251%'. Окончание _ci (case insensitive) в названии правил сравнения означает, что при сравнении

и сортировке регистр символов не учитывается, окончание `_cs` (case sensitive) — что регистр учитывается, окончание `_bin` (binary) — сравнение и сортировка выполняются по числовым кодам символов. Для большинства правил сравнения вы можете найти описание (то есть порядок следования символов, в соответствии с которым будут упорядочиваться текстовые значения) на веб-странице <http://www.collation-charts.org/mysql60/>.

Кодировка, указанная при создании базы данных, будет по умолчанию использоваться для таблиц этой базы данных, однако вы можете при создании таблицы задать и другую кодировку.

Изменить кодировку и/или правило сравнения символьных значений для базы данных можно с помощью команды:

```
ALTER DATABASE <Имя базы данных>  
  CHARACTER SET <Имя кодировки>  
  COLLATE <Имя правила сравнения>;
```

При этом кодировка, используемая в уже существующих таблицах базы данных, остается прежней, меняется только кодировка, назначаемая по умолчанию для вновь создаваемых таблиц.

Чтобы удалить ненужную или ошибочно созданную базу данных, выполним команду:

```
DROP DATABASE <Имя базы данных>;
```

ВНИМАНИЕ



Удаление базы данных — очень ответственная операция, поскольку она приводит к удалению всех таблиц этой базы и всех данных, хранившихся в таблицах. Рекомендуется перед удалением создать резервную копию базы данных.

Одну из баз данных, созданных на сервере MySQL, вы можете выбрать в качестве текущей с помощью команды:

```
USE <Имя базы данных>;
```

Например:

```
USE SalesDept;
```

После этого вы можете выполнять операции с таблицами этой базы данных, не добавляя имя базы в виде префикса к имени таблицы. Например, для обращения к таблице `Customers` базы данных `SalesDept` можно вместо `SalesDept.Customers` писать просто `Customers`. Указав текущую базу, вы тем не менее можете

обращаться и к таблицам других баз данных, но использование имени базы данных в виде префикса при этом обязательно. Выбор текущей базы данных сохраняется до момента отсоединения от сервера или до выбора другой текущей базы данных.

Чтобы увидеть список всех баз данных, существующих на данном сервере MySQL, выполним команду:

```
SHOW DATABASES;
```

Даже если вы еще не создали ни одной базы данных, в полученном списке вы увидите три системные базы:

- ❑ `INFORMATION_SCHEMA` — информационная база данных, из которой вы можете получить сведения обо всех остальных базах данных, структуре данных в них и о всевозможных объектах: таблицах, столбцах, первичных и внешних ключах, правах доступа, хранимых процедурах, кодировках и др. Эта база данных доступна только для чтения и является виртуальной, то есть не хранится в виде каталога на диске. В действительности вся информация, запрашиваемая из этой базы данных, предоставляется динамически сервером MySQL;
- ❑ `mysql` — служебная база данных, которую использует сервер MySQL. В ней хранятся сведения о зарегистрированных пользователях и их правах доступа, справочная информация и др.;
- ❑ `test` — пустая база данных, которую можно использовать для «пробы пера» или просто удалить.

Итак, мы освоили основные операции, выполняемые с базой данных как единым целым, а именно команды `CREATE DATABASE` (создание), `ALTER DATABASE` (изменение), `DROP DATABASE` (удаление), `USE` (выбор текущей базы данных) и `SHOW DATABASES` (просмотр списка баз данных). Далее мы рассмотрим операции с таблицами. При этом будем считать, что вы выбрали какую-либо базу данных в качестве текущей и работаете с ее таблицами.

Работа с таблицами

Из этого раздела вы узнаете, как создать, изменить и удалить таблицу, как просмотреть информацию о таблице и список всех таблиц в текущей базе данных. Начнем с наиболее сложной команды — команды создания таблицы.

Типы таблиц

Тип таблицы — наиболее важное из опциональных свойств таблицы. В MySQL существует множество типов таблиц, каждый из которых лучше всего подходит для решения определенной задачи. Основными типами таблиц являются InnoDB и MyISAM.

Таблицы InnoDB обеспечивают поддержку транзакций (о транзакциях мы говорили в начале главы, когда обсуждали понятие целостности данных) и блокировок отдельных строк, благодаря которым обеспечивается высокая производительность операций изменения данных в многопользовательском режиме. Кроме того, как вы видели в предыдущем подразделе, в таблицах InnoDB можно настроить внешние ключи для поддержания целостности связей между таблицами.

Таблицы MyISAM не поддерживают объединение нескольких операций в единую транзакцию, поэтому, в частности, невозможно автоматическое поддержание целостности связей между такими таблицами. Однако таблицы MyISAM также часто используются: их преимуществами являются высокая скорость выполнения поисковых запросов и меньшая нагрузка на системные ресурсы.

При установке пакета XAMPP в качестве типа таблицы по умолчанию автоматически выбирается MyISAM. В этом случае, если требуется создать таблицу с типом InnoDB, в команду создания таблицы следует добавить выражение `ENGINE InnoDB` (именно так мы и поступим в следующем разделе).

Все таблицы, используемые в нашем примере, созданы с параметром `ENGINE InnoDB`. Это дает нам возможность продемонстрировать настройку внешних ключей в таблицах примера.

Вместо ключевого слова `ENGINE` можно использовать ключевое слово `TYPE` — они являются синонимами.

Создание таблицы

Чтобы создать таблицу, выполним команду, представленную в листинге 3.1.

Листинг 3.1. Команда создания таблицы

```
CREATE TABLE <Имя таблицы>
(<Имя столбца 1> <Тип столбца 1> [<Свойства столбца 1>],
 <Имя столбца 2> <Тип столбца 2> [<Свойства столбца 2>],
```

```
...  
[<Информация о ключевых столбцах и индексах>]]  
[<Опциональные свойства таблицы>];
```

Как видите, команда создания таблицы может включать множество параметров. Однако многие из них задавать не обязательно (в листинге 3.1 такие параметры заключены в квадратные скобки). В действительности для создания таблицы достаточно указать ее имя, а также имена и типы всех столбцов; остальные параметры используются при необходимости.

Рассмотрим сначала несколько примеров, которые помогут вам освоить команду `CREATE TABLE` и сразу же, не изучая ее многочисленных параметров, начать создавать собственные (простые по структуре) таблицы.

Предположим, что мы строим базу данных, которую спроектировали в начале этой главы. Используя команды из предыдущего раздела, мы создали пустую базу данных `SalesDept` и выбрали ее в качестве текущей. Теперь создадим три таблицы: `Customers`, `Products` и `Orders`.

В листинге 3.2 представлена команда создания таблицы `Customers`.

Листинг 3.2. Команда создания таблицы `Customers`

```
CREATE TABLE Customers  
(id SERIAL,  
 name VARCHAR(100),  
 phone VARCHAR(20),  
 address VARCHAR(150),  
 rating INT,  
 PRIMARY KEY (id))  
ENGINE InnoDB CHARACTER SET utf8;
```

Поясню, какие параметры мы использовали в этой команде. Во-первых, мы дали название таблице. Во-вторых, мы перечислили названия и типы столбцов, из которых будет состоять таблица.

- ❑ `id` — идентификатор записи. Этому столбцу мы назначили тип `SERIAL`, позволяющий автоматически нумеровать строки таблицы. Ключевое слово `SERIAL` расшифровывается как `BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE`. Это означает, что в столбец можно вводить большие целые (`BIGINT`) положительные (`UNSIGNED`) числа, при этом автоматически контролируется отсутствие

неопределенных и повторяющихся значений (`NOT NULL UNIQUE`), а если при добавлении строки в таблицу вы не укажете значение для этого столбца, то программа MySQL внесет в этот столбец очередной порядковый номер (`AUTO_INCREMENT`).



ПРИМЕЧАНИЕ

NULL — это константа, указывающая на отсутствие значения. Если в столбце находится значение NULL, то считается, что никакого определенного значения для этого столбца не задано (поэтому мы также называем NULL неопределенным значением). Не следует путать NULL с пустой строкой (""), или числом 0. Значения NULL обрабатываются особым образом: большинство функций и операторов возвращают NULL, если один из аргументов равен NULL. Например, результат сравнения `1 = 1` — истинное значение (`true`), а результат сравнения `NULL = NULL` — неопределенное значение (NULL), то есть два неопределенных значения не считаются равными.

- ❑ `name` — имя клиента, `phone` — номер телефона, `address` — адрес. Мы присвоили этим столбцам тип `VARCHAR`, поскольку они будут содержать символьные значения. В скобках указывается максимальное допустимое количество символов в значении столбца.
- ❑ `rating` — рейтинг. Тип `INT` означает, что столбец будет содержать обычные целые числа.

В-третьих, мы указали, что столбец `id` будет первичным ключом таблицы, включив в команду создания таблицы определение `PRIMARY KEY (id)`.

В-четвертых, мы задали для этой таблицы два опциональных параметра. Параметр `ENGINE` определяет тип таблицы. Таблице `Customers` мы присвоили тип `InnoDB`, так как только он обеспечивает поддержание целостности связей между таблицами. Параметр `CHARACTER SET` определяет кодировку по умолчанию для данных в таблице. Поскольку мы не задали кодировку отдельно для столбцов `name`, `phone` и `address`, данные в этих столбцах будут храниться в кодировке `UTF-8`, которая назначена в качестве кодировки по умолчанию для таблицы `Customers`.

Следующий пример, который мы рассмотрим, — команда создания таблицы `Products`, представленная в листинге 3.3.

Листинг 3.3. Команда создания таблицы `Products`

```
CREATE TABLE Products
(id SERIAL,
description VARCHAR(100).
```

```
details TEXT,  
price DECIMAL(8,2),  
PRIMARY KEY (id))  
ENGINE InnoDB CHARACTER SET utf8;
```

Эта команда очень похожа на команду создания таблицы `Customers` и отличается от нее только названием таблицы и набором столбцов. Столбцы `id` и `description` (наименование товара) таблицы `Products` имеют уже знакомые нам типы. Столбец `details` (описание) имеет тип `TEXT`. Этот тип удобно использовать вместо типа `VARCHAR`, если столбец будет содержать длинные значения: суммарная длина значений всех столбцов с типом `VARCHAR` ограничена 65 535 байтами для каждой таблицы, а на общую длину столбцов с типом `TEXT` ограничений нет. Недостаток типа `TEXT` — невозможность включать такие столбцы во внешний ключ таблицы, то есть создавать связь между таблицами на основе этих столбцов.

Столбец `price` (цена) имеет тип `DECIMAL`, предназначенный для хранения денежных сумм и других значений, для которых важно избежать ошибок округления. В скобках мы указали два числа: первое из них определяет максимальное количество цифр в значении столбца, второе — максимальное количество цифр после десятичного разделителя. А именно, цена товара может содержать до шести цифр в целой части ($6 = 8 - 2$) и до двух цифр в дробной части.

И наконец, последний пример — команда создания таблицы `Orders`, представленная в листинге 3.4.

Листинг 3.4. Команда создания таблицы `Orders`

```
CREATE TABLE Orders  
(id SERIAL,  
date DATE,  
product_id BIGINT UNSIGNED NOT NULL,  
qty INT UNSIGNED,  
amount DECIMAL(10,2),  
customer_id BIGINT UNSIGNED,  
PRIMARY KEY (id),  
FOREIGN KEY (product_id) REFERENCES Products (id)  
ON DELETE RESTRICT ON UPDATE CASCADE,  
FOREIGN KEY (customer_id) REFERENCES Customers (id)
```

```
ON DELETE RESTRICT ON UPDATE CASCADE)  
ENGINE InnoDB CHARACTER SET utf8;
```

Особенность таблицы `Orders` — наличие внешних ключей: столбец `product_id` (товар) содержит номера товаров из таблицы `Products`, а столбец `customer_id` (клиент) — номера клиентов из таблицы `Customers`. Поскольку номера товаров и номера клиентов являются большими целыми положительными числами, столбцам `product_id` и `customer_id` мы назначили тип `BIGINT UNSIGNED`.

Далее, чтобы обеспечить автоматическое поддержание целостности связей (о целостности рассказывалось ранее в этой главе), мы сообщили программе MySQL, какому первичному ключу соответствует каждый внешний ключ. Так, конструкция `FOREIGN KEY (customer_id) REFERENCES Customers (id)` означает, что в столбце `customer_id` могут содержаться только значения из столбца `id` таблицы `Customers` и неопределенные значения (`NULL`), остальные значения запрещены. Аналогичное ограничение мы задали и для столбца `product_id`, а кроме того, присвоили этому столбцу свойство `NOT NULL`, чтобы запретить регистрировать заказы с неопределенным товаром. Дополнительно мы указали для каждой из связей правила поддержания целостности (их мы также обсуждали в начале этой главы). Правило `ON DELETE RESTRICT` означает, что нельзя удалить запись о клиенте, если у этого клиента есть зарегистрированный заказ, и нельзя удалить запись о товаре, если этот товар был кем-то заказан. Правило `ON UPDATE CASCADE` означает, что при изменении номера клиента в таблице `Customers` или номера товара в таблице `Products` соответствующие изменения вносятся и в таблицу `Orders`.



ПРИМЕЧАНИЕ

Обратите внимание, что таблицу `Orders` мы создали в последнюю очередь, так как первичные ключи в таблицах `Customers` и `Products` должны быть созданы раньше, чем ссылающиеся на них внешние ключи в таблице `Orders`. Впрочем, можно было бы создать таблицы без внешних ключей в любой последовательности, а затем добавить внешние ключи командой `ALTER TABLE`, которую мы рассмотрим в подразделе ниже.

В этих примерах показаны лишь некоторые параметры команды создания таблицы. Теперь перечислим все основные параметры, которые могут вам пригодиться при создании таблиц.

Типы данных в MySQL

Как уже говорилось, при создании таблицы нужно указать тип данных для каждого столбца. В MySQL предусмотрено множество типов данных для хранения чисел,

даты/времени и символьных строк (текстов). Кроме того, существуют типы данных для хранения пространственных объектов, которые в этой книге мы рассматривать не будем.

Начнем с изучения числовых типов данных.

Числовые типы данных

К числовым типам данных относятся следующие.

- ❑ `BIT [(<Количество битов>)]` — битовое число, содержащее заданное количество битов. Если количество битов не указано, число состоит из одного бита.
- ❑ `TINYINT` — целое число в диапазоне либо от -128 до 127 , либо (если указано свойство `UNSIGNED`) от 0 до 255 .
- ❑ `BOOL` или `BOOLEAN` — синонимы к типу данных `TINYINT (1)` (число в скобках — это количество отображаемых цифр). При этом ненулевое значение рассматривается как истинное (`true`), нулевое — как ложное (`false`).
- ❑ `SMALLINT` — целое число в диапазоне либо от $-32\,768$ до $32\,767$, либо (если указано свойство `UNSIGNED`) от 0 до $65\,535$.
- ❑ `MEDIUMINT` — целое число в диапазоне либо от $-8\,388\,608$ до $8\,388\,607$, либо (если указано свойство `UNSIGNED`) от 0 до $16\,777\,215$.
- ❑ `INT` или `INTEGER` — целое число в диапазоне либо от $-2\,147\,483\,648$ до $2\,147\,483\,647$, либо (если указано свойство `UNSIGNED`) от 0 до $4\,294\,967\,295$.
- ❑ `BIGINT` — целое число в диапазоне либо от $-9\,223\,372\,036\,854\,775\,808$ до $9\,223\,372\,036\,854\,775\,807$, либо (если указано свойство `UNSIGNED`) от 0 до $18\,446\,744\,073\,709\,551\,615$.
- ❑ `SERIAL` — синоним выражения `BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE` (большое целое число без знака, принимающее автоматически увеличиваемые уникальные значения; значения `NULL` запрещены). Используется для автоматической генерации уникальных значений в столбце первичного ключа. Свойства `UNSIGNED`, `AUTO_INCREMENT`, `NOT NULL` и `UNIQUE` описаны чуть ниже.

ПРИМЕЧАНИЕ



Для всех целочисленных типов данных, кроме `BOOL (BOOLEAN)` и `SERIAL`, можно в скобках указать количество отображаемых цифр, которое используется

совместно с параметром ZEROFILL: если число содержит меньшее количество цифр, то при выводе оно дополняется слева нулями. Например, если столбец таблицы определен как INT(5) ZEROFILL, то значения 1234567 и 12345 отображаются «как есть», а значение 123 отображается как 00123. Для типа данных BIT в скобках указывается размер числа, то есть максимальное количество хранимых битов.

- ❑ FLOAT — число с плавающей точкой в диапазоне от $-3,402823466^{38}$ до $-1,175494351^{-38}$ и от $1,175494351^{-38}$ до $3,402823466^{38}$ (а также значение 0) с точностью около семи значащих цифр (точность зависит от возможностей вашего компьютера).
- ❑ DOUBLE, DOUBLE PRECISION или REAL — число с плавающей точкой в диапазоне от $-1,7976931348623157^{308}$ до $-2,2250738585072014^{-308}$ и от $2,2250738585072014^{-308}$ до $1,7976931348623157^{308}$ (а также значение 0) с точностью около 15 значащих цифр (точность зависит от возможностей вашего компьютера).
- ❑ FLOAT(<Точность>) — при значении точности от 0 до 24 этот тип данных эквивалентен типу FLOAT, при значении от 25 до 53 — типу DOUBLE.
- ❑ DECIMAL, DEC, NUMERIC или FIXED — точное (неокругляемое) число с фиксированной точкой. Может содержать до 65 значащих цифр и до 30 цифр после десятичного разделителя (по умолчанию 10 значащих цифр и 0 после десятичного разделителя).



ПРИМЕЧАНИЕ

Для всех десятичных (нецелочисленных) типов данных, кроме FLOAT(точность), можно в скобках указать точность и шкалу, то есть максимальное количество хранимых значащих цифр и максимальное количество хранимых цифр после десятичного разделителя. Например, если для столбца задан тип данных FLOAT(7,5), то в этот столбец нельзя добавить значение более чем с двумя ($2 = 7 - 5$) цифрами в целой части и все введенные значения будут округляться до пяти знаков после десятичного разделителя. Для чисел с плавающей точкой можно указать точность до 255 и шкалу до 30, однако указывать слишком большую точность и шкалу не имеет смысла, так как в базе данных сохраняются приближенные значения, которые совпадают с реальными лишь в первых 7 (для типа FLOAT) или 15 (для типа DOUBLE) значащих цифрах, а последующие цифры при сохранении могут быть искажены. Для чисел с фиксированной точкой можно указать точность до 65 и шкалу до 30. Если точность и шкала не указаны, то они равны 10 и 0 соответственно. При сохранении чисел с фиксированной точкой искажений не происходит.

Завершая рассмотрение числовых типов данных, обсудим три свойства, которые можно указать для числовых столбцов.

- ❑ UNSIGNED — означает, что в столбце запрещены отрицательные (со знаком «-») значения. Указывать это свойство можно для любых столбцов с числовым типом

данных, кроме `BIT`, `BOOL (BOOLEAN)` и `SERIAL`. Для целочисленных столбцов при добавлении свойства `UNSIGNED` максимальное допустимое значение столбца увеличивается вдвое.

- ❑ `ZEROFILL` — говорит о том, что значения при отображении будут дополнены нулями. Целые числа дополняются нулями слева в соответствии с указанным количеством отображаемых цифр, десятичные — слева и справа в соответствии с указанными точностью и шкалой. Например, если столбец определен как `DOUBLE (10, 5) ZEROFILL`, то значение `12.23` отображается как `0012.23000`. Кроме того, данное свойство запрещает отрицательные значения, как и свойство `UNSIGNED`. Указывать свойство `ZEROFILL` можно для любых столбцов с числовым типом данных, кроме `BIT`, `BOOL (BOOLEAN)` и `SERIAL`.
- ❑ `AUTO_INCREMENT` — обеспечивает автоматическую нумерацию строк таблицы. Это означает, что при добавлении в столбец неопределенного (`NULL`) или нулевого значения это значение автоматически заменяется следующим номером, на единицу больше предыдущего (нумерация по умолчанию начинается с единицы, установить другой начальный номер можно с помощью соответствующего свойства таблицы). Указывать это свойство можно для любых столбцов с числовым типом данных, кроме `BIT` и `DECIMAL (DEC, NUMERIC, FIXED)`. В таблице может быть только один столбец с таким свойством, и для него должен быть создан ключ или индекс (см. ниже).

В следующем подразделе мы рассмотрим типы данных, используемые при хранении даты и времени.

Типы данных даты и времени

Для столбца, который будет содержать дату и/или время, вы можете использовать один из следующих типов данных.

- ❑ `DATE` — дата в формате `YYYY-MM-DD` в диапазоне от `0000-01-01` до `9999-12-31`.
- ❑ `DATETIME` — дата и время в формате `YYYY-MM-DD HH:MM:SS` в диапазоне от `0000-01-01 00:00:00` до `9999-12-31 23:59:59`.
- ❑ `TIMESTAMP` — отметка времени в формате `YYYY-MM-DD HH:MM:SS` в диапазоне от `1970-01-01 00:00:00` до некоторой даты в 2038 году. При добавлении или изменении строки таблицы в столбце с типом `TIMESTAMP` автоматически устанавливаются дата и время выполнения операции (если значение этого столбца не указано явно или указано неопределенное значение). Если нужно, чтобы отметка времени проставлялась только при добавлении строки, то

после слова `TIMESTAMP` следует добавить свойство `DEFAULT CURRENT_TIMESTAMP`.

Если в таблице несколько столбцов с типом `TIMESTAMP`, то отметка времени автоматически проставляется только в первом из них. Если необходимо также вносить отметку времени в какой-либо из последующих столбцов с типом `TIMESTAMP`, то при добавлении/изменении строки укажем для этого столбца значение `NULL`, которое будет автоматически заменено текущей датой.

- ❑ `TIME` — время в формате `HH:MM:SS` в диапазоне от `-838:59:59` до `838:59:59`.
- ❑ `YEAR`, `YEAR (2)`, `YEAR (4)` — год в формате `YYYY` или `YY` (если количество цифр не указано, используется формат `YYYY`). Диапазон значений — от 1901 до 2155, если используется формат `YYYY`, или от 70 (соответствует 1970 году) до 69 (соответствует 2069 году), если используется формат `YY`.

Надо отметить, что MySQL воспринимает даты не только в указанном выше формате. Вы можете ввести дату с любым знаком препинания в качестве разделителя, например `2007@12@31 23%59%59`, или даже совсем без разделителя, например `20071231235959`. Более того, если в столбец с типом даты или времени вносится символьное либо числовое значение в одном из таких форматов, MySQL автоматически преобразует это значение в дату и/или время.

Изучение типов данных завершим рассмотрением символьных типов.

Символьные типы данных

Столбцам, которые будут содержать текст, можно присвоить один из следующих типов данных.

- ❑ `CHAR (<Количество символов>)` или `NATIONAL CHAR (<Количество символов>)` — символьная строка фиксированной длины. В таком столбце всегда хранится указанное количество символов, при необходимости значение дополняется справа пробелами. Вы можете задать количество символов от 0 до 255. Если количество символов не определено, используется длина строки по умолчанию — 1 символ.

Тип данных `NATIONAL CHAR` отличается от `CHAR` тем, что для столбцов с первым типом используется кодировка UTF-8, в то время как для столбцов со вторым типом можно указать любую кодировку, поддерживаемую MySQL.

- ❑ `VARCHAR (<Максимальное количество символов>)` или `NATIONAL VARCHAR (<Максимальное количество символов>)` — символьная строка

переменной длины, содержащая не более указанного количества символов. Вы можете указать максимальное количество символов от 0 до 65 535, но не более 65 535 байт в сумме для всех столбцов таблицы с типом CHAR, VARCHAR, BINARY или VARBINARY. Таким образом, если во всей таблице вы используете однобайтовую кодировку (где каждому символу соответствует 1 байт, например кодировку KOI8-R, CP-866 или CP-1251), то суммарное количество символов, указанное при описании этих столбцов, не должно превышать 65 535. Если же вы используете кодировку UTF-8 (для которой сервер MySQL выделяет до 3 байт на символ), то суммарное количество символов, указанное при описании этих столбцов, не должно превышать 21 844 (в три раза меньше, чем для однобайтовых кодировок).

Тип данных NATIONAL VARCHAR отличается от VARCHAR тем, что для столбцов с типом NATIONAL VARCHAR используется кодировка UTF-8, в то время как для столбцов с типом VARCHAR можно указать любую кодировку, поддерживаемую MySQL.

- ❑ BINARY (<Количество байтов>) — байтовая (бинарная) строка фиксированной длины. Этот тип аналогичен типу CHAR, только строка содержит не символы, а байты, и значение меньшей длины дополняется справа не пробелами, а нулевыми байтами.
- ❑ VARBINARY (<Максимальное количество байтов>) — байтовая (бинарная) строка переменной длины. Этот тип аналогичен типу VARCHAR, только строка содержит не символы, а байты.
- ❑ TINYBLOB — байтовая (бинарная) строка переменной длины. Максимальная длина — 255 байт.
- ❑ TINYTEXT — символьная строка переменной длины. Максимальная длина — 255 байт (не символов!).

ПРИМЕЧАНИЕ



Обратите внимание, что для типов данных TINYTEXT, TEXT, MEDIUMTEXT или LONGTEXT длина значения ограничена максимальным количеством байтов, а не символов. Для однобайтовых кодировок (таких как KOI8-R, CP-866 или CP-1251) длина значения в байтах и в символах одинакова. Однако для многобайтовых кодировок реальное количество символов в значении может быть меньше, чем количество байтов. Так, в кодировке UTF-8 для кодирования символов английского алфавита используется 1 байт на символ, для русского алфавита — 2 байта на символ, поэтому максимальное количество символов русского алфавита, которое можно ввести в такой столбец, приблизительно в два раза меньше, чем максимальное допустимое количество байтов для этого столбца.

- ❑ `BLOB[(<Максимальное количество байтов>)]` — байтовая (бинарная) строка переменной длины. Если количество байтов не указано, то значение столбца ограничено 65 535 байтами. Если количество байтов указано, то создается столбец с типом данных `TINYBLOB`, `BLOB`, `MEDIUMBLOB` или `LONGBLOB`: выбирается тип данных с наименьшим размером, достаточным для хранения этого количества байтов.
- ❑ `TEXT[(<Максимальное количество символов>)]` — символьная строка переменной длины. Если количество символов не указано, то значение столбца ограничено 65 535 байтами. Если количество символов указано, то создается столбец с типом данных `TINYTEXT`, `TEXT`, `MEDIUMTEXT` или `LONGTEXT`: выбирается тип данных с наименьшим размером, достаточным для хранения этого количества символов.
- ❑ `MEDIUMBLOB` — байтовая (бинарная) строка переменной длины. Максимальная длина — 16 777 215 байт.
- ❑ `MEDIUMTEXT` — символьная строка переменной длины. Максимальная длина — 16 777 215 байт.
- ❑ `LONGBLOB` — байтовая (бинарная) строка переменной длины. Максимальная длина — не более 4 294 967 295 байт (4 Гбайт), в зависимости от используемого протокола взаимодействия с сервером MySQL и доступных системных ресурсов.
- ❑ `LONGTEXT` — символьная строка переменной длины. Максимальная длина — не более 4 294 967 295 байт (4 Гбайт), в зависимости от используемого протокола взаимодействия с сервером MySQL и доступных системных ресурсов.
- ❑ `ENUM('<значение 1>', '<значение 2>', ...)` — строка, содержащая ровно один элемент из заданного списка. Например, если столбец определен как `ENUM('a', 'b')`, то допустимыми значениями этого столбца являются 'a', 'b' и `NULL` (а также пустая строка "", которая может появиться при попытке вставки некорректного значения в данный столбец; о добавлении строк в таблицу и возможных вариантах обработки некорректных значений будет сказано чуть ниже). В список вы можете включить до 65 535 элементов.
- ❑ `SET('<значение 1>', '<значение 2>', ...)` — строка, содержащая любой набор элементов из заданного списка (в том числе пустой). Например, если столбец определен как `SET('a', 'b')`, то он может содержать значения "" (пустая строка), 'a', 'b', 'a,b' и `NULL`. В список вы можете включить до 64 элементов. Элементы списка не должны содержать запятых. Каждый из

элементов может присутствовать в значении столбца только один раз, причем элементы должны следовать только в том порядке, в котором они перечислены в списке. Например, при вставке значений 'a, b, a, b' и 'b, a' они автоматически преобразуются в значение 'a, b'.

В заключение отмечу, что в MySQL можно указать кодировку отдельно для каждого символьного столбца. А именно для столбцов с типом CHAR, VARCHAR, TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT, ENUM и SET разрешено задать свойство CHARACTER SET <имя кодировки> и/или COLLATE <имя правила сравнения>.

Например, чтобы имена клиентов хранились в кодировке CP-1251, тогда как кодировкой по умолчанию для таблицы Customers является UTF-8, столбец name можно определить следующим образом:

```
name VARCHAR(100)
CHARACTER SET cp1251 COLLATE cp1251_general_ci
```

Если кодировка для столбца не задана, то используется кодировка, определенная для таблицы в целом (см. ниже). Если не задана кодировка и для таблицы, то используется кодировка, установленная для базы данных. Наконец, если и для базы данных не была указана кодировка, то используется кодировка, установленная в качестве кодировки по умолчанию при настройке MySQL.

Итак, мы рассмотрели типы данных, которые вы можете назначать столбцам таблицы, а также свойства, специфичные для отдельных типов столбцов: свойства UNSIGNED, ZEROFILL и AUTO_INCREMENT для числовых столбцов и свойства CHARACTER SET и COLLATE для символьных столбцов. Перейдем теперь к свойствам, используемым независимо от типа столбцов.

Свойства столбцов

При создании или изменении таблицы вы можете указать следующие свойства столбцов.

- ❑ NOT NULL — указывает, что в данном столбце не допускаются неопределенные значения (NULL).

В качестве примера рассмотрим столбец product_id таблицы Orders (см. листинг 3.4 выше), который мы определили как:

```
product_id BIGINT UNSIGNED NOT NULL
```

Тем самым мы запретили неопределенные номера товаров, поскольку регистрировать заказ с неизвестным товаром не имеет смысла.

Если для столбца задано свойство `NOT NULL`, то, в частности, `NULL` не может использоваться в качестве значения по умолчанию для этого столбца. Значение по умолчанию, отличное от `NULL`, вы можете задать свойством `DEFAULT <значение>`, которое описано ниже. Если же вы задали для столбца свойство `NOT NULL`, но не определили значение по умолчанию и не указали значение для этого столбца при вставке строки в таблицу, то поведение программы MySQL зависит от того, в каком режиме вы работаете (об этом мы подробно расскажем далее в этой главе).

- ❑ `NULL` — указывает, что в столбце разрешены неопределенные значения (`NULL`). Задавать это свойство имеет смысл только для столбцов с типом `TIMESTAMP`, которые по умолчанию не допускают неопределенных значений. Остальные типы столбцов допускают неопределенные значения, если только для них не задано свойство `NOT NULL`.
- ❑ `DEFAULT <значение>` — определяет значение по умолчанию для столбца, которое используется, если при вставке строки в таблицу значение столбца не задано явно. Значением по умолчанию может быть только константа; исключения составляют столбцы с типом `TIMESTAMP`, для которых в качестве значения по умолчанию можно задать переменную величину `CURRENT_TIMESTAMP` (текущие дату и время). Нельзя установить значение по умолчанию для столбцов с типом `TINYBLOB`, `TINYTEXT`, `BLOB`, `TEXT`, `MEDIUMBLOB`, `MEDIUMTEXT`, `LONGBLOB` и `LONGTEXT`, а также для числовых столбцов, для которых задано свойство `AUTO_INCREMENT`. Кроме того, нельзя использовать неопределенное значение по умолчанию (`NULL`), если для столбца задано свойство `NOT NULL`.

Например, чтобы задать для поля `phone` таблицы `Customers` значение по умолчанию, равное пустой строке, можно определить это поле следующим образом:

```
phone VARCHAR(20) DEFAULT ''
```

- ❑ `COMMENT 'Текст комментария'` — произвольное текстовое описание столбца длиной до 255 символов. Например, описание для поля `rating` таблицы `Customers` можно задать следующим образом:

```
rating INT COMMENT 'Рейтинг клиента'
```

Помимо данных свойств, для столбца можно задать свойства `UNIQUE` и `PRIMARY KEY`, однако соответствующие настройки ключевых столбцов и индексов можно указать и после определения всех столбцов таблицы. Мы будем рассматривать

только второй вариант создания ключевых столбцов и индексов. Об этом и пойдет речь в следующем подразделе.

Ключевые столбцы и индексы

После того как определены все столбцы таблицы, можно перечислить через запятую ключевые столбцы и индексы (см. листинги 3.1–3.4). В частности, допустимо использовать следующие конструкции.

- ❑ `[CONSTRAINT <Имя ключа>] PRIMARY KEY (<Список столбцов>)` — определяет первичный ключ таблицы (о первичных ключах мы говорили в начале главы). В таблице может быть только один первичный ключ, который может состоять из одного или нескольких столбцов. Столбцам, входящим в первичный ключ, автоматически присваивается свойство `NOT NULL`. Ключевое слово `CONSTRAINT` и имя ключа можно опустить, так как для первичного ключа заданное имя игнорируется и используется имя `PRIMARY`.

Если в состав первичного ключа входят столбцы с типом `TINYBLOB`, `TINYTEXT`, `BLOB`, `TEXT`, `MEDIUMBLOB`, `MEDIUMTEXT`, `LONGBLOB` и `LONGTEXT`, то необходимо указать количество символов в начале значения столбца; при этом первичный ключ содержит не полные значения столбца, а только начальные подстроки значений.

Пример определения первичного ключа:

```
PRIMARY KEY (id)
```

Именно так мы создали первичный ключ для таблиц `Customers`, `Orders` и `Products` (см. листинги 3.2–3.4). Если бы мы решили не использовать дополнительный столбец `id` в таблице `Products`, а образовать первичный ключ из столбцов `description` и `details`, то в команду создания таблицы `Products` нужно было бы включить определение:

```
PRIMARY KEY (description,details(10))
```

В этом случае в первичный ключ вошли бы столбец `description` и начальные подстроки значений столбца `details` длиной 10 символов.

- ❑ `INDEX [<Имя индекса>] (<Список столбцов>)` — создает индекс для указанных столбцов. Индекс — это вспомогательный объект, позволяющий значительно повысить производительность запросов с условием на значение столбцов, включенных в индекс. Например, чтобы создать индекс для быстрого поиска по именам клиентов, в команду создания таблицы `Customers` (см. листинг 3.2) можно включить определение:

```
INDEX (name)
```

Аналогично первичному ключу, при создании индекса для столбцов с типом `TINYBLOB`, `TINYTEXT`, `BLOB`, `TEXT`, `MEDIUMBLOB`, `MEDIUMTEXT`, `LONGBLOB` и `LONGTEXT` необходимо указать количество символов в начале значения столбца, по которым будет проведено индексирование.

Имя индекса указывать не обязательно. Если вы не зададите имя индекса, оно будет автоматически сгенерировано.

Вместо ключевого слова `INDEX` можно использовать ключевое слово `KEY`, эти слова являются синонимами.

- ❑ `[CONSTRAINT <Имя ограничения>] UNIQUE [<Имя индекса>] (<Список столбцов>)` — создает уникальный индекс для указанных столбцов. Уникальный индекс отличается от обычного наличием дополнительного ограничения: наборы значений в столбцах, включенных в уникальный индекс, должны быть различны. Иными словами, в таблице не должно быть строк, у которых значения во всех этих столбцах совпадают. Исключение составляют неопределенные значения (`NULL`): индекс может содержать два (и более) одинаковых набора значений, если хотя бы одно из значений в этих наборах — `NULL`. Например, ограничение:

`UNIQUE (address, phone)`

запрещает добавлять в таблицу `Customers` две строки, в которых и адрес, и номер телефона определены и совпадают, но разрешает добавлять строки, в которых адрес совпадает, а номер телефона не определен (то есть столбец `phone` в обеих строках содержит значение `NULL`), а также строки, в которых и адрес, и номер телефона не определены.

Для столбцов с типом `TINYBLOB`, `TINYTEXT`, `BLOB`, `TEXT`, `MEDIUMBLOB`, `MEDIUMTEXT`, `LONGBLOB` и `LONGTEXT` необходимо указать количество символов в начале значения столбца, по которым будет проведено индексирование.

Имя ограничения и имя индекса указывать не обязательно. Если ни имя ограничения, ни имя индекса не указаны, то имя индекса присваивается программой автоматически.

Вместо ключевого слова `UNIQUE` можно использовать его синонимы — выражения `UNIQUE INDEX` или `UNIQUE KEY`.

- ❑ `FULLTEXT [<Имя индекса>] (<Список столбцов>)` — создает полнотекстовый индекс для указанных столбцов. Полнотекстовый индекс обеспечивает ускоренный поиск по значениям символьных столбцов (типы `CHAR`, `VARCHAR`, `TINYTEXT`, `TEXT`, `MEDIUMTEXT` и `LONGTEXT`) независимо от длины значений.

Такой индекс подобен предметному указателю в книге: он представляет собой список всех слов, встречающихся в значениях столбцов, со ссылками на те значения, в которых каждое слово содержится.

Полнотекстовый индекс можно создать только в таблицах с типом `MYSAM`. Для поиска с использованием полнотекстового индекса предназначен оператор `MATCH . . . AGAINST`, о котором мы поговорим в главе 4.

Имя индекса указывать не обязательно. Если вы не зададите имя индекса, оно будет автоматически сгенерировано.

Вместо ключевого слова `FULLTEXT` можно использовать его синонимы — выражения `FULLTEXT INDEX` или `FULLTEXT KEY`.

- ❑ `SPATIAL [<Имя индекса>] (<Список столбцов>)` — создает индекс для поиска по пространственным и географическим значениям, которые остаются за рамками этой книги.
- ❑ `[CONSTRAINT <Имя внешнего ключа>] FOREIGN KEY [<Имя индекса>] (<Список столбцов>) REFERENCES <Имя родительской таблицы> (<Список столбцов первичного ключа родительской таблицы>) [<Правила поддержания целостности связи>]` — определяет внешний ключ таблицы, которые мы рассматривали в начале главы. Настроив внешний ключ, мы тем самым создадим связь между данной (дочерней) таблицей и родительской таблицей. Внешние ключи поддерживаются только для таблиц с типом `InnoDB` (причем и дочерняя и родительская таблица должны иметь тип `InnoDB`), для остальных типов таблиц выражение `FOREIGN KEY` игнорируется.

Столбцы, составляющие внешний ключ, должны иметь типы, аналогичные типам столбцов первичного ключа в родительской таблице. Для числовых столбцов должны совпадать размер и знак, для символьных — кодировка и правило сравнения значений. Столбцы с типом `TINYBLOB`, `TINYTEXT`, `BLOB`, `TEXT`, `MEDIUMBLOB`, `MEDIUMTEXT`, `LONGBLOB` и `LONGTEXT` не могут входить во внешний ключ.

Имя внешнего ключа и имя индекса указывать не обязательно. Если вы не зададите эти имена, они будут автоматически сгенерированы.

Вы можете также указать, какие именно правила поддержания целостности связи необходимо использовать для операций удаления и для операций изменения строк родительской таблицы (все эти правила мы обсуждали в начале этой главы). Для операций удаления вы можете указать одно из следующих выражений:

- `ON DELETE CASCADE` — каскадное удаление строк дочерней таблицы (строка родительской таблицы удаляется вместе со всеми ссылающимися на нее строками дочерней таблицы);

- `ON DELETE SET NULL` — обнуление значений внешнего ключа в соответствующих строках дочерней таблицы;
- `ON DELETE RESTRICT` или `ON DELETE NO ACTION` (в MySQL эти выражения являются синонимами) — запрет удаления строк родительской таблицы при наличии ссылающихся на них строк дочерней таблицы.

Если вы не задали правило поддержания целостности для операций удаления, то по умолчанию используется правило `ON DELETE RESTRICT`.

Для операций изменения строк родительской таблицы вы можете указать одно из следующих выражений:

- `ON UPDATE CASCADE` — каскадное обновление значений внешнего ключа дочерней таблицы (вместе со значением первичного ключа в строке родительской таблицы изменяется значение внешнего ключа во всех ссылающихся на нее строках дочерней таблицы);
- `ON UPDATE SET NULL` — обнуление значений внешнего ключа в соответствующих строках дочерней таблицы;
- `ON UPDATE RESTRICT` или `ON UPDATE NO ACTION` (в MySQL эти выражения являются синонимами) — запрет изменения значений первичного ключа в строках родительской таблицы при наличии ссылающихся на них строк дочерней таблицы.

Если вы не задали правило поддержания целостности для операций изменения, то по умолчанию используется правило `ON UPDATE RESTRICT`.

Для столбцов внешнего ключа автоматически создается индекс, поэтому проверки значений внешних ключей в ходе контроля целостности связи выполняются быстро.

Пример определения внешнего ключа в таблице `Orders` (см. листинг 3.4):

```
FOREIGN KEY (product_id) REFERENCES Products (id)
ON DELETE RESTRICT ON UPDATE CASCADE
```

Это выражение означает, что столбец `product_id` таблицы `Orders` является внешним ключом, который ссылается на столбец `id` родительской таблицы `Products`. При этом запрещается удаление строки таблицы `Products`, если на нее ссылается хотя бы одна строка таблицы `Orders`, а изменение значения в столбце `id` таблицы `Products` приводит к автоматическому обновлению значений столбца `product_id` таблицы `Orders`.

Итак, мы изучили индексы и ключи, которые можно настроить при создании таблицы. Наконец, рассмотрим последнюю составляющую команды создания таблицы, а именно опциональные свойства таблицы.

Опциональные свойства таблицы

При создании таблицы указывать опциональные свойства не обязательно. Тем не менее рассмотрим отдельные свойства, которые вы можете задать для таблицы.

- ❑ `ENGINE <Тип таблицы>` — в начале главы говорилось про типы таблиц и указывалось, что все таблицы нашего примера (см. листинги 3.2–3.4) были созданы с параметром `ENGINE InnoDB`. Это дало нам возможность настроить внешние ключи в таблице `Orders` для поддержания целостности связей этой таблицы с таблицами `Customers` и `Products`.
- ❑ `AUTO_INCREMENT <Начальное значение>` — задание этого свойства для таблицы, в которой есть столбец со свойством `AUTO_INCREMENT`, позволяет начать нумерацию в этом столбце не с единицы, а с указанного вами значения. Например, если номера заказов должны начинаться с 1000, нужно в команду создания таблицы `Orders` (см. листинг 3.4) включить параметр `AUTO_INCREMENT 1000`.
- ❑ `CHARACTER SET <Имя кодировки>` — определяет кодировку по умолчанию для символьных столбцов таблицы.

Все таблицы нашего примера (см. листинги 3.2–3.4) были созданы с параметром `CHARACTER SET utf8`. Поэтому все данные о клиентах и товарах будут храниться в этой кодировке.

Если кодировка для таблицы не задана, то по умолчанию используется кодировка, установленная для базы данных. Если и для базы данных кодировка не была указана, то используется кодировка, установленная в качестве кодировки по умолчанию при настройке MySQL, в пакете XAMPP это кодировка `utf8`. Подробнее о кодировках и правилах сравнения символьных значений мы говорили выше в этой главе.

- ❑ `COLLATE <Имя правила сравнения>` — определяет правило сравнения значений, используемое по умолчанию для символьных столбцов таблицы.
- Если правило сравнения для таблицы не задано, то по умолчанию используется правило, установленное для базы данных (см. выше).
- ❑ `CHECKSUM 1` — включает проверку контрольной суммы для строк таблицы типа `MyISAM`, что позволяет быстро обнаруживать поврежденные таблицы.
 - ❑ `COMMENT 'Текст комментария'` — произвольное текстовое описание таблицы, длиной до 60 символов. Например, описание для таблицы `Customers` можно задать, включив в команду создания таблицы параметр `COMMENT 'Сведения о клиентах'`.

Прочие опциональные параметры таблицы используются в тех целях, которые в данной книге не рассматриваются.

На этом мы завершаем изучение команды создания таблицы — `CREATE TABLE`. В следующем подразделе мы рассмотрим команду, с помощью которой можно изменить структуру уже существующей таблицы.

Изменение структуры таблицы

В этом разделе рассказывается, как изменить те параметры таблицы, которые мы обсуждали в предыдущем подразделе. Для модификации ранее созданной таблицы используется команда `ALTER TABLE`. Задавая различные параметры этой команды, вы можете внести в таблицу следующие изменения.

❑ Добавить столбец можно командой:

```
ALTER TABLE <Имя таблицы>
```

```
ADD <Имя столбца> <Тип столбца> [<Свойства столбца>]
```

```
[FIRST или AFTER <Имя предшествующего столбца>];
```

В этой команде мы указываем имя таблицы, в которую добавляется столбец, а также имя и тип добавляемого столбца. При необходимости можно также задать свойства этого столбца. Кроме того, можно определить место нового столбца среди уже существующих: добавляемый столбец может стать первым (`FIRST`) или следовать после указанного предшествующего столбца (`AFTER`). Если место столбца не задано, то он становится последним столбцом таблицы.

Например, чтобы добавить в таблицу `Products` столбец `store` (название склада, где хранится каждый вид товара), выполним команду:

```
ALTER TABLE Products ADD store VARCHAR(100) AFTER details;
```



ПРИМЕЧАНИЕ

Добавить столбец со свойством `AUTO_INCREMENT` можно только с одновременным созданием индекса или ключа для этого столбца. Например, если бы столбец `id` не был включен в таблицу `Products` при ее создании, мы могли бы добавить его с помощью команды:

```
ALTER TABLE Products ADD id BIGINT AUTO_INCREMENT, ADD PRIMARY KEY (id);
```

❑ Добавить первичный ключ вы можете командой:

```
ALTER TABLE <Имя таблицы>
```

```
ADD [CONSTRAINT <Имя ключа>]
```

```
PRIMARY KEY (<Список столбцов>);
```

При добавлении в таблицу первичного ключа мы указываем имя таблицы, в которую нужно добавить ключ, и имена столбцов, которые будут образовывать первичный ключ (эти столбцы должны уже существовать в таблице).

Например, если бы первичный ключ не был определен при создании таблицы `Orders`, мы могли бы добавить его с помощью команды:

```
ALTER TABLE Orders ADD PRIMARY KEY (id);
```

❑ Добавить внешний ключ вы можете командой:

```
ALTER TABLE <Имя таблицы>  
ADD [CONSTRAINT <Имя внешнего ключа>  
FOREIGN KEY [<Имя индекса>] (<Список столбцов>  
REFERENCES <Имя родительской таблицы>  
(<Список столбцов первичного ключа родительской таблицы>  
[<Правила поддержания целостности связи>];
```

При добавлении в таблицу внешнего ключа мы указываем имя таблицы, в которую нужно добавить ключ, имена столбцов, которые будут образовывать внешний ключ (эти столбцы должны уже существовать в таблице), а также имя родительской таблицы (на нее будет ссылаться данная таблица) и имена столбцов, образующих первичный ключ в родительской таблице.

При необходимости можно также задать имя создаваемого внешнего ключа, имя индекса, автоматически добавляемого для столбцов внешнего ключа, и правила поддержания целостности связи при удалении и изменении строк родительской таблицы.

Например, если бы внешний ключ не был определен при создании таблицы `Orders`, мы могли бы добавить его командой

```
ALTER TABLE Orders  
ADD FOREIGN KEY (customer_id) REFERENCES Customers (id)  
ON DELETE RESTRICT ON UPDATE CASCADE);
```

❑ Добавить в таблицу обычный индекс вы можете с помощью команды:

```
ALTER TABLE <Имя таблицы>  
ADD INDEX [<Имя индекса>] (<Список столбцов>);
```

Добавить уникальный индекс — с помощью команды:

```
ALTER TABLE <Имя таблицы>  
ADD [CONSTRAINT <Имя ограничения>]  
UNIQUE (<Список столбцов>);
```

Добавить полнотекстовый индекс — с помощью команды:

```
ALTER TABLE <Имя таблицы>
```

```
ADD FULLTEXT [<Имя индекса>] (<Список столбцов>);
```

При добавлении в таблицу индекса мы указываем имя таблицы, в которую нужно добавить индекс, и имена столбцов, включенных в индекс. При необходимости можно также задать имя индекса.

Например, добавить индекс для столбца `store` таблицы `Products` можно с помощью команды:

```
ALTER TABLE Products ADD INDEX (store);
```

❑ Изменить столбец таблицы вы можете с помощью следующих команд.

- Чтобы полностью изменить описание столбца, выполним команду:

```
ALTER TABLE <Имя таблицы>
```

```
CHANGE <Прежнее имя столбца>
```

```
<Новое имя столбца>
```

```
<Новый тип столбца> [<Свойства столбца>]
```

```
[FIRST или AFTER <Имя предшествующего столбца>];
```

ВНИМАНИЕ



Изменять тип столбца, который уже содержит какие-либо значения, необходимо с осторожностью, так как при этом возможно внесение корректировок в значения. Например, если тип данных с плавающей точкой меняется на целочисленный, то числовые значения будут округлены.

В этой команде мы указываем имя таблицы, текущее имя столбца, новое имя столбца (которое может совпадать с текущим), новый тип столбца (который также может совпадать с текущим типом), а также при необходимости свойства столбца (старые свойства при выполнении команды `CHANGE` удаляются). Кроме того, можно определить место столбца среди остальных столбцов таблицы: изменяемый столбец может стать первым (`FIRST`) или следовать после указанного предшествующего столбца (`AFTER`).

Например, чтобы переименовать столбец `store` таблицы `Products` в `warehouse` (склад) и изменить его тип на `CHAR(100)`, выполним такую команду:

```
ALTER TABLE Products CHANGE store warehouse CHAR(100);
```

Чтобы присвоить столбцу свойство `AUTO_INCREMENT`, необходимо либо одновременно с этим, либо заранее создать индекс для данного столбца.

- Чтобы изменить описание столбца без переименования, выполним команду:

```
ALTER TABLE <Имя таблицы>
```

```
MODIFY <Имя столбца>
```

```
<Новый тип столбца> [<Свойства столбца>]
```

```
[FIRST или AFTER <Имя предшествующего столбца>];
```

Она полностью аналогична предыдущей, за исключением возможности переименования столбца.

- Чтобы установить значение по умолчанию для столбца, выполним команду:

```
ALTER TABLE <Имя таблицы>
```

```
ALTER <Имя столбца>
```

```
SET DEFAULT <Значение по умолчанию>;
```

Например, чтобы установить для столбца `warehouse` таблицы `Products` значение по умолчанию "Склад №1", выполним команду:

```
ALTER TABLE Products
```

```
ALTER warehouse SET DEFAULT 'Склад №1';
```

Чтобы удалить значение по умолчанию, выполним команду:

```
ALTER TABLE <Имя таблицы>
```

```
ALTER <Имя столбца> DROP DEFAULT;
```

Например, удалить значение по умолчанию, установленное для столбца `warehouse`, можно с помощью команды:

```
ALTER TABLE Products ALTER warehouse DROP DEFAULT;
```

- ❑ Удалить столбец таблицы вы можете с помощью такой команды:

```
ALTER TABLE <Имя таблицы> DROP <Имя столбца>;
```

При удалении столбца он удаляется также из всех индексов, в которые был включен (в отличие от первичного и внешнего ключа — их требуется удалить прежде, чем удалять входящие в них столбцы). Если при этом в индексе не остается столбцов, то он также автоматически удаляется.

Например, для удаления столбца `warehouse` из таблицы `Products` выполним команду:

```
ALTER TABLE Products DROP warehouse;
```

- ❑ Удалить первичный ключ вы можете с помощью команды:

```
ALTER TABLE <Имя таблицы> DROP PRIMARY KEY;
```

Например, удалить первичный ключ из таблицы `Orders` можно с помощью команды:

```
ALTER TABLE Orders DROP PRIMARY KEY;
```

❑ Удалить внешний ключ вы можете командой:

```
ALTER TABLE <Имя таблицы>  
DROP FOREIGN KEY <Имя внешнего ключа>;
```

В этой команде необходимо указать имя внешнего ключа. Если вы не задали имя внешнего ключа при его создании, то имя было присвоено автоматически и узнать его можно с помощью команды `SHOW CREATE TABLE` (мы обсудим ее чуть ниже).

Например, удалить внешний ключ из таблицы `Orders` можно с помощью команды:

```
ALTER TABLE Orders DROP FOREIGN KEY orders_ibfk_1;
```

(здесь `orders_ibfk_1` — имя внешнего ключа, автоматически присвоенное при создании этого ключа).

❑ Удалить индекс (обычный, уникальный или полнотекстовый) вы можете с помощью команды:

```
ALTER TABLE <Имя таблицы> DROP INDEX <Имя индекса>;
```

Здесь необходимо указать имя индекса. Если вы не задали имя индекса при его создании, то оно было присвоено автоматически и узнать его можно с помощью команды `SHOW CREATE TABLE` (см. ниже).

Например, удалить индекс, созданный для поля `name` таблицы `Customers`, можно с помощью команды:

```
ALTER TABLE Customers DROP INDEX name;
```

(здесь `name` — имя индекса: по умолчанию индексу присваивается имя первого индексируемого столбца).

❑ Включить и отключить обновление неуникальных индексов в таблице с типом `MyISAM` вы можете с помощью следующей команды:

```
ALTER TABLE <Имя таблицы> DISABLE KEYS;
```

Она временно отключает обновление неуникальных индексов, что позволяет увеличить быстродействие операций добавления строк в таблицу.

После того как строки добавлены, восстановить индексы можно, выполнив команду:

```
ALTER TABLE <Имя таблицы> ENABLE KEYS;
```

- ❑ Переименовать таблицу вы можете с помощью команды:

```
ALTER TABLE <Имя таблицы> RENAME <Новое имя таблицы>;
```

- ❑ Упорядочить строки таблицы по значениям одного или нескольких столбцов можно, выполнив команду:

```
ALTER TABLE <Имя таблицы>  
ORDER BY <Имя столбца 1> [ASC или DESC],  
[<Имя столбца 2> [ASC или DESC],...];
```

Упорядочение строк позволяет ускорить последующие операции сортировки по значениям указанных столбцов. Однако порядок строк в таблице может нарушиться после операций добавления и удаления строк.

По умолчанию строки таблицы упорядочиваются по возрастанию значений. Вы можете выбрать направление сортировки, указав ключевое слово ASC (по возрастанию) или DESC (по убыванию).

Для таблиц с типом InnoDB, в которых есть первичный ключ или уникальный индекс, не допускающий неопределенных значений (NOT NULL UNIQUE), эта команда игнорируется, поскольку строки таких таблиц автоматически упорядочиваются по значениям этого ключа/индекса.

- ❑ Задать опциональные свойства таблицы можно с помощью команды:

```
ALTER TABLE <Имя таблицы> <Опциональное свойство таблицы>;
```

Например, изменить тип таблицы можно, выполнив команду:

```
ALTER TABLE <Имя таблицы> ENGINE <Новый тип таблицы>;
```

- ❑ Изменить кодировку и правило сравнения символьных значений, используемые по умолчанию для новых символьных столбцов таблицы, можно, как любое другое опциональное свойство таблицы, с помощью команды:

```
ALTER TABLE <Имя таблицы>  
CHARACTER SET <Имя кодировки>  
[COLLATE <Имя правила сравнения>;]
```

Например, если для таблицы `Products` требуется установить в качестве кодировки по умолчанию кодировку CP-1251, это можно сделать с помощью команды:

```
ALTER TABLE Products CHARACTER SET cp1251;
```

После выполнения этой команды существующие столбцы таблицы `Products` по-прежнему будут иметь кодировку UTF-8. Если же в таблицу будут добав-

ляться новые символьные столбцы с помощью команды `ALTER TABLE`, то им будет по умолчанию присваиваться кодировка CP-1251.

Если вы хотите не только изменить кодировку, используемую по умолчанию для новых столбцов, но и преобразовать в новую кодировку существующие символьные столбцы таблицы, то нужно выполнить команду:

```
ALTER TABLE <Имя таблицы>  
  CONVERT TO CHARACTER SET <Имя кодировки>  
  [COLLATE <Имя правила сравнения>];
```

В результате не просто меняются описания символьных столбцов, но данные в этих столбцах также преобразуются в новую кодировку.

Например, после выполнения команды:

```
ALTER TABLE Products CONVERT TO CHARACTER SET cp1251;
```

все наименования и описания товаров (значения столбцов `description` и `details`) будут преобразованы в кодировку CP-1251.

Если же данные в столбце фактически закодированы с помощью одной кодировки, а в описании столбца указана другая, то исправить эту ошибку можно, изменив кодировку *только в описании столбца*, без преобразования данных. Для этого можно преобразовать столбец из символьного типа в бинарный (то есть тип `CHAR` в тип `BINARY`, тип `VARCHAR` — в тип `VARBINARY`, тип `TEXT` — в тип `BLOB` и т. п.), а затем обратно в символьный, уже в правильной кодировке.

Выполним последовательно две команды:

```
ALTER TABLE <Имя таблицы>  
  CHANGE <Имя столбца> <Имя столбца> <Бинарный тип данных>;  
ALTER TABLE <Имя таблицы>  
  CHANGE <Имя столбца> <Имя столбца> <Исходный тип данных>  
  CHARACTER SET <Фактическая кодировка значений>;
```

Например, если для столбца `details` таблицы `Products` установлены тип `TEXT` и кодировка UTF-8, а данные в нем фактически находятся в кодировке CP-1251, приведем описание столбца в соответствие с реальной кодировкой с помощью команд:

```
ALTER TABLE Products CHANGE details details BLOB;  
ALTER TABLE Products  
  CHANGE details details TEXT CHARACTER SET cp1251;
```

Итак, мы изучили команду изменения таблицы. В следующем подразделе мы рассмотрим еще несколько полезных команд работы с таблицами.

Другие команды работы с таблицами

В этом разделе мы познакомимся с командами для получения информации о таблицах, а также с командой для удаления таблицы.

Получить детальную информацию о конкретной таблице вы можете с помощью одной из команд:

```
DESCRIBE <Имя таблицы>;
```

или

```
SHOW CREATE TABLE <Имя таблицы>;
```

Эти команды можно использовать, чтобы, например, узнать имена и порядок следования столбцов таблицы, проверить правильность изменений, внесенных в структуру таблицы с помощью команды `ALTER TABLE`, и т. п.

Команда `DESCRIBE` выводит информацию о столбцах таблицы. Например, для получения информации о столбцах таблицы `Customers` выполним команду:

```
DESCRIBE Customers;
```

Результат представлен в табл. 3.3.

Таблица 3.3. Результат выполнения команды `DESCRIBE Customers`;

Field	Type	Null	Key	Default	Extra
id	bigint(20) unsigned	NO	PRI	NULL	auto_increment
name	varchar(100)	YES		NULL	
phone	varchar(20)	YES		NULL	
address	varchar(150)	YES		NULL	
rating	int(11)	YES		NULL	

Для каждого столбца таблицы команда `DESCRIBE` отображает следующие характеристики:

- ❑ `Field` — имя столбца;
- ❑ `Type` — тип столбца;
- ❑ `Null` — информация, указывающая, допускает ли столбец неопределенные значения (`NULL`): `YES` — допускает, `NO` — не допускает;
- ❑ `Key` — вхождение столбца в ключи и индексы:
 - `PRI` — столбец входит в первичный ключ или, если в таблице нет первичного ключа, в уникальный индекс, не допускающий неопределенных значений;

- UNI, MUL — столбец является первым столбцом индекса;
- ❑ Default — значение столбца по умолчанию;
- ❑ Extra — дополнительная информация.

Команда `SHOW CREATE TABLE` выводит полную информацию обо всех параметрах таблицы в виде текста команды `CREATE TABLE`, позволяющей создать таблицу, идентичную текущей. Эта команда может не совпадать с командой, с помощью которой была в действительности создана таблица, если, например, таблица была изменена командой `ALTER TABLE` или программа MySQL автоматически внесла корректировки в структуру таблицы (например, добавлено значение по умолчанию для столбца или присвоено имя индексу). Например, команда:

```
SHOW CREATE TABLE Customers;
```

выводит следующий результат (табл. 3.4).

Таблица 3.4. Результат выполнения команды `SHOW CREATE TABLE Customers;`

Table	Create Table
Customers	<pre>CREATE TABLE 'customers' ('id' bigint(20) unsigned NOT NULL auto_increment, 'name' varchar(100) default NULL, 'phone' varchar(20) default NULL, 'address' varchar(150) default NULL, 'rating' int(11) default NULL, PRIMARY KEY ('id'), UNIQUE KEY 'id' ('id')) ENGINE=InnoDB DEFAULT CHARSET=utf8</pre>

Если сравнить данные из табл. 3.4 с «настоящей» командой создания таблицы `Customers` (см. листинг 3.2), то вы увидите, как изменились определения столбцов.

Просмотреть список таблиц текущей базы данных вы можете с помощью команды:

```
SHOW TABLES;
```

Если вы выбрали в качестве текущей базу данных `SalesDept` и создали в ней три таблицы — `Customers`, `Products` и `Orders`, то команда `SHOW TABLES` выведет следующий результат (табл. 3.5).

Таблица 3.5. Результат выполнения команды SHOW TABLES;

Tables_in_salesdept
customers
orders
products

И наконец, чтобы удалить ненужную или ошибочно созданную таблицу, выполним команду:

```
DROP TABLE <Имя таблицы>;
```

ВНИМАНИЕ



Удаление таблицы — очень ответственная операция, поскольку она приводит к удалению всех данных, хранившихся в таблице. Желательно перед удалением таблицы создать резервную копию базы данных.

Итак, вы освоили основные операции, выполняемые с таблицами, а именно команды CREATE TABLE (создание), ALTER TABLE (изменение), DROP TABLE (удаление), SHOW TABLES (просмотр списка таблиц), DESCRIBE и SHOW CREATE TABLE (просмотр информации о таблице). Теперь перейдем к работе с отдельными строками.

Ввод данных в таблицу

После создания таблиц можно приступить к заполнению их данными. В текущем разделе мы поговорим о двух операциях, с помощью которых можно добавить строки в таблицу. Сначала мы рассмотрим загрузку данных из текстового файла, а затем — вставку отдельных строк.

Загрузка данных из файла

Если требуется добавить в таблицу большой массив данных, то удобно использовать для этого команду загрузки данных из файла. Загрузка из файла выполняется программой MySQL значительно быстрее, чем вставка строк с помощью команды INSERT, о которой пойдет речь в следующем подразделе.

Например, чтобы загрузить данные в таблицу Customers, команда создания которой показана в листинге 3.2, выполните следующие действия.

1. Запустите стандартную программу Windows Блокнот (Пуск ► Все программы ► Стандартные ► Блокнот).

2. В окне программы Блокнот введите данные, используя для отделения значений клавишу Tab, а для перехода на следующую строку — клавишу Enter (рис. 3.3).

**ПРИМЕЧАНИЕ**

Вместо отсутствующего значения при заполнении файла необходимо ввести символы \N. Тогда в базу данных будет загружено неопределенное значение (NULL).

3. Для сохранения файла с данными нажмите сочетание клавиш Ctrl+S. В стандартном окне Windows Сохранить как выберите папку, в которую нужно поместить файл (например, C:\data). Введите имя файла (например, Customers.txt). Нажмите кнопку Сохранить.

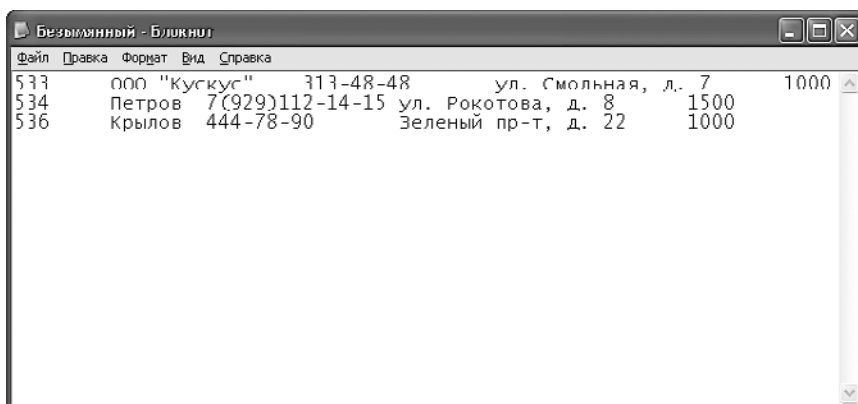


Рис. 3.3. Ввод данных в текстовый файл

4. Для загрузки данных из созданного файла выполните команду:

```
LOAD DATA LOCAL INFILE 'C:/data/Customers.txt'
INTO TABLE Customers
CHARACTER SET cp1251;
```

Обратите внимание, что в пути к файлу необходимо использовать прямую косую черту, а не обратную.

Файл Customers.txt мы создали в формате, принятом по умолчанию в MySQL, поэтому при загрузке потребовалось указать только один дополнительный параметр — кодировку Windows.

Однако, если вам нужно загрузить в таблицу данные из текстового файла, который был создан в другом формате (например, выгружен из другой базы данных), могут

потребоваться и иные параметры. Полностью команда `LOAD DATA` имеет следующий вид:

```
LOAD DATA [LOCAL] INFILE 'Путь и имя файла'
[REPLACE или IGNORE]
INTO TABLE <Имя таблицы>
CHARACTER SET <Имя кодировки>
[
  FIELDS
  [TERMINATED BY <Разделитель значений в строке>]
  [[OPTIONALLY]
   ENCLOSED BY <Символ, в который заключены значения>]
  [ESCAPED BY <Экранирующий символ>]
]
[
  LINES
  [STARTING BY <Префикс строки>]
  [TERMINATED BY <Разделитель строк>]
]
[IGNORE <Количество строк в начале файла> LINES]
[(<Список столбцов>)]
[SET <Имя столбца>=<Выражение>,...];
```

В этой команде вы можете использовать следующие параметры.

- ❑ `LOCAL` — укажите этот параметр, если файл с данными находится на клиентском компьютере (то есть на том компьютере, где работает клиентское приложение, в котором вы и вводите эту команду). Если файл расположен на компьютере, где работает сервер MySQL, то параметр `LOCAL` задавать не нужно.
- ❑ `'Путь и имя файла'` — введите полный путь к файлу, например: `C:/Data/mytable.txt` (необходимо использовать прямую косую черту вместо принятой в Windows обратной косой черты).
- ❑ `REPLACE` или `IGNORE` — укажите один из этих параметров, чтобы сообщить программе MySQL, как нужно обрабатывать загружаемую строку, если в таблице уже есть строка с таким же значением первичного ключа или уникального индекса. Если указан параметр `REPLACE`, то существующая в таблице строка

заменяется новой. Если указан параметр IGNORE, то новая строка в таблицу не загружается.

- ❑ CHARACTER SET <Имя кодировки> — укажите кодировку данных в файле. Этот параметр используется для корректного преобразования кодировок. Предполагается, что все данные в файле имеют одну и ту же кодировку.

ВНИМАНИЕ



Загрузка данных в кодировке UTF-8 может некорректно работать из-за переменного количества байтов на символ в этой кодировке. Желательно файл с данными в этой кодировке перед загрузкой преобразовать в файл в какой-либо однобайтовой кодировке. Например, откройте файл в программе Блокнот, в меню Файл выберите команду Сохранить как, а затем в стандартном окне Windows Сохранить как в поле Кодировка выберите из списка значение ANSI и нажмите кнопку Сохранить. После этого загрузите файл, указав в команде LOAD DATA параметр CHARACTER SET cp1251.

- ❑ FIELDS — добавьте этот параметр, чтобы сообщить программе MySQL, в каком формате заданы значения в файле.
 - TERMINATED BY <Разделитель значений в строке> — укажите символ, разделяющий значения в строке файла. Например, если значения разделены запятыми, задайте параметр TERMINATED BY ',', если значения разделены символами табуляции — TERMINATED BY '\t', если значения разделены косой чертой — TERMINATED BY '/ '.
 - ENCLOSED BY <Символ, в который заключены значения> — определите символ, которым обрамляются значения. Например, если все значения заключены в одинарные кавычки, укажите ENCLOSED BY '\'', если в одинарные кавычки заключены только символьные значения — OPTIONALLY ENCLOSED BY '\'', а если никакие значения не обрамляются никакими символами, укажите ENCLOSED BY '' или вообще опустите этот параметр.
 - ESCAPED BY <Экранирующий символ> — определите экранирующий символ (его также называют escape-символом). Он сообщает программе MySQL, что следующий за ним символ нужно интерпретировать особым образом. Иными словами, обычный символ, следующий после экранирующего, будет рассматриваться как специальный символ, а специальный символ, наоборот, — как обычный символ.

Чаще всего экранирующим символом служит обратная косая черта, и в этом случае устанавливают значение ESCAPED BY '\\'. Тогда, например, записанное в файле значение \N будет прочитано и загружено в базу данных как

NULL. Другой пример: если значения в файле разделены запятыми, то экранирующий символ помещается перед запятой, которая должна восприниматься как часть значения, а не как разделитель, то есть последовательность символов `\,` интерпретируется как символ запятой в значении.

Если параметр `FIELDS` не установлен, то программа MySQL считает, что значения в файле разделяются табуляцией и не заключаются ни в какие кавычки, а в качестве экранирующего символа используется обратная косая черта.

- ❑ `LINE`s — укажите этот параметр, чтобы сообщить программе MySQL, в каком формате заданы строки в файле.
 - `STARTING BY` <Префикс строки> — определите последовательность символов в начале каждой строки, которая должна игнорироваться программой вместе со всеми предшествующими символами. После префикса должны начинаться значения.
 - `TERMINATED BY` <Разделитель строк> — укажите символ, которым заканчиваются строки. Например, если строки заканчиваются символом перевода строки, установите параметр `TERMINATED BY '\n'`, если символом возврата каретки — `TERMINATED BY '\r'`, если сочетанием этих символов — `TERMINATED BY '\r\n'`, если нулевым байтом — `TERMINATED BY '\0'`.

Если параметр `LINE`s не указан, то программа MySQL считает, что строки в файле не имеют префикса и заканчиваются символом перевода строки `\n`.

- ❑ `IGNORE` <Количество строк в начале файла> `LINE`s — укажите этот параметр, если первые несколько строк в файле не содержат значений (иными словами, являются заголовком) и при загрузке их нужно пропустить.
- ❑ (<Список столбцов>) — перечислите столбцы таблицы, в которые будут загружаться данные. Это необходимо, если файл содержит данные не для всех столбцов таблицы или порядок следования значений в файле отличается от порядка столбцов в таблице.
- ❑ `SET` <Имя столбца>=<Выражение> — вы можете записывать в столбцы не только значения из файла, но и значения, вычисленные с помощью выражений. Для этого создайте одну или несколько переменных, присвойте им считанные из файла значения и запишите в столбец значение выражения, использующего эти переменные. Пусть, например, имеется таблица `t1` с числовым столбцом `c1` и столбцом `c2` с типом `TIMESTAMP`. В столбец `c1` нужно загрузить значение из файла, если оно не превосходит 1000, либо `NULL`, если значение в файле

больше 1000, а в столбец c2 при этом нужно записать текущие дату и время. Это можно сделать с помощью команды:

```
LOAD DATA INFILE 'C:/DATA/t1.txt'  
INTO TABLE t1 (@var1)  
SET c1=IF(@var1<=1000,@var1,NULL), c2=CURRENT_TIMESTAMP;
```

(о функции IF и о других функциях, используемых в выражениях, будет рассказано в главе 4).

Далее мы рассмотрим команду INSERT, с помощью которой также можно добавлять строки в таблицу.

Вставка отдельных строк

Для добавления одной или нескольких строк в таблицу можно использовать команду:

```
INSERT [INTO] <Имя таблицы>  
[(<Список столбцов>)]  
VALUES  
(<Список значений 1>),  
(<Список значений 2>),  
...  
(<Список значений N>);
```

В команде INSERT используются следующие основные параметры.

- ❑ Имя таблицы, в которую добавляются строки.
- ❑ Список имен столбцов, для которых будут заданы значения. Если значения будут приведены для всех столбцов таблицы, то приводить список столбцов не обязательно.



ПРИМЕЧАНИЕ

Если столбец таблицы не включен в список, то в этом столбце при добавлении строки будет автоматически установлено значение по умолчанию.

- ❑ Значения, которые нужно добавить в таблицу. Значения указываются в следующем формате.
 - Набор значений для каждой добавляемой строки заключается в скобки. Набор значений внутри каждой пары скобок должен соответствовать указанному списку столбцов, а если список столбцов не указан, то упорядоченному списку

всех столбцов, составляющих таблицу (список столбцов таблицы можно просмотреть с помощью команды `DESCRIBE`, описанной выше). Значения внутри набора отделяются друг от друга запятыми, наборы также отделяются друга от друга запятыми.

- Символьные значения, а также значения даты и времени приводятся в одинарных кавычках. Для числовых значений кавычки необязательны. Десятичным разделителем для чисел с дробной частью служит точка. Даты и время вводятся в формате `YYYY-MM-DD` и `HH:MM:SS` соответственно.
- Чтобы ввести в столбец неопределенное значение, необходимо указать вместо значения ключевое слово `NULL` *без кавычек* (слово в кавычках рассматривается как обычная символьная строка).
- Вместо значения можно указать ключевое слово `DEFAULT` *без кавычек*, тогда в столбец будет введено значение по умолчанию (если оно задано для этого столбца).

Например, добавим в таблицу `Products` сведения о продукции компании с помощью команды, представленной в листинге 3.5.

Листинг 3.5. Команда добавления строк в таблицу `Products`

```
INSERT INTO Products (description,details,price)
VALUES
('Обогреватель Мосбытприбор ВГД 121R',
 'Инфракрасный обогреватель. 3 режима нагрева:
  400 Вт, 800 Вт, 1200 Вт','1145.00'),
('Гриль Мосбытприбор СТ-14',
 'Мощность 1440 Вт. Быстрый нагрев. Термостат.
  Цветовой индикатор работы','2115.00'),
('Кофеварка Мосбытприбор ЕКЛ-1032',
 'Цвет: черный. Мощность: 450 Вт.
  Вместительность: 2 чашки','710.00'),
('Чайник Мосбытприбор МН',
 'Цвет: белый. Мощность: 2200 Вт. Объем: 2 л','925.00'),
('Утюг Мосбытприбор с паром АБ 200',
 'Цвет: фиолетовый. Мощность: 1400 вт','518.00');
```


Эта команда добавляет значения в столбцы `description`, `details` и `price` таблицы `Products`. При этом в столбец `id` автоматически вносятся порядковые номера строк, поскольку он имеет тип данных `SERIAL` (см. листинг 3.3).

Теперь, когда данные внесены и в таблицу `Customers` (в предыдущем подразделе мы говорили о том, как загрузить в эту таблицу данные из файла), и в таблицу `Products`, можно заполнять таблицу `Orders`. (Напомню, что каждая строка таблицы `Orders` ссылается на строку таблицы `Customers` и строку таблицы `Products`, и в момент добавления строки в таблицу `Orders` соответствующие строки в таблицах `Customers` и `Products` должны уже существовать.) Внесем в таблицу `Orders` сведения о заказах, выполнив команду, представленную в листинге 3.6.

Листинг 3.6. Команда добавления строк в таблицу `Orders`

```
INSERT INTO Orders
VALUES
  (1012, '2007-12-12', 5.8, '4500', 533),
  (1013, '2007-12-12', 2.14, '22000', 536),
  (1014, '2008-01-21', 5.12, '5750', 533);
```

Если вы пытаетесь добавить в таблицу некорректное значение, то результат выполнения команды `INSERT` зависит от того, в каком режиме ваше клиентское приложение взаимодействует с сервером `MySQL`. Остановимся на этом подробнее.

Узнать, в каком режиме вы в данный момент работаете, можно с помощью команды:

```
SHOW VARIABLES LIKE 'sql_mode';
```

Она показывает значение переменной `sql_mode`. Если в значении нет ключевых слов `STRICT_TRANS_TABLES` и `STRICT_ALL_TABLES`, то сервер работает в нестрогом режиме.

В нестрогом режиме вставляемое некорректное значение преобразуется в допустимое, в частности:

- ❑ некорректная дата заменяется нулевой датой (`0000-00-00 00:00:00`);
- ❑ «лишние» символы в очень длинном символьном значении отбрасываются (так, значение `'abc '`, вставляемое в столбец с типом `CHAR (2)`, сокращается до `'ab '`);
- ❑ слишком большое число заменяется максимально возможным значением для данного типа столбца;

- ❑ при внесении в числовой столбец символьного значения все символы, начиная с первой буквы, отбрасываются и в таблицу вносится начальная числовая часть значения, а если первый символ в значении — буква, а не цифра, то оно заменится нулем (даже если задано отличное от нуля значение по умолчанию);
- ❑ если вы установили для столбца свойство `NOT NULL`, но не задали значение по умолчанию, а при вставке строки не указали значение для этого столбца, то в столбец будет добавлено следующее значение:
 - для числовых столбцов — 0, а если задано свойство `AUTO_INCREMENT`, то очередной порядковый номер;
 - для столбцов, которые имеют тип даты и времени, — нулевые дата и/или время (0000-00-00 00:00:00), а для первого в таблице столбца с типом `TIMESTAMP` — текущие дата и время;
 - для символьных столбцов — пустая строка, а для столбца с типом `ENUM` — первый из элементов списка.

При этом операция добавления завершается успешно и генерируется предупреждение, которое можно просмотреть, выполнив непосредственно после команды `INSERT` команду:

`SHOW WARNINGS;`

ВНИМАНИЕ



В любом из режимов попытка добавить повторяющееся значение в столбец первичного ключа или уникального индекса вызывает ошибку и прекращение выполнения операции. Кроме того, в любом из режимов вызывает ошибку и отмену операции попытка добавить во внешний ключ таблицы с типом `InnoDB` значение, отсутствующее в первичном ключе родительской таблицы.

Если в значении переменной `sql_mode` содержится ключевое слово `STRICT_TRANS_TABLES` или `STRICT_ALL_TABLES`, то сервер работает в строгом режиме. Для таблиц с поддержкой транзакций (то есть таблиц с типом `InnoDB`) эти два режима эквивалентны и действуют одинаково: при попытке вставки некорректного значения операция `INSERT` полностью отменяется (то есть в таблицу не загружается ни одна из добавляемых строк, даже если некорректное значение обнаружено только в одной из них) и выдается сообщение об ошибке.

Для таблиц, не поддерживающих транзакции (то есть таблиц с типом `MyISAM` и др.), режимы `STRICT_TRANS_TABLES` или `STRICT_ALL_TABLES` функционируют следующим образом.

- ❑ Если некорректное значение обнаружено в первой из добавляемых строк, то операция `INSERT` полностью отменяется и выдается сообщение об ошибке (то есть результат выполнения такой же, как для таблиц `InnoDB`).
- ❑ Если установлен режим `STRICT_TRANS_TABLES`, а некорректное значение обнаружено в одной из последующих добавляемых строк, то некорректное значение преобразуется в допустимое и генерируется предупреждение (как в нестрогом режиме).
- ❑ Если установлен режим `STRICT_ALL_TABLES`, а некорректное значение обнаружено в одной из последующих добавляемых строк, то операция частично отменяется: строки, предшествующие ошибочной, добавляются в таблицу, а ошибочная и следующие за ней — игнорируются. В связи с этим в таблицы, не поддерживающие транзакции, рекомендуется добавлять по одной строке в каждой команде `INSERT`.

ВНИМАНИЕ



В любом из режимов числа с дробной частью, добавляемые в целочисленный столбец, округляются до целого, причем такая операция не вызывает ни ошибок, ни предупреждений.

Если вы задали для столбца таблицы свойство `NOT NULL` (тем самым запретив использовать значение `NULL` в качестве значения по умолчанию), но не задали отличное от `NULL` значение по умолчанию (свойство `DEFAULT`), а также если вы удалили значение столбца по умолчанию командой `ALTER TABLE`, то в строгом режиме считается, что у такого столбца нет значения по умолчанию. Поэтому при добавлении строки в таблицу необходимо явно определить значение для такого столбца, в противном случае операция `INSERT` полностью отменяется и выдается сообщение об ошибке. Исключение составляют столбцы с типом `TIMESTAMP` и `ENUM`, а также числовые столбцы со свойством `AUTO_INCREMENT`: для них при отсутствии явно заданного значения используются такие же значения, что и в нестрогом режиме.

Изменить режим взаимодействия вашего клиентского приложения с сервером вы можете с помощью команды:

```
SET SQL_MODE='<Режим>';
```

Например, команда:

```
SET SQL_MODE='';
```

устанавливает нестрогий режим, а команды:

```
SET SQL_MODE='STRICT_TRANS_TABLES';
```

и

```
SET SQL_MODE='STRICT_ALL_TABLES';
```

устанавливают соответствующий строгий режим.

Команда `SET SQL_MODE` изменяет режим взаимодействия с сервером *только для текущего соединения* и не влияет на взаимодействие сервера с другими клиентскими приложениями. Новый режим вступает в силу немедленно после выполнения команды и сохраняется только до момента отключения от сервера.

Если же вы хотите, чтобы новый режим действовал глобально (то есть для всех клиентских приложений), необходимо включить в команду изменения режима ключевое слово `GLOBAL`:

```
SET GLOBAL SQL_MODE='<Режим>';
```

Режим, установленный глобально, применяется для всех вновь подключающихся к серверу клиентских приложений; ранее подключенные приложения продолжают работать в прежнем режиме. Чтобы новый режим вступил в силу для вашего приложения, необходимо отключиться от сервера и затем снова подключиться к нему. Глобальный режим сохраняется до момента перезапуска сервера MySQL.

Теперь, когда данные внесены в таблицы, можно пользоваться базой данных для нахождения нужных сведений. Об этом и пойдет речь в следующем разделе.

Извлечение данных из таблиц

Для получения информации из таблиц базы данных используются запросы — SQL-команды, начинающиеся с ключевого слова `SELECT`. В этом разделе вы познакомитесь со структурой запросов.

Простые запросы

Знакомство с запросами мы начнем с наиболее простой команды, которая выводит все данные, содержащиеся в таблице:

```
SELECT * FROM <Имя таблицы>;
```

Например, в результате выполнения запроса:

```
SELECT * FROM Customers;
```

мы получим всю информацию о клиентах (см. рис. 3.3 выше).

**ПРИМЕЧАНИЕ**

Последующие примеры запросов не будут иллюстрироваться изображениями окон с результатом запроса, поскольку окна различных клиентских приложений при выполнении одного и того же запроса будут отображать один и тот же результат. Ограничусь тем, что приведу таблицу, содержащую выводимые запросом данные.

Вместо звездочки можно указать список столбцов таблицы, из которых нужно получить информацию. Например, чтобы вывести только имя, телефон и рейтинг каждого клиента, выполним запрос:

```
SELECT name,phone,rating FROM Customers;
```

Результатом будет следующий набор данных (табл. 3.6).

Таблица 3.6. Результат выполнения запроса

name	phone	rating
ООО "Кускус"	313-48-48	1000
Петров	7 (929) 112-14-15	1500
Крылов	444-78-90	1000

Запросом можно получать не только значения столбцов, но и значения, вычисленные с помощью выражений. Например, запрос:

```
SELECT name,phone,rating/1000 FROM CUSTOMERS;
```

возвращает результат, аналогичный предыдущему, только значения рейтинга разделены на 1000 (табл. 3.7).

Таблица 3.7. Результат выполнения запроса

name	phone	rating
ООО "Кускус"	313-48-48	1.0000
Петров	7 (929) 112-14-15	1.5000
Крылов	444-78-90	1.0000

О функциях и операторах, которые можно использовать в выражениях, будет подробно рассказано в главе 4.

С помощью запросов можно также вычислять значения без обращения к какой-либо таблице. Например, запрос:

```
SELECT 2*2;
```

возвращает следующий результат (табл. 3.8).

Таблица 3.8. Результат выполнения запроса

2*2
4

Результат запроса может содержать повторяющиеся строки. Например, клиенты могут иметь одинаковые рейтинги, поэтому запрос:

```
SELECT rating FROM Customers;
```

выдает результат, в котором есть одинаковые строки (табл. 3.9).

Таблица 3.9. Результат выполнения запроса

rating
1000
1500
1000

Чтобы исключить повторения из результата запроса, добавим в текст запроса ключевое слово `DISTINCT`. Например, чтобы просмотреть список рейтингов клиентов, не содержащий дубликатов, выполним запрос:

```
SELECT DISTINCT rating FROM Customers;
```

и получим следующий результат (табл. 3.10).

Таблица 3.10. Результат выполнения запроса

rating
1000
1500

Чтобы упорядочить строки, выведенные запросом, по значениям одного из столбцов, добавим в текст запроса выражение:

```
ORDER BY <Имя столбца> [ASC или DESC]
```

Ключевое слово `ASC` означает, что сортировка выполняется по возрастанию значений, `DESC` — что требуется сортировка по убыванию. Если ни то, ни другое слово не указано, выполняется сортировка по возрастанию. Кроме того, для сортировки можно использовать сразу несколько столбцов, тогда строки будут отсортированы по значениям первого из столбцов, строки с одинаковым значением в первом столбце будут отсортированы по значениям второго из столбцов и т. д. Например, запрос:

```
SELECT name,phone,rating FROM Customers  
ORDER BY rating DESC, name;
```

возвращает следующий результат (табл. 3.11).

Таблица 3.11. Результат выполнения запроса

name	phone	rating
Петров	7 (929) 112-14-15	1500
Крылов	444-78-90	1000
ООО "Кускус"	313-48-48	1000

Как видим, в результате запроса сначала приводится строка с наибольшим рейтингом, а строки с одинаковым рейтингом располагаются в алфавитном порядке имен клиентов.

Вместо имен столбцов в выражении `ORDER BY` можно использовать их порядковые номера в результате запроса. Например, приведенный выше запрос аналогичен запросу:

```
SELECT name,phone,rating FROM Customers  
ORDER BY 3 DESC, 1;
```

Запросы, которые рассматривались в этом подразделе, выводили информацию из всех имеющихся в таблице строк. Из следующего подраздела вы узнаете, как использовать запросы для отбора строк таблицы.

Условия отбора

Чтобы выбрать из таблицы строки, удовлетворяющие какому-либо критерию, добавим в текст запроса выражение:

```
WHERE <Условие отбора>
```

Например, запрос:

```
SELECT name,phone,rating FROM Customers WHERE rating=1000;
```

возвращает только те строки, в которых значение рейтинга равно 1000 (табл. 3.12).

Таблица 3.12. Результат выполнения запроса

name	phone	rating
ООО "Кускус"	313-48-48	1000
Крылов	444-78-90	1000

В условии отбора можно использовать любые операторы и функции языка SQL (о них мы подробно поговорим в главе 4), в том числе логические операторы `AND` и `OR` для создания составных условий отбора. Например, запрос:

```
SELECT name,phone,rating FROM Customers
```

```
WHERE name LIKE '000%' OR rating>1000  
ORDER BY rating DESC;
```

выводит информацию о тех клиентах, чье имя начинается с «ООО», и о тех, чей рейтинг превосходит 1000. При этом строки упорядочиваются в порядке убывания значения рейтинга (табл. 3.13).

Таблица 3.13. Результат выполнения запроса

name	phone	rating
Петров	7 (929) 112-14-15	1500
ООО "Кускус"	313-48-48	1000

Пока мы рассматривали запросы, получающие данные только из одной таблицы. В следующем разделе мы поговорим о запросах, позволяющих выводить информацию сразу из нескольких таблиц.

Объединение таблиц

Получить информацию из нескольких таблиц можно, указав в запросе список столбцов и список таблиц, из которых требуется получить информацию:

```
SELECT <Список столбцов> FROM <Список таблиц>  
WHERE <Условие отбора>;
```

Например, если нужно вывести информацию обо всех заказанных товарах за определенную дату с указанием имен и адресов заказчиков, выполним команду:

```
SELECT name,address,product_id,qty  
FROM Customers, Orders  
WHERE Customers.id=customer_id AND date='2007-12-12';
```

Эта команда выводит следующий результат (табл. 3.14).

Таблица 3.14. Результат выполнения запроса

name	address	product_id	qty
ООО "Кускус"	ул. Смольная, д. 7	5	8
Крылов	Зеленый пр-т, д. 22	2	14

С помощью этого запроса мы получили данные из столбцов name и address таблицы Customers и столбцов product_id и qty таблицы Orders. Указав условие WHERE Customers.id=customer_id, мы сообщили программе MySQL, что для каждого клиента должны выводиться сведения только о его заказах. Если бы это условие не было задано, мы получили бы бессмысленный набор всевозможных комбинаций данных из таблицы Customers с данными из таблицы Orders. Об-

ратите внимание, что столбец с именем `id` есть и в таблице `Customers`, и в таблице `Orders`, поэтому мы добавили имя таблицы `Customers` в виде префикса к имени столбца.

По такому же принципу допускается объединять в запросе и более двух таблиц, и даже таблицу с самой собой. Объединение таблицы с собой можно представить себе как объединение нескольких идентичных таблиц. Чтобы было легко различать эти таблицы, им присваиваются разные псевдонимы. В качестве примера объединения таблицы с самой собой рассмотрим запрос, который выводит всевозможные пары клиентов с одинаковым рейтингом:

```
SELECT L.name,R.name FROM Customers L, Customers R
WHERE L.rating=R.rating;
```

Создавая этот запрос, мы присвоили «первому экземпляру» таблицы `Customers` псевдоним `L`, «второму экземпляру» — псевдоним `R`. В результате объединения «таблиц» мы получили всевозможные пары клиентов: первый клиент в каждой паре — это строка из «таблицы» `L`, второй — строка из «таблицы» `R`. Благодаря условию `WHERE L.rating=R.rating` мы выбрали те пары, в которых рейтинг клиента из таблицы `L` (`L.rating`) равен рейтингу клиента из таблицы `R` (`R.rating`). Как и в предыдущем примере, к именам столбцов мы добавили в виде префикса имена «таблиц» (в данном случае — псевдонимы), чтобы указать, к какому из экземпляров таблицы относится каждый из столбцов.

Таким образом, запрос выводит следующие пары имен (табл. 3.15).

Таблица 3.15. Результат выполнения запроса

name	name
ООО "Кускус"	ООО "Кускус"
Крылов	ООО "Кускус"
Петров	Петров
ООО "Кускус"	Крылов
Крылов	Крылов

Поскольку наборы строк в «таблицах» `L` и `R` одинаковые, в результате запроса появилось много лишних данных: пары одинаковых имен (они возникли при сравнении строки «таблицы» `L` с точной копией этой строки в «таблице» `R`), а также одна и та же пара имен сначала в прямом, затем в обратном порядке. Чтобы избавиться от повторений, введем дополнительное условие отбора:

```
SELECT L.name,R.name FROM Customers L, Customers R
WHERE L.rating=R.rating AND L.name<R.name;
```

Поскольку в действительности одинаковый рейтинг имеют только клиенты Крылов и ООО «Кускус», результатом этого запроса станет единственная строка (табл. 3.16).

Таблица 3.16. Результат выполнения запроса

name	name
Крылов	ООО "Кускус"

Запросы, объединяющие таблицу с самой собой, можно использовать, в частности, для поиска ошибок дублирования данных в таблице, например для поиска клиентов с разными идентификаторами, но одинаковыми именами, адресами и телефонами.

Далее мы рассмотрим возможности комбинирования запросов.

Вложенные запросы

Поскольку результатом запроса является массив данных в виде таблицы, вы можете использовать результат одного запроса в другом. Во многих случаях вложенными запросами можно заменить объединение таблиц. Например, получить список имен клиентов, когда-либо заказывавших товар № 5, можно с помощью вложенного запроса:

```
SELECT name FROM Customers
WHERE id IN
(SELECT DISTINCT customer_id FROM Orders
WHERE product_id=5);
```

Здесь вложенный запрос получает из таблицы `Orders` номера клиентов, заказавших товар № 5. Для обработки результатов подзапроса мы применили оператор `IN`, который возвращает истинное значение (`true`), если элемент слева от оператора совпадает с одним из элементов списка справа от оператора. В данном случае оператор `IN` проверяет, содержится ли номер клиента (значение столбца `id`) в списке номеров, выданных под запросом. Таким образом, внешний запрос выводит имена тех клиентов, номера которых получены в результате подзапроса (табл. 3.17).

Таблица 3.17. Результат выполнения запроса

name
ООО "Кускус"

Такой же результат можно получить и с использованием объединения таблиц:

```
SELECT DISTINCT name FROM Customers, Orders
WHERE Customers.id=customer_id AND product_id=5;
```

Однако не всегда вложенные запросы и объединения таблиц взаимозаменяемы. В частности, запросы с объединениями могут выводить данные из всех участвующих в запросе таблиц, а запросы с вложенными запросами, — только из таблиц, участвующих во внешнем запросе. А с помощью запросов, использующих групповые (агрегатные) функции в подзапросах, можно получить результат, недостижимый другими способами. Например, вывести заказ с наибольшей суммой можно только благодаря вложенному запросу, подсчитывающему максимальную сумму заказа:

```
SELECT * FROM Orders
WHERE amount=(SELECT MAX(amount) FROM Orders);
```

Во вложенном запросе групповая функция MAX возвращает наибольшее из значений столбца amount (сумма) таблицы Orders — в данном случае 22000. Внешний запрос, в свою очередь, выводит те строки таблицы Orders, в которых значение столбца amount равно значению, выданному подзапросом, то есть 22000 (табл. 3.18).

Таблица 3.18. Результат выполнения запроса

id	date	product_id	qty	amount	customer_id
1013	12.12.2007	2	14	22000	536

О других групповых функциях, а также об операторах, используемых для обработки результатов подзапроса, мы поговорим в главе 4.

Надо отметить, что вы можете включить в запрос одновременно и подзапросы, и объединения таблиц и тем самым получить еще более мощные возможности для поиска и отбора данных.

В следующем подразделе мы рассмотрим еще один способ совместного использования запросов — объединение запросов.

Объединение результатов запросов

Чтобы объединить несколько запросов в одну SQL-команду и, соответственно, объединить результаты запросов, используется ключевое слово UNION. Запросы,

объединяемые с помощью `UNION`, должны выводить одинаковое количество столбцов, и типы данных столбцов должны быть совместимы. При объединении результатов автоматически удаляются повторяющиеся строки; чтобы запретить удаление повторяющихся строк, вместо слова `UNION` нужно использовать выражение `UNION ALL`. Наконец, строки объединенного запроса можно упорядочить выражением `ORDER BY`. В качестве примера рассмотрим запрос, выводящий информацию о заказах с наибольшей и наименьшей суммой заказа:

```
SELECT * FROM Orders
WHERE amount=(SELECT MAX(amount) FROM Orders)
UNION
SELECT * FROM Orders
WHERE amount=(SELECT MIN(amount) FROM Orders)
ORDER BY 1;
```

Результатом выполнения этого запроса будет следующий набор данных (табл. 3.19).

Таблица 3.19. Результат выполнения запроса

id	date	product_id	qty	amount	customer_id
1012	12.12.2007	5	8	4500	533
1013	12.12.2007	2	14	22000	536

Первый запрос возвращает строку таблицы `Orders`, в которой значение поля `amount` максимально (это строка с `id = 1013`), второй — строку, в которой значение поля `amount` минимально (это строка с `id = 1012`), и при упорядочении по значению столбца `id` строки меняются местами.

Итак, мы рассмотрели основные возможности поиска и отбора данных, предоставляемые командой `SELECT`. Далее мы поговорим о том, как выгрузить результат запроса в файл.

Выгрузка данных в файл

Чтобы результат запроса был сохранен в файл, добавим в команду `SELECT` выражение:

```
INTO OUTFILE 'Путь и имя файла' [FIELDS ...] [LINES ...]
```

В этой команде нужно указать полный путь к файлу, в который будут выгружены данные (этот файл должен быть новым, не существующим на момент выгрузки). При задании пути к файлу необходимо использовать прямую косую черту вместо принятой в Windows обратной косой черты. Указанный файл создается на компьютере, где работает сервер MySQL. Данные выгружаются в той кодировке, в которой они хранятся в базе данных.

При необходимости вы можете также задать параметры `FILEDS` и `LINES`, которые имеют то же назначение, что и параметры `FILEDS` и `LINES` команды `LOAD DATA`. Если впоследствии файл будет загружаться в базу данных MySQL с помощью команды `LOAD DATA`, то в команде `LOAD DATA` нужно будет указать те же самые значения параметров `FILEDS` и `LINES`, которые использовались при выгрузке.

Команды `SELECT ... INTO OUTFILE` и `LOAD DATA` можно использовать для резервного копирования таблиц или для переноса данных на другой сервер MySQL. Например, данные из таблицы `Customers`, сохраненные в файл с помощью команды:

```
SELECT * from Customers INTO OUTFILE 'C:/data/Customers.txt';
```

можно загрузить в таблицу `Customers_copy` (имеющую такую же структуру, что и таблица `Customers`) командой:

```
LOAD DATA INFILE 'C:/data/Customers.txt'  
  INTO TABLE Customers_copy;
```

ВНИМАНИЕ



При выгрузке и последующей загрузке символьных данных в кодировке UTF-8 могут возникнуть проблемы, связанные с переменным количеством байтов на символ в этой кодировке. Если вы выгрузили данные из таблицы с кодировкой UTF-8, рекомендуется перед загрузкой преобразовать файл, определив для него какую-либо однобайтовую кодировку. Например, откройте файл с помощью программы Блокнот (Пуск ► Все программы ► Стандартные ► Блокнот), в меню Файл выберите команду Сохранить как, а затем в стандартном окне Windows Сохранить как в поле Кодировка выберите из списка значение ANSI и нажмите кнопку Сохранить. При загрузке преобразованного файла укажите в команде `LOAD DATA` параметр `CHARACTER SET cp1251` (см. обсуждение выше).

Итак, мы рассмотрели работу команды `SELECT`, которая предоставляет широкие возможности поиска и отбора данных. Последняя операция, которую мы обсудим в этой главе, — редактирование данных в таблицах.

Изменение данных

В этом разделе мы поговорим о командах изменения, замещения и удаления строк таблицы. Начнем с рассмотрения команды `UPDATE`, которая позволяет установить новые значения в одной или нескольких строках:

```
UPDATE <Имя таблицы>
SET <Имя столбца 1>=<Значение 1>,
    ...,
    <Имя столбца N>=<Значение N>
[WHERE <Условие отбора>]
[ORDER BY <Имя столбца> [ASC или DESC]]
[LIMIT <Количество строк>];
```

Например, если у клиента по фамилии Крылов изменился номер телефона, то обновить информацию в базе данных можно с помощью команды:

```
UPDATE Customers SET phone='444-25-27' WHERE id=536;
```

В команде `UPDATE` используются следующие основные параметры.

- ❑ **Имя редактируемой таблицы.**
- ❑ `SET <Имя столбца 1>=<Значение 1>, ..., <Имя столбца N>=<Значение N>` — укажите список столбцов и новых значений для этих столбцов. Более подробная информация о вставке значений в таблицу и о режимах взаимодействия с сервером MySQL содержится в соответствующем подразделе выше.

Задать новое значение вы также можете с помощью выражения, использующего прежние значения в строке. Например, удвоить рейтинги для всех клиентов можно командой:

```
UPDATE Customers SET rating=rating*2;
```

- ❑ `WHERE <Условие отбора>` — укажите условие отбора, чтобы изменения были применены только к тем строкам таблицы, которые удовлетворяют этому условию. Если условие отбора не задано, то изменения будут применены ко всем строкам таблицы. Условия отбора мы обсуждали в подразделе выше. В них допустимо использовать вложенный запрос, однако он не должен обращаться к самой модифицируемой таблице.
- ❑ `ORDER BY <Имя столбца> [ASC или DESC]` — при необходимости вы можете указать, в каком порядке следует применять изменения к строкам таблицы. Обычно порядок применения изменений не влияет на результат выполнения

операции. Однако в некоторых случаях последовательность действий может быть важна. Например, если вы установили предельное количество изменяемых строк (см. следующий пункт), то некоторые строки, удовлетворяющие условию отбора, могут остаться неизменными, а какие именно это будут строки — зависит от последовательности применения изменений. Другой подобный случай — обновление значений первичного ключа или уникального индекса, которые не должны содержать повторяющихся значений, а наличие или отсутствие повторяющихся значений в столбце может зависеть от порядка применения изменений.

- ❑ `LIMIT <Количество строк>` — при необходимости вы можете указать максимальное количество строк таблицы, которые могут быть изменены командой `UPDATE`. Если это количество измененных строк достигнуто, то операция завершается, даже если в таблице еще остались строки, которые удовлетворяют условию отбора и не были изменены.



ПРИМЕЧАНИЕ

При обновлении значения первичного ключа в родительской таблице выполняются проверки целостности (описание параметров внешнего ключа приведено выше).

Следующая команда, которую мы рассмотрим, называется `REPLACE`. Она либо добавляет, либо замещает строки таблицы. Эта команда имеет те же параметры, что и команда `INSERT` (см. подраздел выше):

```
REPLACE [INTO] <Имя таблицы>
[(<Список столбцов>)]
VALUES
  (<Список значений 1>),
  (<Список значений 2>),
  ...
  (<Список значений N>);
```

Если в строке, вставляемой в таблицу с помощью команды `REPLACE`, значение первичного ключа или уникального индекса не совпадает ни с одним из уже существующих значений, то эта команда работает так же, как `INSERT` (если в таблице нет ни первичного ключа, ни уникального индекса, то команда `REPLACE` всегда работает как `INSERT`). Если же в таблице есть строка с таким же значением первичного ключа или уникального индекса, то перед добавлением новой строки прежняя строка удаляется.

Например, выполнив команду:

```
REPLACE INTO Products
```

VALUES

```
(3, 'Соковыжималка Мосбытприбор СШ-800',  
  'Цвет: кремовый. Мощность: 350 Вт', 1299.99);
```

мы тем самым заменим в таблице `Products` прежнюю строку с идентификатором 3, содержащую информацию о кофеварке (см. листинг 3.5 выше), новой строкой, также имеющей идентификатор 3, но содержащей информацию о соковыжималке.

Надо отметить, что в команде `REPLACE`, в отличие от `UPDATE`, нельзя задавать новые значения с помощью выражений, вычисляемых с использованием прежних значений, хранившихся в строке. Если вы укажете в команде `REPLACE` такое выражение, то вместо прежнего значения будет подставлено значение данного столбца по умолчанию.

И наконец, последняя команда, которую мы рассмотрим, предназначена для удаления строк таблицы:

```
DELETE FROM <Имя таблицы>  
[WHERE <Условие отбора>]  
[ORDER BY <Имя столбца> [ASC или DESC]]  
[LIMIT <Количество строк>];
```

Например, информацию о клиенте по фамилии Петров вы можете удалить из таблицы `Customers` с помощью команды:

```
DELETE FROM Customers WHERE id=534;
```

Параметры команды `DELETE` аналогичны соответствующим параметрам команды `UPDATE`. В результате выполнения этой команды будут удалены строки таблицы, удовлетворяющие условию отбора, а если условие отбора не задано — все строки таблицы. При этом с помощью параметра `LIMIT` можно указать предельное количество удаляемых строк, а с помощью параметра `ORDER BY` — последовательность удаления строк.

ПРИМЕЧАНИЕ



При удалении строк родительской таблицы выполняются проверки целостности связи (см. описание параметров внешнего ключа в подразделе выше).

Итак, мы освоили операции изменения и удаления строк таблицы, а именно команды `UPDATE` (обновление), `REPLACE` (замещение) и `DELETE` (удаление). Подведем теперь итоги.

3.5. Резюме

Из этой главы вы узнали все необходимое для построения собственной базы данных. Вы научились создавать, изменять и удалять базы данных и таблицы, настраивать ключевые столбцы и индексы, а также познакомились с типами данных, используемыми в MySQL. Вы освоили операции работы с данными, а именно добавление строк в таблицу, изменение и удаление строк таблицы. Кроме того, вы научились находить в базе данных нужную вам информацию с помощью запросов.

В следующей главе мы продолжим обсуждение средств языка SQL на более продвинутом уровне.

Глава 4

Операторы и функции языка SQL

Из этой главы вы узнаете о функциях и операторах, с помощью которых сможете создавать выражения — формулы, вычисляющие какое-либо значение (числовое, логическое, символьное и др.). Наиболее часто выражения используются в SQL-запросах: как для вычисления значений, выводимых запросом, так и в условиях отбора. Кроме того, с помощью выражений можно задавать условия отбора в SQL-командах `UPDATE` и `DELETE`, определять значения, добавляемые в таблицу, в командах `INSERT` и `UPDATE` и многое другое.

Отличие операторов от функций заключается, по существу, только в форме записи. Аргументы функции записываются после имени функции в скобках через запятую, в то время как аргументы оператора (операнды) могут располагаться по обе стороны от значка или имени оператора. Поэтому, рассматривая операторы и функции, мы будем подразделять их на группы, руководствуясь их назначением, а не внешними различиями.

В первую очередь рассмотрим наиболее часто используемую группу операторов — операторы, осуществляющие проверку какого-либо условия.

4.1. Операторы и функции проверки условий

В этом разделе мы поговорим об операторах, которые предназначены для создания условий отбора, а именно об операторах, сравнивающих две или несколько

величин, и о логических операторах, позволяющих создавать комбинированные условия.

Кроме того, мы обсудим функции и операторы, возвращающие один из своих аргументов, выбранный согласно некоторому критерию.

Операторы сравнения

С помощью операторов, о которых пойдет речь в этом подразделе, можно сравнивать между собой значения столбцов таблиц, значения выражений и константы, относящиеся к любым типам данных. Результатом сравнения является логическое значение:

- ❑ 1 (true) — истинное значение, свидетельствует о том, что сравнение верно, условие выполнено;
- ❑ 0 (false) — ложное значение, свидетельствует о том, что сравнение неверно, условие не выполнено;
- ❑ NULL — неопределенное значение, свидетельствует о том, что проверить условие невозможно, поскольку один из операндов равен NULL.



ПРИМЕЧАНИЕ

Иногда проверить условие можно, несмотря на то что один из операндов равен NULL (см., например, описание операторов BETWEEN и IN в этом подразделе); в этом случае возвращается значение 1 или 0.

Начнем с рассмотрения оператора, проверяющего равенство двух операндов.

x = y

Оператор «равно» возвращает следующие значения:

- ❑ 1 (true), если *x* и *y* совпадают;
- ❑ 0 (false), если *x* и *y* различны;
- ❑ NULL, если по крайней мере один из операндов равен NULL.

Например, выберем из таблицы Customers строки, в которых значение в столбце name равно "КРЫЛОВ":

```
SELECT * FROM Customers WHERE name='КРЫЛОВ';
```

Результат этого запроса представлен в табл. 4.1.

Таблица 4.1. Результат выполнения запроса

Id	name	phone	address	rating
536	Крылов	444-78-90	Зеленый пр-т, д. 22	1000

Как видите, при сравнении строк с помощью этого оператора регистр символов не учитывается.

Следующий оператор также проверяет равенство двух операндов.

$x \leq y$

Если оба операнда не равны NULL, данный оператор аналогичен оператору «равно». Если один из операндов равен NULL, оператор \leq возвращает значение 0 (*false*), а если оба операнда равны NULL — значение 1 (*true*). Например, запрос:

```
SELECT 100=NULL, 100<=NULL, NULL=NULL, NULL<=NULL;
```

возвращает результат, представленный в табл. 4.2 и наглядно иллюстрирующий различие между операторами = и \leq .

Таблица 4.2. Результат выполнения запроса

100=NULL	100<=NULL	NULL=NULL	NULL<=NULL
NULL	0	NULL	1

Следующие операторы проверяют равенство операнда какому-либо логическому значению.

$x \text{ IS } y$, где y — *true*, *false*, *UNKNOWN* или *NULL*

Выражение $x \text{ IS } \textit{true}$ возвращает 1 (*true*), если x — отличное от нуля число или отличные от нулевых (0000-00-00 00:00:00) дата и/или время, и 0 (*false*) в остальных случаях.

Выражение $x \text{ IS } \textit{false}$ возвращает 1 (*true*), если x равен нулю либо нулевым дате и/или времени, и 0 (*false*) в остальных случаях.



ПРИМЕЧАНИЕ

Если x является символьной строкой, то перед сравнением с *true* или *false* эта строка преобразуется в число. Для этого отбрасываются все символы, начиная с первого символа, недопустимого в числовом значении, а начальная подстрока рассматривается как число. Если первый символ в значении — буква или строка пустая (""), то x приравнивается к нулю.

Выражения `x IS UNKNOWN` и `x IS NULL` возвращают 1 (`true`), если `x` равен `NULL`, и 0 (`false`) в остальных случаях.

Например, запрос:

```
SELECT 100 IS true, 0 IS true, '2007-12-12' IS true,  
       '0000-00-00' IS true, NULL IS true;
```

возвращает результат, представленный в табл. 4.3.

Таблица 4.3. Результат выполнения запроса

100 IS true	0 IS true	'2007-12-12' IS true	'0000-00-00' IS true	NULL IS true
1	0	1	0	0

Запрос:

```
SELECT 100 IS false, 0 IS false, '2007-12-12' IS false,  
       '0000-00-00' IS false, NULL IS false;
```

возвращает результат, представленный в табл. 4.4.

Таблица 4.4. Результат выполнения запроса

100 IS false	0 IS false	'2007-12-12' IS false	'0000-00-00' IS false	NULL IS false
0	1	0	1	0

Запрос:

```
SELECT 100 IS NULL, 0 IS NULL, '2007-12-12' IS NULL,  
       '0000-00-00' IS NULL, NULL IS NULL;
```

возвращает результат, представленный в табл. 4.5.

Таблица 4.5. Результат выполнения запроса

100 IS NULL	0 IS NULL	'2007-12-12' IS NULL	'0000-00-00' IS NULL	NULL IS NULL
0	0	0	0	1



ПРИМЕЧАНИЕ

Если столбец определен как `DATE NOT NULL` (или `DATETIME NOT NULL`), то значение этого столбца, равное `0000-00-00` (или `0000-00-00 00:00:00`), рассматривается оператором `IS NULL` как `NULL`. Например, если при создании таблицы `Orders`

(см. листинг 3.4 в главе 3) задать для столбца `date` свойство `NOT NULL`, то запрос `SELECT * FROM Orders WHERE date IS NULL`; выведет строки, в которых дата заказа равна 0000-00-00.

Следующие операторы проверяют несовпадение двух операндов.

`x != y, x <> y`

Оператор «не равно» возвращает следующие значения:

- ❑ 1 (`true`), если x и y различны;
- ❑ 0 (`false`), если x и y совпадают;
- ❑ `NULL`, если по крайней мере один из операндов равен `NULL`.

Например, запрос:

```
SELECT * FROM Customers WHERE name != 'КРЫЛОВ';
```

возвращает результат, обратный приведенному в табл. 4.1, то есть все строки, кроме строк с фамилией Крылов (табл. 4.6).

Таблица 4.6. Результат выполнения запроса

Id	name	Phone	address	rating
533	ООО "Кускус"	313-48-48	ул. Смольная, д. 7	1000
534	Петров	7 (929) 112-14-15	ул. Рокотова, д. 8	1500

Следующие операторы проверяют несовпадение операнда с каким-либо логическим значением.

`x IS NOT y`, где y — `true`, `false`, `UNKNOWN` или `NULL`

Выражение `x IS NOT true` возвращает 0 (`false`), если x — отличное от нуля число или отличное от нуля (0000-00-00 00:00:00) дата и/или время, и 1 (`true`) в остальных случаях.

Выражение `x IS NOT false` возвращает 0 (`false`), если x равен нулю, нулевым дата и/или времени, и 1 (`true`) в остальных случаях.

Выражения `x IS NOT UNKNOWN` и `x IS NOT NULL` возвращают 0 (`false`), если x равен `NULL`, и 1 (`true`) в остальных случаях.

Например, запрос:

```
SELECT 100 IS NOT true, 0 IS NOT true,
```

```
'2007-12-12' IS NOT true, '0000-00-00' IS NOT true,
NULL IS NOT true;
```

возвращает результат, представленный в табл. 4.7.

Таблица 4.7. Результат выполнения запроса

100 IS NOT true	0 IS NOT true	'2007-12-12' IS NOT true	'0000-00-00' IS NOT true	NULL IS NOT true
0	1	0	1	1

Запрос

```
SELECT 100 IS NOT false, 0 IS NOT false,
'2007-12-12' IS NOT false, '0000-00-00' IS NOT false,
NULL IS NOT false;
```

возвращает результат, представленный в табл. 4.8.

Таблица 4.8. Результат выполнения запроса

100 IS NOT false	0 IS NOT false	'2007-12-12' IS NOT false	'0000-00-00' IS NOT false	NULL IS NOT false
1	0	1	0	1

Запрос:

```
SELECT 100 IS NOT NULL, 0 IS NOT NULL,
'2007-12-12' IS NOT NULL, '0000-00-00' IS NOT NULL,
NULL IS NOT NULL;
```

возвращает результат, представленный в табл. 4.9.

Таблица 4.9. Результат выполнения запроса

100 IS NOT NULL	0 IS NOT NULL	'2007-12-12' IS NOT NULL	'0000-00-00' IS NOT NULL	NULL IS NOT NULL
1	1	1	1	0

Как видите, операторы x IS NOT y и x IS y возвращают противоположные результаты.

Следующий оператор проверяет, меньше ли первый операнд, чем второй.

$x < y$

Оператор «меньше» возвращает следующие значения:

- ❑ 1 (true), если x меньше y ;
- ❑ 0 (false), если x равен y или x больше y ;
- ❑ NULL, если по крайней мере один из операндов равен NULL.

Например, запрос:

```
SELECT * FROM Customers WHERE name < 'КРЫЛОВ';
```

возвращает пустой результат, поскольку "Крылов" — наименьшее в алфавитном порядке значение в столбце name таблицы Customers, предшествующих ему значений в столбце нет, а следовательно, ни одна строка не удовлетворяет условию отбора.

Следующий оператор проверяет, не превосходит ли первый операнд второй.

 $x \leq y$

Оператор «меньше либо равно» возвращает следующие значения:

- ❑ 1 (true), если x равен y или x меньше y ;
- ❑ 0 (false), если x больше y ;
- ❑ NULL, если по крайней мере один из операндов равен NULL.

Например, запрос:

```
SELECT * FROM Customers WHERE name <= 'КРЫЛОВ';
```

возвращает результат, представленный в табл. 4.1.

Следующий оператор проверяет, больше ли первый операнд, чем второй.

 $x > y$

Оператор «больше» возвращает следующие значения:

- ❑ 1 (true), если x больше y ;
- ❑ 0 (false), если x равен y или x меньше y ;
- ❑ NULL, если по крайней мере один из операндов равен NULL.

Например, запрос:

```
SELECT * FROM Customers WHERE name>'КРЫЛОВ';
```

возвращает результат, представленный в табл. 4.6.

Следующий оператор проверяет, является ли первый операнд большим либо равным по отношению ко второму.

$x \geq y$

Оператор «больше либо равно» возвращает следующие значения:

- ☐ 1 (true), если x равен y или x больше y ;
- ☐ 0 (false), если x меньше y ;
- ☐ NULL, если по крайней мере один из операндов равен NULL.

Например, запрос:

```
SELECT * FROM Customers WHERE name>='КРЫЛОВ';
```

возвращает все строки таблицы Customers (табл. 4.10).

Таблица 4.10. Результат выполнения запроса

id	name	phone	address	rating
533	ООО "Кускус"	313-48-48	ул. Смольная, д. 7	1000
534	Петров	7 (929) 112-14-15	ул. Рокотова, д. 8	1500
536	Крылов	444-78-90	Зеленый пр-т, д. 22	1000

Следующий оператор проверяет, находится ли первый операнд в промежутке между вторым и третьим.

$x \text{ BETWEEN } a \text{ AND } b$

Оператор «между» возвращает следующие значения:

- ☐ 1 (true), если $a \leq x \leq b$;
- ☐ 0 (false), если x меньше a или больше b ;
- ☐ NULL в остальных случаях.

Например, запрос:

```
SELECT * FROM Customers
```

```
WHERE name BETWEEN 'КРЫЛОВ' AND 'ООО "Кускус"';
```

возвращает следующие строки таблицы Customers (табл. 4.11).

Таблица 4.11. Результат выполнения запроса

id	name	Phone	address	rating
533	ООО "Кускус"	313-48-48	ул. Смольная, д. 7	1000
536	Крылов	444-78-90	Зеленый пр-т, д. 22	1000

Следующий оператор проверяет, находится ли первый операнд за пределами промежутка между вторым и третьим операндами.

x NOT BETWEEN a AND b

Оператор возвращает результат, противоположный результату оператора «между»:

- ☐ 0 (false), если $a \leq x \leq b$;
- ☐ 1 (true), если x меньше a или больше b ;
- ☐ NULL в остальных случаях.

Например, запрос:

```
SELECT * FROM Customers
```

```
WHERE name NOT BETWEEN 'КРЫЛОВ' AND 'ООО "Кускус"';
```

возвращает следующие строки таблицы Customers (табл. 4.12).

Таблица 4.12. Результат выполнения запроса

id	name	Phone	address	rating
534	Петров	7 (929) 112-14-15	ул. Рокотова, д. 8	1500

Следующий оператор проверяет наличие первого операнда в списке значений, который является вторым операндом.

x IN (Список значений)

Оператор «содержится в списке» возвращает такие значения:

- ☐ 1 (true), если x совпадает с одним из элементов списка;
- ☐ 0 (false), если x не совпадает ни с одним из элементов списка;
- ☐ NULL, если x равен NULL, а также в тех случаях, когда в списке присутствует значение NULL и при этом x не совпадает ни с одним из элементов списка.

Например, запрос:

```
SELECT * FROM Customers WHERE rating IN (500,1500,2500);
```

возвращает результат, представленный в табл. 4.12.

С помощью оператора `IN` можно также сравнивать составные значения, то есть значение x и элементы списка могут представлять собой наборы из нескольких величин (количество компонентов во всех наборах должно быть одинаковым).

Например, запрос:

```
SELECT * FROM Orders WHERE (date,product_id) IN  
  (('2007-12-12',1),('2007-12-12',2),  
   ('2007-12-13',1),('2007-12-13',2));
```

сравнивает каждую пару, состоящую из даты заказа (`date`) и номера товара (`customer_id`), со списком пар, и, если оба компонента в паре совпадают с соответствующими компонентами какой-либо пары из списка, то строка таблицы `Orders` будет включена в результат запроса. Таким образом, запрос отбирает заказы товаров № 1 и 2, сделанные 12 и 13 декабря 2007 г. (табл. 4.13).

Таблица 4.13. Результат выполнения запроса

id	date	product_id	Qty	amount	customer_id
1013	12.12.2007	2	14	22000	536

В отличие от функций `LEAST`, `GREATEST`, `INTERVAL` и `COALESCE`, списком значений для оператора `IN` может быть не только фиксированный перечень аргументов, но и результат подзапроса (соответствующий пример мы рассматривали в главе 3).

Следующий оператор проверяет отсутствие первого операнда в списке значений, который является вторым операндом.

x NOT IN (Список значений)

Оператор «не содержится в списке» возвращает результат, противоположный результату оператора `IN`:

- ☐ 0 (`false`), если x совпадает с одним из элементов списка;
- ☐ 1 (`true`), если x не совпадает ни с одним из элементов списка;
- ☐ `NULL`, если x равен `NULL`, а также в тех случаях, когда в списке присутствует значение `NULL` и при этом x не совпадает ни с одним из элементов списка.

Например, запрос:

```
SELECT * FROM Customers WHERE rating NOT IN (500,1500);
```

возвращает результат, представленный в табл. 4.11.

Этот оператор, как и оператор IN, может работать с составными значениями, а также со списком, полученным в результате подзапроса.

Следующий оператор проверяет соответствие первого операнда шаблону, который является вторым операндом.

x LIKE y

Оператор сравнения с шаблоном возвращает следующие значения:

- 1 (true), если *x* соответствует шаблону *y*;
- 0 (false), если *x* не соответствует шаблону *y*;
- NULL, если *x* или *y* равен NULL.

В шаблоне можно использовать два специальных подстановочных символа:

- '%' — на месте знака процента может быть любое количество произвольных символов операнда *x*;
- '_' — на месте знака подчеркивания может быть только один произвольный символ операнда *x*.

Например, следующий запрос выводит данные о тех клиентах, чьи имена содержат кавычки:

```
SELECT * FROM Customers WHERE name LIKE '%"%"';
```

Результат этого запроса представлен в табл. 4.14.

Таблица 4.14. Результат выполнения запроса

id	name	Phone	address	rating
533	ООО "Кускус"	313-48-48	ул. Смольная, д. 7	1000

Если требуется включить в шаблон знак процента или подчеркивания, которые должны рассматриваться не как подстановочные, а как обычные символы, перед ними нужно поставить обратную косую черту ('\\%', '_'). Если же шаблон должен содержать символ обратной косой черты, то ее нужно удвоить ('\\\\'). Например, значение выражения '_%' LIKE '_\\%' истинное.

По умолчанию сравнение с помощью оператора `LIKE` выполняется без учета регистра символов (то есть прописная и строчная буквы рассматриваются как одинаковые). Для сравнения с учетом регистра (чтобы прописная и строчная буквы рассматривались как разные) необходимо указать ключевое слово `BINARY` или правило сравнения (`COLLATE`). Например, выражение `'Крылов' LIKE 'крылов'` истинно, а выражения `'Крылов' LIKE BINARY 'крылов'` и `'Крылов' LIKE 'крылов' COLLATE utf8_bin` ложны (правило сравнения должно соответствовать кодировке, в которой работает ваше клиентское приложение; о правилах сравнения рассказывалось в главе 3).



ПРИМЕЧАНИЕ

Более сложные шаблоны вы можете создавать с помощью регулярных выражений. Такие выражения представляют собой универсальный язык описания текстов. Для сравнения строки с шаблоном, содержащим регулярные выражения, необходимо вместо оператора `LIKE` использовать оператор `REGEXP`.

Следующий оператор проверяет несоответствие первого операнда шаблону, который является вторым операндом.

x NOT LIKE y

Оператор `NOT LIKE` возвращает результат, противоположный результату выполнения оператора `LIKE`:

- ❑ 0 (`false`), если *x* соответствует шаблону *y*;
- ❑ 1 (`true`), если *x* не соответствует шаблону;
- ❑ `NULL`, если *x* или *y* равен `NULL`.

Например, следующий запрос выводит данные о тех клиентах, чьи имена не содержат кавычек:

```
SELECT * FROM Customers WHERE name NOT LIKE '%\"';
```

Результат этого запроса представлен в табл. 4.15.

Таблица 4.15. Результат выполнения запроса

id	Name	phone	address	rating
534	Петров	7 (929) 112-14-15	ул. Рокотова, д. 8	1500
536	Крылов	444-78-90	Зеленый пр-т, д. 22	1000

К операторам сравнения близка функция `STRCMP()`, которую мы также рассмотрим в этом подразделе, несмотря на то что она может возвращать, помимо значений 1 (`true`), 0 (`false`) и `NULL`, значение -1 (`true`).

STRCMP(x,y)

Функция STRCMP () сравнивает строки *x* и *y* в соответствии с текущими правилами сравнения и возвращает:

- ❑ -1, если *x* предшествует *y* в алфавитном порядке;
- ❑ 0, если *x* и *y* совпадают;
- ❑ 1, если *x* следует после *y* в алфавитном порядке;
- ❑ NULL, если по крайней мере один из аргументов равен NULL.

Например, зададим для таблицы Customers правило сравнения, не учитывающее регистр:

```
ALTER TABLE Customers
  CONVERT TO CHARACTER SET cp1251 COLLATE cp1251_general_ci;
```

В этом случае запрос:

```
SELECT name, STRCMP(name,'крылов') FROM Customers;
```

возвращает результат, представленный в табл. 4.16.

Таблица 4.16. Результат выполнения запроса

name	STRCMP(name,'крылов')
ООО "Кускус"	1
Петров	1
Крылов	0

Если же для таблицы Customers задать правило сравнения, учитывающие регистр, тот же самый запрос вернет уже другой результат (табл. 4.17).

```
ALTER TABLE Customers
  CONVERT TO CHARACTER SET cp1251 COLLATE cp1251_general_cs;
SELECT name, STRCMP(name,'крылов') FROM Customers;
```

Таблица 4.17. Результат выполнения запроса

name	STRCMP(name,'крылов')
ООО "Кускус"	1
Петров	1
Крылов	-1

Различие результатов объясняется тем, что без учета регистра строки "Крылов" и "крылов" эквивалентны, а с учетом регистра — различны.

При использовании сравнения по числовым кодам символов мы получим третий результат, отличающийся от первых двух (табл. 4.18):

```
ALTER TABLE Customers
  CONVERT TO CHARACTER SET cp1251 COLLATE cp1251_bin;
SELECT name, STRCMP(name, 'крылов') FROM Customers;
```

Таблица 4.18. Результат выполнения запроса

name	STRCMP(name, 'крылов')
ООО "Кускус"	-1
Петров	-1
Крылов	-1

Наконец, рассмотрим оператор полнотекстового поиска.

MATCH (Список столбцов) AGAINST (Критерий поиска)

Оператор `MATCH ... AGAINST ...` выполняет поиск по заданным ключевым словам в значениях указанных столбцов. При этом для столбцов должен быть создан полнотекстовый индекс (о полнотекстовых индексах мы говорили в главе 3). Для каждой строки таблицы оператор `MATCH ... AGAINST ...` возвращает величину *релевантности*, которая характеризует степень соответствия строки критерию поиска. Если оператор используется в параметре `WHERE` команды `SELECT`, то результатом запроса будут строки с отличной от нуля релевантностью, упорядоченные по убыванию релевантности (подобно результату поиска в Интернете с помощью поисковых систем).

Например, создадим полнотекстовый индекс для столбца `description` таблицы `Products`. Полнотекстовый индекс можно создать только для таблиц с типом `MyISAM`, который не поддерживает связи между таблицами. Поэтому сначала удалим связь между таблицами `Products` и `Orders`, удалив внешний ключ из таблицы `Orders`:

```
ALTER TABLE Orders DROP FOREIGN KEY orders_ibfk_1;
```

Затем изменим тип таблицы `Products` на `MyISAM`:

```
ALTER TABLE Products ENGINE MyISAM;
```

И наконец, создадим полнотекстовый индекс для столбца `description`:

```
ALTER TABLE Products ADD FULLTEXT (description);
```

После этого можно выполнять полнотекстовый поиск по столбцу `description`. Например, запрос:

```
SELECT * FROM Products
WHERE MATCH (description) AGAINST ('Чайник Мосбытприбор');
```

возвращает единственную строку (табл. 4.19).

Таблица 4.19. Результат выполнения запроса

id	Description	Details	price
4	Чайник Мосбытприбор МН	Цвет: белый. Мощность: 2200 Вт. Объем: 2 л	925.00

В других наименованиях товаров также присутствует ключевое слово «Мосбытприбор», однако сервер MySQL игнорирует те слова из критерия поиска, которые встречаются более чем в половине строк. Игнорируются также слишком короткие (из трех и менее символов) и общепотребительные слова (список этих слов — стоп-лист — приведен на веб-странице <http://dev.mysql.com/doc/refman/5.0/en/fulltext-stopwords.html>).

Если необходимо выполнить поиск по словам, которые могут встречаться более чем в 50 % строк, то необходимо использовать поиск в логическом режиме. Для поиска в логическом режиме следует включить в выражение `MATCH ... AGAINST ...` параметр `IN BOOLEAN MODE`. Управлять поиском в логическом режиме можно с помощью следующих спецсимволов:

- ☐ `+` перед словом означает, что будут найдены только строки, содержащие это слово;
- ☐ `-` перед словом означает, что будут найдены только строки, не содержащие это слово;
- ☐ `<` перед словом уменьшает «вес» этого слова при вычислении релевантности;
- ☐ `>` перед словом увеличивает «вес» этого слова при вычислении релевантности;
- ☐ `~` перед словом делает «вес» слова отрицательным (уменьшающим релевантность);
- ☐ `*` после слова означает произвольное окончание; например, запрос по слову `+чай*` выведет строки, содержащие слова «чайник», «чайница», «чайка» и т. п.;
- ☐ `"` — сочетание слов, заключенное в двойные кавычки, рассматривается как единое слово;
- ☐ `(и)` — круглые скобки позволяют создавать вложенные выражения.

Например, запрос:

```
SELECT * FROM Products
WHERE MATCH (description)
AGAINST ('-Чайник +Мосбытприбор' IN BOOLEAN MODE);
```

возвращает строки, содержащие слово «Мосбытприбор», но не содержащие слово «Чайник» (табл. 4.20).

Таблица 4.20. Результат выполнения запроса

id	description	details	price
1	Обогреватель Мосбытприбор ВГД 121R	Инфракрасный обогреватель. 3 режима нагрева: 400 Вт, 800 Вт, 1200 Вт	1145
2	Гриль Мосбытприбор СТ-14	Мощность: 1440 Вт. Быстрый нагрев. Термостат. Цветовой индикатор работы	2115
3	Кофеварка Мосбытприбор ЕКЛ-1032	Цвет: черный. Мощность: 450 Вт. Вместительность: 2 чашки	710
5	Утюг Мосбытприбор с паром АБ 200	Цвет: фиолетовый. Мощность: 1400 Вт	518

Результат полнотекстового поиска в логическом режиме не упорядочивается.

Еще один режим полнотекстового поиска — расширенный режим. Он отличается от обычного тем, что в результат запроса, помимо строк, отвечающих заданному критерию поиска, включаются строки, найденные по принципу схожести с несколькими первыми строками, наиболее релевантными исходному критерию. Расширенный режим полезен при поиске «наугад», когда заранее не ясно, по какому критерию искать нужную строку. Для поиска в логическом режиме необходимо включить в выражение `MATCH ... AGAINST ...` параметр `WITH QUERY EXPANSION`:

```
SELECT * FROM Products
WHERE MATCH (description)
AGAINST ('Чайник Мосбытприбор' WITH QUERY EXPANSION);
```

Итак, мы рассмотрели основные операторы сравнения, на которых базируются условия отбора в запросах и командах изменения и удаления строк. В следующем подразделе мы рассмотрим группу операторов и ключевых слов, которые также используются для сравнения, только одним из операндов служит результат вложенного запроса.

Операторы сравнения с результатами вложенного запроса

В этом разделе мы поговорим об операторах и ключевых словах, используемых для обработки результатов вложенного запроса. Перечислю их.

EXISTS

Оператор EXISTS возвращает значение 1 (true), если результат подзапроса содержит хотя бы одну строку, и значение 0 (false), если подзапрос выдает пустой результат.

Например, получить список товаров, заказанных по крайней мере одним клиентом, можно с помощью запроса:

```
SELECT * FROM Products
WHERE EXISTS
(SELECT * FROM Orders
WHERE product_id=Products.id
AND customer_id IS NOT NULL);
```

Обратите внимание, что в этом примере мы столкнулись с новой разновидностью вложенного запроса. В примерах, которые мы рассматривали ранее (в главе 3), вложенный запрос не использовал данные из внешнего запроса и поэтому выполнялся только один раз, после чего найденные вложенным запросом данные обрабатывались внешним запросом. Однако в текущем примере вложенный запрос *связан* с внешним: в нем используется значение столбца id таблицы Products — таблицы, которая участвует во внешнем запросе. Следовательно, вложенный запрос *выполняется отдельно для каждой строки* таблицы Products, каждый раз с новым значением столбца id.

Таким образом, для каждого товара запускается вложенный запрос, который выбирает заказы с этим товаром, сделанные каким-либо клиентом (то есть в столбце customer_id таблицы Orders должно быть значение, отличное от NULL). Если этот вложенный запрос выдал хотя бы одну строку (то есть заказ с такими параметрами существует), то условие отбора во внешнем запросе выполняется и текущая запись о товаре включается в результат, выводимый внешним запросом. В итоге мы получаем следующий список товаров (табл. 4.21).

Таблица 4.21. Результат выполнения запроса

id	description	details	price
2	Гриль Мосбытприбор СТ-14	Мощность: 1440 Вт. Быстрый нагрев. Термостат. Цветовой индикатор работы	2115.00
5	Утюг Мосбытприбор с паром АБ 200	Цвет: фиолетовый. Мощность: 1400 Вт	518.00

Далее мы рассмотрим оператор NOT EXISTS.

NOT EXISTS

Оператор возвращает результат, противоположный результату выполнения оператора `EXISTS`: значение 1 (`true`), если итог подзапроса не содержит ни одной строки, и значение 0 (`false`), если итог подзапроса непустой.

Например, получить список клиентов, заказавших все виды товаров, можно с помощью следующего запроса:

```
SELECT * FROM Customers WHERE NOT EXISTS
  (SELECT * FROM Products WHERE NOT EXISTS
    (SELECT * FROM Orders
     WHERE product_id=Products.id
     AND customer_id=Customers.id));
```

В этом запросе для каждого клиента и каждого товара самый «глубоко вложенный» подзапрос отбирает заказы, в которых фигурируют этот клиент и этот товар. Если ни одного такого заказа не найдено (то есть этот клиент не заказывал данный товар), то выполнено условие отбора в «среднем» подзапросе. Следовательно, «средний» подзапрос выдает непустой список товаров, которые не были заказаны данным клиентом, условие внешнего запроса не выполняется и запись об этом клиенте не попадет в результат запроса. Если же оказывается, что этот клиент заказывал данный товар, то, наоборот, условие отбора в «среднем» подзапросе не выполняется, «средний» подзапрос возвращает пустой результат, а значит, условие отбора во внешнем запросе выполнено и запись об этом клиенте будет включена в результат запроса.

Поскольку в нашей базе данных нет ни одного клиента, который бы заказал все наименования товаров (см. листинги 3.5 и 3.6 в главе 3), рассмотренный нами запрос возвращает пустой результат.

Далее мы рассмотрим операторы `IN` и `NOT IN` применительно к вложенным запросам.

IN и NOT IN

Операторы `IN` и `NOT IN`, с которыми мы познакомились в соответствующем подразделе выше, позволяют проверить, содержится ли определенное значение в результате подзапроса. Рассмотрим еще один пример использования оператора `IN`:

```
SELECT * FROM Customers WHERE '2007-12-12' IN
  (SELECT date FROM Orders WHERE Customers.id=customer_id);
```

Для каждого клиента, то есть строки таблицы `Customers`, вложенный запрос выдает даты заказов этого клиента. Если дата 2007-12-12 есть среди этих дат, то строка таблицы `Customers` включается в результат запроса. Таким образом, запрос выводит информацию о тех клиентах, которые сделали заказ 12 декабря 2007 г. Результат этого запроса представлен в табл. 4.11 выше.

Вложенный запрос, результат которого обрабатывается с помощью оператора `IN`, может возвращать несколько столбцов, но в этом случае и значение слева от оператора должно быть составным с таким же количеством компонентов (о составных значениях рассказывалось выше в этой главе).

Далее мы рассмотрим ключевые слова `ANY` и `SOME`.

ANY, SOME

Ключевое слово `ANY` («какой-либо») используется совместно с операторами сравнения, описанными в подразделе выше. При использовании `ANY` результат сравнения будет верным, если он верен хотя бы для одного из значений, выданных подзапросом. Иными словами, строго говоря, результатом вычисления выражения:

$x < \text{Оператор сравнения} > \text{ANY} (< \text{Вложенный запрос} >)$

является следующее значение:

- 1 (`true`), если среди выданных подзапросом значений есть хотя бы одно значение y , для которого выполнено условие $x < \text{Оператор сравнения} > y$;
- 0 (`false`), если среди выданных подзапросом значений нет ни одного такого значения y , для которого выражение $x < \text{Оператор сравнения} > y$ истинно (`true`) или не определено (`NULL`), в том числе если подзапрос возвращает пустой результат;
- `NULL`, если среди выданных подзапросом значений нет ни одного такого значения y , для которого выражение $x < \text{Оператор сравнения} > y$ истинно (`true`), но в то же время есть одно или несколько значений y , для которых это выражение не определено (`NULL`).

Например, вывести информацию о клиентах, которые сделали хотя бы один заказ на сумму более 5000, можно с помощью запроса:

```
SELECT * FROM Customers WHERE 5000 < ANY  
(SELECT amount FROM Orders WHERE Customers.id=customer_id);
```

Для каждого клиента вложенный подзапрос получает из таблицы `Orders` суммы заказов (столбец `amount`) этого клиента. Затем эти суммы сравниваются с вели-

чиной 5000, и запись о клиенте попадет в результат запроса, если хотя бы одна из этих сумм превышает 5000. Таким образом, запрос возвращает результат, представленный в табл. 4.11.

Надо отметить, что вложенный запрос может быть только *правым* операндом для оператора сравнения: например, рассмотренный выше запрос нельзя переписать в виде:

```
SELECT * FROM Customers WHERE  
  ANY (SELECT amount FROM Orders  
        WHERE Customers.id=customer_id)  
  > 5000;
```

Ключевое слово *SOME* является синонимом ключевого слова *ANY*.

В следующем подразделе мы рассмотрим ключевое слово *ALL*.

ALL

Ключевое слово *ALL* («все»), как и *ANY*, используется совместно с операторами сравнения, описанными в соответствующем подразделе выше. При использовании *ALL* результат сравнения будет верным, если он верен для *всех* значений, выданных подзапросом. Иными словами, результатом вычисления выражения:

x <Оператор сравнения> *ALL* <Вложенный запрос>

является следующее значение:

- 1 (true), если условие x <Оператор сравнения> y выполнено для всех y , выданных подзапросом, а также если подзапрос возвращает пустой результат;
- 0 (false), если среди выданных подзапросом значений есть такое значение y , для которого выражение x <Оператор сравнения> y ложно (false);
- NULL в остальных случаях.

Например, запрос:

```
SELECT * FROM Customers WHERE 5000 < ALL  
(SELECT amount FROM Orders WHERE Customers.id=customer_id);
```

выведет информацию не только о тех клиентах, у которых в каждом из заказов сумма превышает 5000, но и о тех, кто не сделал ни одного заказа, ведь в последнем случае результат запроса окажется пустым и условие отбора во внешнем запросе будет выполнено. Таким образом, запрос возвращает результат, представленный в табл. 4.15.

Чтобы исключить тех клиентов, для которых нет зарегистрированных заказов, можно ввести дополнительное условие отбора, например:

```
SELECT * FROM Customers
WHERE 5000 < ALL
  (SELECT amount FROM Orders
   WHERE Customers.id=customer_id)
AND EXISTS
  (SELECT amount FROM Orders
   WHERE Customers.id=customer_id);
```

Как и при использовании ключевого слова ANY, в запросе с ключевым словом ALL вложенный запрос может быть только правым операндом для оператора сравнения.

Итак, мы рассмотрели операторы сравнения, в том числе их применение для обработки результатов вложенных запросов. Теперь перейдем к изучению логических операторов, с помощью которых вы можете создавать составные условия отбора.

Логические операторы

Логические операторы позволяют построить сложное условие отбора на основе операторов сравнения. Операнды логических операторов рассматриваются как логические значения: `true`, `false` и `NULL`. При этом число 0 и нулевые дата и/или время (`0000-00-00 00:00:00`) считаются ложными значениями (`false`), а отличные от нуля числа и даты — истинными значениями (`true`) (более подробно об этом рассказывалось выше в этой главе).

Начнем с изучения оператора `AND`.

x AND y

Оператор «И» возвращает следующие значения:

- ❑ 1 (`true`), если оба операнда — истинные значения;
- ❑ 0 (`false`), если один или оба операнда — ложные значения;
- ❑ `NULL` в остальных случаях.

Иными словами, если вы соединили два условия отбора с помощью оператора `AND`, то составное условие выполняется, только когда выполняются одновременно оба составляющих условия.

Например, запрос:

```
SELECT * FROM Customers
WHERE name LIKE '000%' AND rating>1000;
```

не выводит ни одной строки. Хотя в таблице `Customers` есть имена, начинающиеся с «000», и рейтинги, превышающие 1000, ни одна из строк не удовлетворяет обоим этим условиям одновременно.

Выражение `&&` является синонимом `AND`.

Следующий оператор, который мы рассмотрим, — оператор `OR`.

x OR y

Оператор «ИЛИ» возвращает такие значения:

- ☐ 1 (`true`), если один или оба операнда — истинные значения;
- ☐ 0 (`false`), если оба операнда — ложные значения;
- ☐ `NULL` в остальных случаях.

Иными словами, если вы соединили два условия отбора с помощью оператора `OR`, то составное условие выполняется, когда выполняется хотя бы одно из составляющих условий.

Например, запрос:

```
SELECT * FROM Customers
WHERE name LIKE '000%' OR rating>1000;
```

выводит и те строки таблицы `Customers`, в которых имя начинается с «000», и те, в которых рейтинг больше 1000 (см. табл. 4.6).

Выражение `||` является синонимом `OR`.

Следующий оператор, который мы рассмотрим, — оператор `XOR`.

x XOR y

Оператор «исключающее ИЛИ» возвращает следующие значения:

- ☐ 1 (`true`), если один из операндов — истинное значение, а другой — ложное значение;
- ☐ 0 (`false`), если оба операнда — истинные значения или оба — ложные значения;
- ☐ `NULL`, если хотя бы один из операндов равен `NULL`.

Иными словами, если вы соединили два условия отбора с помощью оператора XOR, то составное условие выполняется, когда выполняется *ровно одно* из составляющих условий.

Например, запрос:

```
SELECT * FROM Customers  
WHERE name LIKE '000%' XOR rating>500;
```

выводит те строки таблицы `Customers`, в которых имя начинается с «000», и те, в которых рейтинг больше 500, за исключением тех строк, в которых эти условия выполняются одновременно (см. табл. 4.15).

И наконец, рассмотрим последний логический оператор — оператор NOT.

NOT x

Оператор «НЕ» («отрицание») возвращает следующие значения:

- ❑ 1 (true), если операнд — ложное значение;
- ❑ 0 (false), если операнд — истинное значение;
- ❑ NULL, если операнд равен NULL.

Иными словами, условие отбора, созданное с помощью оператора NOT, выполняется, если исходное условие не выполнено и не равно NULL.

Например, запрос:

```
SELECT * FROM Customers  
WHERE NOT (name LIKE '000%' OR rating>1000);
```

выводит те строки таблицы `Customers`, для которых условие `name LIKE '000%' OR rating>1000` не выполнено. Таким образом, запрос возвращает результат, представленный в табл. 4.1.

Завершая изучение операторов и функций проверки условий, обсудим еще несколько функций, используемых для сравнения различных величин. Эти функции отличаются тем, что возвращаемое ими значение не обязательно должно быть логическим.

Операторы и функции, основанные на сравнении

В этом подразделе мы кратко рассмотрим операторы и функции, которые выполняют сравнение двух или нескольких величин, однако возвращают не логическое

значение (true, false или NULL), а один из своих аргументов (или порядковый номер аргумента). Перечислю эти функции.

- ❑ $\text{LEAST}(a_1, a_2, \dots, a_n)$ — возвращает наименьший из своих аргументов (либо NULL, если один из аргументов равен NULL). Например, выражение $\text{LEAST}('000 \text{ "Кускус"}', 'Петров', 'Крылов')$ возвращает значение 'Кускус'. Отмечу, что в функции $\text{LEAST}()$ можно указать только фиксированное количество аргументов. Например, невозможно получить первое в алфавитном порядке имя клиента с помощью запроса $\text{SELECT LEAST}(name) \text{ FROM Customers};$. Вместо этого необходимо использовать групповую функцию $\text{MIN}()$ (о ней будет рассказано чуть ниже).
- ❑ $\text{GREATEST}(a_1, a_2, \dots, a_n)$ — возвращает наибольший из своих аргументов (либо NULL, если по крайней мере один из аргументов равен NULL). Например, выражение $\text{GREATEST}('000 \text{ "Кускус"}', 'Петров', 'Крылов')$ возвращает значение 'Петров'. Как и в функции $\text{LEAST}()$, в функции $\text{GREATEST}()$ можно указать только фиксированное количество аргументов. Например, невозможно получить последнее в алфавитном порядке имя клиента с помощью запроса $\text{SELECT GREATEST}(name) \text{ FROM Customers};$. Вместо этого необходимо использовать групповую функцию $\text{MAX}()$ (о ней будет рассказано чуть ниже).
- ❑ $\text{INTERVAL}(a, b_1, b_2, \dots, b_n)$, где $b_1 < b_2 < \dots < b_n$ — функция возвращает порядковый номер наибольшего из чисел b_i , не превосходящих числа a :
 - если $b_i \leq a < b_{i+1}$, то функция возвращает номер i ;
 - если $a < b_1$, то функция возвращает значение 0;
 - если $a > b_n$, то функция возвращает значение n ;
 - если a равно NULL, то функция возвращает значение -1.

Все аргументы этой функции являются целыми числами (если вы укажете аргумент с другим типом данных, то он будет преобразован к целочисленному значению). Чтобы функция возвращала корректный результат, необходимо, чтобы значения b_i были упорядочены, то есть выполнялось условие $b_1 < b_2 < \dots < b_n$.

Например, выражение $\text{INTERVAL}(1500, 1000, 2000, 3000)$ возвращает значение 1.

- ❑ $\text{COALESCE}(a_1, a_2, \dots, a_n)$ — возвращает первый из аргументов, который отличен от NULL (а если все аргументы равны NULL, то возвращает NULL). Например, выражение $\text{COALESCE}(\text{NULL}, 1/0, \text{'Текст'})$ возвращает значение 'Текст', поскольку это первый аргумент, отличный от NULL (при делении на 0 результатом будет NULL).

- $IF(a, b, c)$ — проверяет, является ли истинным логическое значение или выражение a . Если a истинно (то есть является числом, датой или временем, отличными от нулевых), то функция возвращает значение b , а если a ложно или равно NULL, функция возвращает значение c . Например, если требуется удвоить те рейтинги клиентов, которые превышают 1000, это можно сделать с помощью команды:

```
UPDATE Customers
```

```
SET rating=IF(rating>1000,rating*2,rating);
```

- $IFNULL(a, b)$ — функция возвращает значение a , если это значение отлично от NULL, и значение b , если a равно NULL. Например, если всем клиентам, чей рейтинг не указан (равен NULL), требуется присвоить рейтинг 500, это можно сделать с помощью команды:

```
UPDATE Customers SET rating=IFNULL(rating,500);
```

- $NULLIF(a, b)$ — возвращает значение NULL, если $a = b$, и значение a в противном случае. Например, если требуется выполнить операцию, обратную операции из предыдущего пункта, то есть всем клиентам с рейтингом 500 присвоить неопределенный рейтинг, это можно сделать с помощью команды:

```
UPDATE Customers SET rating=NULLIF(rating,500);
```

Рассмотрим еще оператор CASE.

```
CASE x WHEN  $a_1$  THEN  $b_1$ 
[WHEN  $a_2$  THEN  $b_2$ ]
...
[WHEN  $a_n$  THEN  $b_n$ ]
[ELSE  $b_0$ ]
END
```

или

```
CASE WHEN  $x_1$  THEN  $b_1$ 
[WHEN  $x_2$  THEN  $b_2$ ]
...
[WHEN  $x_n$  THEN  $b_n$ ]
[ELSE  $b_0$ ]
END
```

Оператор CASE обеспечивает последовательную проверку списка условий и возвращает значение в зависимости от того, какое из условий выполнено. В первом варианте значение выражения x сравнивается со значениями a_1, a_2, \dots, a_n :

- ❑ если $x = a_i$, то оператор возвращает значение b_i ;
- ❑ если значение выражения x не совпало ни с одним из a_i , то оператор возвращает значение b_0 , заданное параметром ELSE;
- ❑ если значение выражения x не совпало ни с одним из a_i , а параметр ELSE не задан, то оператор возвращает NULL.

Во втором варианте последовательно проверяется истинность логических выражений x_i :

- ❑ если x_i истинно, то оператор возвращает значение b_i ;
- ❑ если ни одно из выражений x_i не является истинным, то оператор возвращает значение b_0 , заданное параметром ELSE;
- ❑ если ни одно из выражений x_i не является истинным, а параметр ELSE не задан, то оператор возвращает значение NULL.

Например, запрос:

```
SELECT date,customer_id,amount,
CASE WHEN amount<=5000 THEN 'Малый'
      WHEN amount BETWEEN 5000 AND 15000 THEN 'Средний'
      WHEN amount>15000 THEN 'Крупный'
END
FROM Orders
ORDER BY customer_id,amount DESC;
```

выводит классификацию заказов в зависимости от их стоимости (табл. 4.22).

Таблица 4.22. Результат выполнения запроса

Date	customer_id	amount	CASE WHEN amount<=5000 THEN 'Малый' WHEN amount BETWEEN 5000 AND 15000 THEN 'Средний' WHEN amount>15000 THEN 'Крупный' END
2008-01-21	533	5750.00	Средний
2007-12-12	533	4500.00	Малый
2007-12-12	536	22000.00	Крупный

Итак, мы рассмотрели операторы и функции, с помощью которых можно сравнивать между собой различные величины, в том числе сопоставлять значение с результатом подзапроса, а также проверять выполнение различных условий. Следующий важный и часто используемый класс функций — групповые функции.

4.2. Групповые функции

Групповые, или агрегатные, функции используются для получения итоговой, сводной информации на основе значений, хранящихся в столбце таблицы. В данном разделе рассказывается об этих функциях, а также об особенностях синтаксиса запросов, использующих групповые функции.

Перечень групповых функций

Для вычисления обобщающего значения столбца таблицы предназначены следующие функции.

SUM()

Функция возвращает сумму значений в столбце. Неопределенные значения при этом не учитываются. Если запросом не найдено ни одной строки или все значения в столбце равны NULL, то функция возвращает значение NULL.

Например, запрос:

```
SELECT SUM(rating) FROM Customers;
```

возвращает сумму рейтингов клиентов — величину 1000 + 1500 + 1000 (табл. 4.23).

Таблица 4.23. Результат выполнения запроса

SUM(rating)
3500

Исключить повторяющиеся значения при подсчете суммы можно с помощью параметра `DISTINCT`. Если он указан, то каждое значение столбца будет учтено в сумме только один раз, даже если в столбце оно встречается несколько раз. Например, запрос:

```
SELECT SUM(DISTINCT rating) FROM Customers;
```

подсчитывает сумму только несовпадающих рейтингов (табл. 4.24).

Таблица 4.24. Результат выполнения запроса

SUM(rating)
2500

Число, возвращаемое этим запросом, является суммой значений 1000 и 1500; еще одно значение 1000, имеющееся в столбце `rating`, запросом игнорируется.

Если в запросе вы укажете какое-либо условие отбора, то суммирование произойдет только по тем строкам, которые удовлетворяют этому условию. Например, запрос:

```
SELECT SUM(amount) FROM Orders WHERE customer_id=533;
```

вычисляет общую сумму заказов клиента с идентификатором 533 (табл. 4.25).

Таблица 4.25. Результат выполнения запроса

SUM(amount)
10250.00

В следующем подразделе мы рассмотрим функцию вычисления среднего значения.

AVG()

Функция возвращает среднее арифметическое значений в столбце (сумму значений, деленную на их количество). Неопределенные значения при этом не учитываются. Если в запросе вы укажете какое-либо условие отбора, то суммирование произойдет только по тем строкам, которые удовлетворяют этому условию. Если запросом не найдено ни одной строки или все значения в столбце равны NULL, то функция возвращает значение NULL.

Например, запрос:

```
SELECT AVG(rating) FROM Customers;
```

возвращает средний рейтинг клиентов — величину $(1000 + 1500 + 1000)/3$ (табл. 4.26).

Таблица 4.26. Результат выполнения запроса

AVG(rating)
1166.6667

Исключить повторяющиеся значения при подсчете среднего можно с помощью параметра `DISTINCT`. Например, запрос:

```
SELECT AVG(DISTINCT rating) FROM Customers;
```

подсчитывает среднее только несовпадающих рейтингов — величину $(1000 + 1500) / 2$; еще одно значение 1000, имеющееся в столбце `rating`, запросом игнорируется (табл. 4.27).

Таблица 4.27. Результат выполнения запроса

AVG(rating)
1250.0000

Функцию `AVG()` можно использовать для отбора тех значений, которые больше среднего, или тех, которые меньше среднего. Например, запрос:

```
SELECT * FROM Customers
WHERE rating > (SELECT AVG(rating) FROM Customers);
```

выводит информацию о клиентах, чей рейтинг выше среднего (см. результат запроса в табл. 4.12). Вложенный запрос возвращает средний рейтинг клиента (см. табл. 4.26), а внешний — отбирает строки таблицы `Customers`, в которых значение столбца `rating` больше значения, возвращенного подзапросом. Отмечу, что в данном случае вложенный запрос возвращает *единственное* значение, поэтому с оператором «больше» нет необходимости использовать ключевое слово `ANY` или `ALL`.

В следующем подразделе мы рассмотрим функцию нахождения максимального значения столбца.

MAX()

Эта функция возвращает максимальное значение в столбце. Если в запросе укажете какое-либо условие отбора, то максимальное значение будет выбрано из строк, удовлетворяющих этому условию. Если запросом не найдено ни одной строки или все значения в столбце равны `NULL`, то функция возвращает значение `NULL`.

Например, запрос:

```
SELECT MAX(rating) FROM Customers;
```

возвращает наибольший из рейтингов клиентов — 1500 (табл. 4.28).

Таблица 4.28. Результат выполнения запроса

MAX(rating)
1500

Функцию `MAX()` можно использовать для поиска строк, в которых достигается максимальное значение столбца. Например, запрос:

```
SELECT * FROM Customers  
WHERE rating = (SELECT MAX(rating) FROM Customers);
```

выводит информацию о клиентах, чей рейтинг равен максимальному (см. результат запроса в табл. 4.12).

В следующем подразделе мы рассмотрим функцию нахождения минимального значения столбца.

MIN()

Данная функция возвращает минимальное значение в столбце. Если в запросе вы укажете какое-либо условие отбора, то минимальное значение будет выбрано из строк, удовлетворяющих этому условию. Если запросом не найдено ни одной строки или все значения в столбце равны NULL, то функция возвращает значение NULL.

Например, запрос:

```
SELECT MIN(rating) FROM Customers;
```

возвращает наибольший из рейтингов клиентов — 1000 (табл. 4.29).

Таблица 4.29. Результат выполнения запроса

MIN(rating)
1000

Функцию MIN() можно использовать для поиска строк, в которых достигается минимальное значение столбца. Например, запрос:

```
SELECT * FROM Customers  
WHERE rating = (SELECT MIN(rating) FROM Customers);
```

выводит информацию о клиентах, чей рейтинг равен максимальному (см. результат запроса в табл. 4.11).

В следующем подразделе мы рассмотрим функцию подсчета количества значений.

COUNT()

Функция возвращает количество отличных от NULL значений, которые содержатся в столбце. Если в запросе вы укажете условие отбора, то в подсчете будут

участвовать только строки, удовлетворяющие ему. Если не найдено ни одного отличного от NULL значения, то функция возвращает 0.

Например, запрос:

```
SELECT COUNT(rating) FROM Customers;
```

возвращает количество отличных от NULL значений в столбце `rating` таблицы `Customers` (табл. 4.30).

Таблица 4.30. Результат выполнения запроса

COUNT(rating)
3

Параметр `DISTINCT` позволяет подсчитать количество различных (уникальных) значений в столбце (при этом неопределенные значения также игнорируются). Например, запрос:

```
SELECT COUNT(DISTINCT rating) FROM Customers;
```

подсчитывает количество различных значений рейтинга в таблице `Customers` (табл. 4.31). Поскольку в таблице есть две строки с одинаковым рейтингом 1000, то результат подсчета будет меньше, чем в предыдущем запросе.

Таблица 4.31. Результат выполнения запроса

COUNT(rating)
2

Если в качестве аргумента функции `COUNT ()` указать не имя столбца, а звездочку, то функция возвратит общее количество строк, удовлетворяющих условию отбора, включая строки, содержащие неопределенные значения. Так, если столбец `rating` содержит неопределенные значения, то значение, выводимое запросом:

```
SELECT COUNT(*) FROM Customers;
```

будет больше значения, выводимого запросом:

```
SELECT COUNT(rating) FROM Customers;
```

(разность этих значений совпадает с количеством строк, в которых значение в столбце `rating` равно NULL).

Функцию `COUNT ()` можно использовать для отбора тех строк родительской таблицы, с которыми связано заданное количество строк дочерней таблицы. Например, запрос:


```
SELECT * FROM Customers
WHERE 2 <= (SELECT COUNT(*) FROM Orders
WHERE Customers.id=customer_id);
```

выводит список клиентов, сделавших не менее двух заказов (результат запроса см. в табл. 4.14). Для каждого клиента вложенный запрос выдает количество заказов этого клиента, и если это количество не меньше двух, то текущая запись о клиенте включается в результат, выводимый внешним запросом.

В следующем разделе мы расскажем о функциях вычисления среднеквадратичного отклонения.

VAR_POP(), VARIANCE(), VAR_SAMP(), STDDEV_POP(), STD(), STDDEV(), STDDEV_SAMP()

Функция `VAR_POP()` вычисляет *дисперсию* значений столбца. Дисперсия характеризует колебание значений от среднего. Если a_1, a_2, \dots, a_n — значения столбца,

$a = \frac{a_1 + a_2 + \dots + a_n}{n}$ — среднее арифметическое значений столбца, то дисперсия равна $\frac{\sum_{i=1}^n (a_i - a)^2}{n}$.

Например, запрос:

```
SELECT VAR_POP(rating) FROM Customers;
```

возвращает величину дисперсии рейтингов клиентов:

$$\frac{(1000 - 1166,6667)^2 + (1500 - 1166,6667)^2 + (1000 - 1166,6667)^2}{3} = 55555,5556$$

(табл. 4.32).

Таблица 4.32. Результат выполнения запроса

VAR_POP(rating)
55555.5556

Функция `VARIANCE()` является синонимом `VAR_POP()`.

Функция `VAR_SAMP()` возвращает величину *выборочной*, или *несмещенной*, дисперсии (в математической статистике выборочная дисперсия является оценкой дисперсии всей изучаемой совокупности значений, при этом значения, по которым

вычисляется несмещенная дисперсия, рассматриваются как выборка из изучаемой

совокупности). Если a_1, a_2, \dots, a_n — значения столбца, $a = \frac{a_1 + a_2 + \dots + a_n}{n}$ — среднее арифметическое значений столбца, то значение выборочной дисперсии равно

$$\frac{\sum_{i=1}^n (a_i - a)^2}{n - 1}.$$

Например, запрос:

```
SELECT VAR_SAMP(rating) FROM Customers;
```

возвращает величину выборочной дисперсии рейтингов клиентов:

$$\frac{(1000 - 1166,6667)^2 + (1500 - 1166,6667)^2 + (1000 - 1166,6667)^2}{2} = 83333,3333$$

(табл. 4.33).

Таблица 4.33. Результат выполнения запроса

VAR_SAMP(rating)
83333.3333

Функция `STDDEV_POP()` вычисляет среднеквадратичное отклонение значений столбца, которое является квадратным корнем из дисперсии.

Например, запрос:

```
SELECT STDDEV_POP(rating) FROM Customers;
```

возвращает величину:

$$\sqrt{\frac{(1000 - 1166,6667)^2 + (1500 - 1166,6667)^2 + (1000 - 1166,6667)^2}{3}} = 235,7023$$

(табл. 4.34).

Таблица 4.34. Результат выполнения запроса

STDDEV_POP(rating)
235.7023

Функции `STD()` и `STDDEV()` являются синонимами `STDDEV_POP()`.

Функция `STDDEV_SAMP()` вычисляет квадратный корень из выборочной дисперсии.

Например, запрос:

```
SELECT STDDEV_SAMP(rating) FROM Customers;
```

возвращает величину:

$$\sqrt{\frac{(1000 - 1166,6667)^2 + (1500 - 1166,6667)^2 + (1000 - 1166,6667)^2}{2}} = 288,6751$$

(табл. 4.35).

Таблица 4.35. Результат выполнения запроса

STDDEV_SAMP(rating)
288.6751

При вычислении всех вышеперечисленных функций неопределенные значения не учитываются. Если в запросе вы укажете какое-либо условие отбора, то в вычислениях будут участвовать только те строки, которые удовлетворяют этому условию. Если запросом не найдено ни одной строки или все значения в столбце равны NULL, то все эти функции возвращают значение NULL.

В следующем подразделе мы рассмотрим функцию объединения строк.

GROUP_CONCAT()

Функция `GROUP_CONCAT()` объединяет в одну строку значения столбца. При этом неопределенные значения не учитываются. Если в запросе вы укажете какое-либо условие отбора, то будут объединены значения только из тех строк, которые удовлетворяют условию отбора. Если запросом не найдено ни одной строки или все значения в столбце равны NULL, то функция возвращает значение NULL.

Например, запрос:

```
SELECT GROUP_CONCAT(name) FROM Customers;
```

возвращает строку, содержащую имена клиентов (табл. 4.36).

Таблица 4.36. Результат выполнения запроса

GROUP_CONCAT(name)
ООО "Кускус", Петров, Крылов

При использовании функции `GROUP_CONCAT()` вы также можете указать дополнительные параметры:

- ☐ `DISTINCT` — исключает при объединении повторяющиеся значения;
- ☐ `ORDER BY` — упорядочивает объединяемые значения;
- ☐ `SEPARATOR` — задает разделитель значений.

Например, запрос:

```
SELECT GROUP_CONCAT(DISTINCT name  
ORDER BY name ASC SEPARATOR ';') FROM Customers;
```

возвращает строку, содержащую имена клиентов без повторений, упорядоченные по алфавиту и разделенные точкой с запятой (табл. 4.37).

Таблица 4.37. Результат выполнения запроса

GROUP_CONCAT(DISTINCT name ORDER BY name ASC SEPARATOR ';')
Крылов; ООО "Кускус"; Петров

Итак, мы изучили все основные групповые функции (за рамками нашего рассмотрения остались функции `BIT_AND()` — побитовое «И», `BIT_OR()` — побитовое ИЛИ и `BIT_XOR()` — побитовое «исключающее ИЛИ»). В следующем подразделе мы рассмотрим ключевое слово `GROUP BY`, с помощью которого можно вычислять групповые функции одновременно для нескольких групп строк.

Параметр GROUP BY

Выше мы рассматривали примеры запросов, в которых групповые функции вычисляют обобщающее значение для *всех* строк, удовлетворяющих условию отбора. Параметр `GROUP BY` позволяет объединять строки в группы, для каждой из которых групповая функция вычисляется отдельно. Для этого в параметре `GROUP BY` нужно указать столбец или несколько столбцов: в одну группу попадут строки с одинаковым набором значений в этих столбцах.

Например, запрос:

```
SELECT customer_id, SUM(amount) FROM Orders  
GROUP BY customer_id;
```

возвращает общую сумму заказов отдельно для каждого клиента (табл. 4.38). В этом запросе заказы сгруппированы по значению столбца `customer_id`, поэтому каждая группа состоит из заказов одного клиента, а функция `SUM(amount)` вычисляет сумму заказов в каждой из групп.

Таблица 4.38. Результат выполнения запроса

customer_id	SUM(amount)
533	10250.00
536	22000.00

Таким же образом можно подсчитать количество заказов каждого клиента, максимальную, минимальную и среднюю сумму заказа и др.

Другой пример — запрос, возвращающий имена клиентов с одинаковым значением рейтинга:

```
SELECT GROUP_CONCAT(name),rating FROM Customers  
GROUP BY rating;
```

Этот запрос группирует клиентов по значению рейтинга и выводит имена клиентов в каждой группе (табл. 4.39).

Таблица 4.39. Результат выполнения запроса

GROUP_CONCAT(name)	rating
ООО "Кускус", Крылов	1000
Петров	1500

Если указано ключевое слово `WITH ROLLUP`, то обобщенные значения выводятся как для отдельных групп строк, так и для всей совокупности строк.

Например, запрос:

```
SELECT customer_id, SUM(amount) FROM Orders  
GROUP BY customer_id WITH ROLLUP;
```

возвращает, помимо общей суммы заказов каждого клиента, сумму всех заказов (табл. 4.40).

Таблица 4.40. Результат выполнения запроса

customer_id	SUM(amount)
533	10250.00
536	22000.00
NULL	32250.00

По сравнению с табл. 4.38 здесь появилась итоговая строка, содержащая общую сумму всех заказов.

В запросе с параметром `GROUP BY` вы можете использовать условия как для отбора отдельных строк перед группировкой, так и для отбора групп строк. Если требуется выбрать из таблицы строки, удовлетворяющие какому-либо критерию, а затем объединить в группы только эти строки, то применяется параметр `WHERE`, который должен быть указан *перед* параметром `GROUP BY`. Например, запрос:

```
SELECT customer_id, COUNT(amount) FROM Orders  
WHERE amount>5000  
GROUP BY customer_id;
```

позволяет подсчитать, сколько заказов на сумму более 5000 сделал каждый клиент (табл. 4.41). Сначала выбираются строки таблицы `Orders`, для которых выполнено условие `amount>5000`, далее эти строки группируются по значению столбца `customer_id` и после этого вычисляется количество строк в каждой из групп.

Таблица 4.41. Результат выполнения запроса

customer_id	COUNT(amount)
533	1
536	1

Для отбора групп строк служит параметр `HAVING`, о котором мы поговорим в следующем подразделе.

Параметр `HAVING`

Параметр позволяет задать условие отбора для групп строк. Он аналогичен параметру `WHERE`, но указывается *после* параметра `GROUP BY` и применяется к агрегированным строкам. В условии отбора параметра `HAVING` можно использовать значения столбцов, выводимых запросом, в том числе значения агрегатных функций.

Например, если требуется вывести общую сумму заказов для каждого клиента, кроме тех клиентов, для кого эта сумма меньше 20 000, выполним запрос:

```
SELECT customer_id, SUM(amount) FROM Orders
GROUP BY customer_id
HAVING SUM(amount)>=20000;
```

Условие `SUM(amount)>=20000` позволяет отобрать только те группы строк, в которых общая сумма заказа равна или превышает 20 000 (табл. 4.42).

Таблица 4.42. Результат выполнения запроса

customer_id	SUM(amount)
536	22000.00

Итак, мы изучили запросы с групповыми функциями, позволяющими получать из таблиц обобщенные данные. Далее мы кратко рассмотрим некоторые полезные функции, оперирующие числовыми величинами.

4.3. Числовые операторы и функции

В данном разделе мы поговорим об основных операторах и функциях, используемых для вычислений: арифметических, алгебраических и тригонометрических. Наиболее часто применяются арифметические операторы.

Арифметические операторы

В выражениях вы можете использовать следующие арифметические операторы.

- ❑ $a + b$ — оператор сложения. Возвращает сумму операндов a и b .
- ❑ $a - b$ — оператор вычитания. Возвращает разность операндов a и b . При использовании с одним операндом меняет его знак, например $-(3 + 2) = -5$.
- ❑ $a * b$ — оператор умножения. Возвращает произведение операндов a и b .
- ❑ a / b — оператор деления. Возвращает частное от деления a на b .
- ❑ $a \text{ DIV } b$ — оператор деления с остатком, или целочисленного деления. Возвращает целую часть частного от деления a на b . Например:
 - $7 \text{ DIV } 2 = 3$;
 - $(-7) \text{ DIV } 2 = -3$;
 - $7 \text{ DIV } (-2) = -3$;
 - $(-7) \text{ DIV } (-2) = 3$.
- ❑ $a \% b$ — оператор вычисления остатка. Возвращает остаток от целочисленного деления a на b : величину $a \% b = a - b \times (a \text{ DIV } b)$. Например:
 - $7 \% 2 = 1$;
 - $(-7) \% 2 = -1$;
 - $7 \% (-2) = 1$;
 - $(-7) \% (-2) = -1$.

В следующем подразделе мы рассмотрим алгебраические функции.

Алгебраические функции

В выражениях вы можете использовать следующие алгебраические функции.

- ❑ $\text{ABS}(x)$ — возвращает абсолютную величину (модуль) числа x . Например, $\text{ABS}(10) = \text{ABS}(-10) = 10$.

❑ $\text{CEIL}(x)$, $\text{CEILING}(x)$ — функция округления в большую сторону. Возвращает наименьшее из целых чисел, которые больше или равны x . Например:

- $\text{CEIL}(12345.6789) = 12\,346$;
- $\text{CEIL}(-12345.6789) = -12\,345$.

❑ $\text{CRC32}(\text{'Символьное значение'})$ — вычисляет контрольную сумму для последовательности символов с помощью алгоритма CRC32. Подробнее об алгоритмах CRC вы можете прочитать здесь: <http://ru.wikipedia.org/wiki/CRC32>.

Например, $\text{CRC32}(\text{'Век живи — век учись'}) = 4\,171\,076\,480$.

❑ $\text{EXP}(x)$ — экспонента. Возвращает e^x (экспоненту числа x).

❑ $\text{FLOOR}(x)$ — функция округления в меньшую сторону. Возвращает наибольшее из целых чисел, не превосходящих x . Например:

- $\text{FLOOR}(12345.6789) = 12\,345$;
- $\text{FLOOR}(-12345.6789) = -12\,346$.

❑ $\text{LN}(x)$, $\text{LOG}(x)$ — возвращает $\ln x$ (натуральный логарифм числа x). Таким образом, $\text{LN}(\text{EXP}(y)) = y$.

❑ $\text{LOG10}(x)$ — возвращает $\log_{10} x$ (логарифм числа x по основанию 10).

Например, $\text{LOG10}(100) = 2$.

❑ $\text{LOG2}(x)$ — возвращает $\log_2 x$ (логарифм числа x по основанию 2).

Например, $\text{LOG2}(16) = 4$.

❑ $\text{LOG}(a, x)$ — возвращает $\log_a x$ (логарифм числа x по основанию a).

Например, $\text{LOG}(2, 16) = \text{LOG2}(16) = 4$.

❑ $\text{MOD}(a, b)$ — синоним выражения $a \% b$, возвращает остаток от целочисленного деления a на b .

❑ $\text{PI}()$ — возвращает число $\pi = 3,14159\dots$

❑ $\text{POW}(x, y)$, $\text{POWER}(x, y)$ — функция возведения в степень. Возвращает x^y .

Например, $\text{POW}(2, 10) = 1024$.

❑ $\text{RAND}()$ — возвращает случайное число в интервале от 0 до 1.

❑ $\text{RAND}(x)$ — возвращает псевдослучайное число в интервале от 0 до 1, при этом целое число x используется как начальное значение генератора псевдослучайных чисел. Возвращаемое значение при этом предопределено, например $\text{RAND}(20)$ всегда возвращает значение 0,15888261251047.

- $\text{ROUND}(x)$ — функция округления до целого. Возвращает целое число, ближайшее к x .
- $\text{ROUND}(x, n)$ — функция округления. Если $n > 0$, то возвращает ближайшее к x число с n знаками после разделителя. Если $n = 0$, возвращает ближайшее к x целое число: $\text{ROUND}(x, 0) = \text{ROUND}(x)$. Если $n < 0$, то возвращает ближайшее к x целое число, заканчивающееся на n нулей.

Например:

- $\text{ROUND}(12345.6789, 2) = 12\,345,68$;
 - $\text{ROUND}(12345.6789, 0) = 12\,346$;
 - $\text{ROUND}(12345.6789, -2) = 12\,300$;
 - $\text{ROUND}(-12345.6789, 2) = -12\,345,68$.
- $\text{SIGN}(x)$ — функция получения знака. Возвращает 1, если $x > 0$, значение 0, если $x = 0$, и значение -1 , если $x < 0$.
 - $\text{SQRT}(x)$ — возвращает \sqrt{x} (квадратный корень из x).
 - $\text{TRUNCATE}(x, n)$ — функция отбрасывания «лишних» цифр. Если $n > 0$, то возвращается число, состоящее из целой части числа x и n его первых знаков после разделителя. Если $n = 0$, возвращается целая часть x . Если $n < 0$, то возвращается число, в котором последние n цифр заменены нулями.

Например:

- $\text{TRUNCATE}(12345.6789, 2) = 12\,345,67$;
- $\text{TRUNCATE}(12345.6789, 0) = 12\,345$;
- $\text{TRUNCATE}(12345.6789, -2) = 12\,300$;
- $\text{TRUNCATE}(-12345.6789, 2) = -12\,345,67$.

В следующем подразделе мы рассмотрим тригонометрические функции.

Тригонометрические функции

В выражениях вы можете использовать следующие тригонометрические функции.

- $\text{SIN}(x)$ — возвращает синус угла величиной x радиан.
- $\text{COS}(x)$ — функция возвращает косинус угла величиной x радиан.
- $\text{TAN}(x)$ — возвращает тангенс угла величиной x радиан.

- ❑ $\text{COT}(x)$ — возвращает котангенс угла величиной x радиан.
- ❑ $\text{ASIN}(x)$ — функция возвращает арксинус числа x , то есть величину угла (в радианах, от $-\pi/2$ до $\pi/2$), синус которой равен x .
- ❑ $\text{ACOS}(x)$ — возвращает арккосинус числа x , то есть величину угла (в радианах, от 0 до π), косинус которой равен x .
- ❑ $\text{ATAN}(x)$ — функция возвращает арктангенс числа x , то есть величину угла (в радианах, от $-\pi/2$ до $\pi/2$), синус которой равен x .
- ❑ $\text{ATAN2}(x, y)$, $\text{ATAN}(x, y)$ — возвращает величину угла (в радианах, от $-\pi$ до π) между векторами с координатами $(1; 0)$ и $(x; y)$, иными словами, величину угла между осью абсцисс и прямой, соединяющей точки $(0; 0)$ и $(x; y)$ на координатной плоскости. Совпадает с $\text{ATAN}(y/x)$, если $x > 0$.
- ❑ $\text{DEGREES}(x)$ — функция возвращает градусную меру угла, радианная мера которого равна x радиан. Например, $\text{DEGREES}(\text{PI}()) = 180$.
- ❑ $\text{RADIANS}(x)$ — возвращает радианную меру угла, градусная мера которого равна x градусов. Например, $\text{RADIANS}(180) = 3,1415926535898$.

Итак, мы обсудили основные числовые функции. Далее кратко рассмотрим функции, оперирующие значениями даты и времени.

4.4. Функции даты и времени

В этом разделе мы поговорим о некоторых полезных функциях, выполняющих различные операции с датами: получение текущей даты и/или времени, получение отдельных компонентов даты и/или времени, арифметические операции с датами (сложение, вычитание) и преобразование форматов даты.

В первую очередь познакомимся с функциями, которые возвращают текущую дату и/или время.

Функции получения текущей даты и времени

Для получения текущей даты и времени вы можете использовать следующие функции.

- ❑ $\text{CURDATE}()$, $\text{CURRENT_DATE}()$, CURRENT_DATE — возвращают текущую дату.
- ❑ $\text{CURTIME}()$, $\text{CURRENT_TIME}()$, CURRENT_TIME — функция возвращают текущее время.

- ❑ `NOW()`, `CURRENT_TIMESTAMP()`, `CURRENT_TIMESTAMP`, `LOCALTIME()`, `LOCALTIME`, `LOCALTIMESTAMP()`, `LOCALTIMESTAMP` — возвращают текущую дату и время.
- ❑ `SYSDATE()` — возвращает текущую дату и время Windows. В отличие от остальных функций, которые выводят дату и/или время начала выполнения SQL-команды, функция `SYSDATE()` возвращает время своего вызова. Таким образом, если в одной SQL-команде функция `SYSDATE()` вызывается несколько раз, то возвращаемые ею значения могут быть различны.
- ❑ `UTC_DATE()`, `UTC_DATE` — функция возвращает текущую дату по UTC в формате YYYY-MM-DD (или, в зависимости от контекста, YYYYMMDD).



ПРИМЕЧАНИЕ

UTC — универсальное скоординированное время, аналог гринвичского времени, основанный на атомном отсчете времени.

- ❑ `UTC_TIME()`, `UTC_TIME` — возвращает текущее время по UTC в формате HH:MM:SS (или, в зависимости от контекста, HHMMSS).
- ❑ `UTC_TIMESTAMP()`, `UTC_TIMESTAMP` — функция возвращает текущую дату и время по UTC в формате YYYY-MM-DD HH:MM:SS (или, в зависимости от контекста, YYYYMMDDHHMMSS).

Далее мы рассмотрим функции, позволяющие выделять какую-либо часть даты.

Функции получения компонентов даты и времени

Перечислю функции, получающие в качестве аргумента дату и/или время и возвращающие один из компонентов аргумента:

- ❑ `DATE('<Дата и время>')` — функция получает в качестве аргумента дату или дату и время и возвращает дату, отсекая время. Так, например, `DATE('2007-12-12 12:30:00')` возвращает значение 2007-12-12.
- ❑ `TIME('<Дата и время>')` — получает в качестве аргумента время либо дату и время и возвращает время, отсекая дату. Например, `TIME('2007-12-12 12:30:00')` возвращает значение 12:30:00.
- ❑ `DAY('<Дата или дата и время>')`, `DAYOFMONTH('<Дата или дата и время>')` — функции `DAY()` и `DAYOFMONTH()` получают в качестве аргумента дату или дату и время и выделяют из нее число (номер дня в месяце). Например, `DAY('2007-12-12')` возвращает значение 12.

- ❑ `DAYNAME ('<Дата или дата и время>')` — получает в качестве аргумента дату или дату и время и возвращает наименование соответствующего дня недели. Например, `DAYNAME ('2007-12-12')` возвращает значение `'Wednesday'`, поскольку 12 декабря 2007 г. — среда.

Если требуется получить название дня недели на русском языке, выполним команду:

```
SET LC_TIME_NAMES='ru_RU';
```

или:

```
SET GLOBAL LC_TIME_NAMES='ru_RU';
```

(о том, как действуют команды `SET` и `SET GLOBAL`, рассказывалось в главе 3, когда обсуждалась установка значения переменной `sql_mode`).

ПРИМЕЧАНИЕ



Чтобы установить для отображения дат украинский или белорусский язык, присвойте переменной `lc_time_names` значение `'uk_UA'` или `'be_BY'` соответственно.

- ❑ `DAYOFWEEK ('<Дата или дата и время>')` — функция получает в качестве аргумента дату или дату и время и вычисляет порядковый номер соответствующего дня недели (1 — воскресенье, 2 — понедельник и т. д.). Например, `DAYOFWEEK ('2007-12-12')` возвращает значение 4, и это означает, что 12 декабря 2007 г. — среда.
- ❑ `WEEKDAY ('<Дата или дата и время>')` — получает в качестве аргумента дату или дату и время и вычисляет порядковый номер соответствующего дня недели (0 — понедельник, 1 — вторник и т. д.). Так, `WEEKDAY ('2007-12-12')` возвращает значение 2, и это означает, что 12 декабря 2007 г. — среда.
- ❑ `DAYOFYEAR ('<Дата или дата и время>')` — функция получает в качестве аргумента дату или дату и время и вычисляет для нее порядковый номер дня в году. Например, `DAYOFYEAR ('2007-12-12')` возвращает значение 346.
- ❑ `LAST_DAY ('<Дата или дата и время>')` — получает в качестве аргумента дату или дату и время и возвращает дату, соответствующую последнему дню в месяце, к которому принадлежит исходная дата. Например, `LAST_DAY ('2007-12-12')` возвращает значение `2007-12-31`, поскольку последнее число декабря — 31.
- ❑ `WEEK ('<Дата или дата и время>' [, Правило нумерации])` — функция получает в качестве аргумента дату или дату и время и возвращает номер недели в году. По умолчанию неделя считается начинающейся с воскресенья, и пер-

вой неделей считается та, воскресенье которой принадлежит данному году, а для дней, предшествующих первой неделе, номер недели равен 0. Например, `WEEK('2007-12-31') = 52` и `WEEK('2008-01-01') = 0`.

Вы также можете задать параметр, определяющий правило нумерации недель.

- 0 — неделя считается начинающейся с воскресенья, первая неделя года — целиком находящаяся в этом году, для дней, предшествующих первой неделе, номер недели равен 0. Например, `WEEK('2008-01-01', 0) = 0`.
- 1 — неделя считается начинающейся с понедельника, первая неделя года — та, более трех дней которой находится в этом году, для дней, предшествующих первой неделе, номер недели равен 0. Например, `WEEK('2008-01-01', 1) = 1`.
- 2 — неделя считается начинающейся с воскресенья, первая неделя года — целиком находящаяся в этом году, для дней, предшествующих первой неделе, номер недели равен номеру последней недели в предыдущем году. Например, `WEEK('2008-01-01', 2) = 52`.
- 3 — неделя считается начинающейся с понедельника, первая неделя года — та, более трех дней которой находится в этом году, для дней, предшествующих первой неделе, номер недели равен номеру последней недели в предыдущем году. Например, `WEEK('2008-01-01', 3) = 1`.
- 4 — неделя считается начинающейся с воскресенья, первая неделя года — та, более трех дней которой находится в этом году, для дней, предшествующих первой неделе, номер недели равен 0. Например, `WEEK('2008-01-01', 4) = 1`.
- 5 — неделя считается начинающейся с понедельника, первая неделя года — целиком находящаяся в этом году, для дней, предшествующих первой неделе, номер недели равен 0. Например, `WEEK('2008-01-01', 5) = 0`.
- 6 — неделя считается начинающейся с воскресенья, первая неделя года — та, более трех дней которой находится в этом году, для дней, предшествующих первой неделе, номер недели равен номеру последней недели в предыдущем году. Например, `WEEK('2008-01-01', 6) = 1`.
- 7 — неделя считается начинающейся с понедельника, первая неделя года — целиком находящаяся в этом году, для дней, предшествующих первой неделе, номер недели равен номеру последней недели в предыдущем году. Например, `WEEK('2008-01-01', 7) = 53`.

□ `WEEKOFYEAR('<Дата или дата и время>')` — является синонимом к `WEEKOFYEAR('<Дата или дата и время>', 3)`.

- ❑ `MONTHNAME ('<Дата или дата и время>')` — функция получает в качестве аргумента дату или дату и время и возвращает наименование месяца, которому принадлежит эта дата. Например, `MONTHNAME ('2007-12-12')` возвращает значение `'December'`. О том, как настроить вывод дат на русском языке, рассказывалось при описании функции `DAYNAME ()`.
- ❑ `MONTH ('<Дата или дата и время>')` — получает в качестве аргумента дату или дату и время и возвращает номер месяца, которому принадлежит эта дата. Например, `MONTH ('2007-12-12')` возвращает значение 12.
- ❑ `QUARTER ('<Дата или дата и время>')` — функция получает в качестве аргумента дату или дату и время и возвращает номер квартала, которому принадлежит эта дата. Например, `QUARTER ('2007-12-12')` возвращает значение 4.
- ❑ `YEAR ('<Дата или дата и время>')` — получает в качестве аргумента дату или дату и время и возвращает номер года, которому принадлежит эта дата. Например, `YEAR ('2007-12-12')` возвращает значение 2007.
- ❑ `YEARWEEK ('<Дата или дата и время>' [, Правило нумерации])` — функция получает в качестве аргумента дату или дату и время и возвращает номер года и номер недели в году в формате `YYYYWW`. По умолчанию неделя считается начинающейся с воскресенья, и первой неделей считается та неделя, воскресенье которой принадлежит данному году, а дни, предшествующие первой неделе, считаются относящимися к последней неделе предыдущего года. Например, `YEARWEEK ('2007-12-31') = YEARWEEK ('2008-01-01') = 200752`, и это означает, что обе эти даты относятся к 52-й неделе 2007 г.

Вы также можете задать параметр, определяющий правило нумерации недель. Этот параметр аналогичен соответствующему параметру функции `WEEK ()`, о которой рассказывалось выше, однако для тех дат, для которых функция `WEEK ()` возвращает значение 0, функция `YEARWEEK ()` возвращает номер предыдущего года и номер последней недели предыдущего года. Например, `WEEK ('2008-01-01', 5) = 200753`.

- ❑ `HOURL ('<Время или дата и время>')` — функция получает в качестве аргумента время или дату и время и выделяет из них часы. Например, `HOURL ('12:30:00')` возвращает значение 12.
- ❑ `MINUTE ('<Время или дата и время>')` — получает в качестве аргумента время или дату и время и выделяет из них минуты. Например, команда `MINUTE ('12:30:00')` возвращает значение 30.
- ❑ `SECOND ('<Время или дата и время>')` — функция получает в качестве аргумента время или дату и время и выделяет из них секунды. Например, `SECOND ('12:30:00')` возвращает значение 0.

❑ `EXTRACT ('<Наименование периода>' FROM '<Дата и/или время>')` — это наиболее общая из функций получения компонентов даты и времени. Первым ее аргументом является наименование компонента или диапазона компонентов, которые нужно выделить из даты:

- `DAY` — число (номер дня в месяце);
- `WEEK` — номер недели в году;
- `MONTH` — номер месяца;
- `QUARTER` — номер квартала;
- `YEAR` — номер года;
- `HOURL` — часы;
- `MINUTE` — минуты;
- `SECOND` — секунды;
- `YEAR_MONTH` — номер года и номер месяца;
- `DAY_HOUR` — число и часы;
- `DAY_MINUTE` — число, часы и минуты;
- `DAY_SECOND` — число, часы, минуты и секунды;
- `HOURL_MINUTE` — часы и минуты;
- `HOURL_SECOND` — часы, минуты и секунды;
- `MINUTE_SECOND` — минуты и секунды.

Вторым аргументом функции могут быть дата и время, а также, в зависимости от извлекаемого компонента, либо дата, либо время.

Например, `EXTRACT (WEEK FROM '2007-12-31')` возвращает, как и `WEEK ('2007-12-31')`, значение 52, а `EXTRACT (DAY_MINUTE FROM '2007-12-31 12:30:00')` возвращает значение 311230 (31-е число, 12 часов и 30 минут).

В следующем подразделе будет рассказано о функциях, позволяющих выполнять арифметические операции с датами.

Функции сложения и вычитания дат

Для выполнения арифметических операций вы можете использовать следующие функции.

❑ `ADDDATE ('<Дата или дата и время>', <Количество дней>)` или `ADDDATE ('<Дата или дата и время>', <Временной интервал>)` —

возвращает дату или дату и время, сдвинутые относительно указанной даты на указанное количество дней или на указанный временной интервал. Для задания интервала можно использовать один из следующих основных форматов:

- `INTERVAL '<Количество секунд>' SECOND;`
- `INTERVAL '<Количество минут>' MINUTE;`
- `INTERVAL '<Количество часов>' HOUR;`
- `INTERVAL '<Количество дней>' DAY;`
- `INTERVAL '<Количество недель>' WEEK;`
- `INTERVAL '<Количество месяцев>' MONTH;`
- `INTERVAL '<Количество кварталов>' QUARTER;`
- `INTERVAL '<Количество лет>' YEAR;`
- `INTERVAL '<Количество минут>:<Количество секунд>' MINUTE_SECOND;`
- `INTERVAL '<Количество часов>:<Количество минут>:<Количество секунд>' HOUR_SECOND;`
- `INTERVAL '<Количество часов>:<Количество минут>' HOUR_MINUTE;`
- `INTERVAL '<Количество дней> <Количество часов>:<Количество минут>:<Количество секунд>' DAY_SECOND;`
- `INTERVAL '<Количество дней> <Количество часов>:<Количество минут>' DAY_MINUTE;`
- `INTERVAL '<Количество дней> <Количество часов>' DAY_HOUR;`
- `INTERVAL '<Количество лет>-<Количество месяцев>' YEAR_MONTH.`

Например, функция `ADDDATE('2007-12-12', 28)` добавляет 28 дней к 12 декабря 2007 г. и возвращает результат 2008-01-09, а функция `ADDDATE('2007-12-12', INTERVAL '28 12:30' DAY_MINUTE)` добавляет 28 дней 12 часов и 30 минут к 12 декабря 2007 г. и возвращает результат 2008-01-09 12:30:00.

- ❑ `DATE_ADD('<Дата или дата и время>', '<Временной интервал>')` — синоним `ADDDATE('<Дата или дата и время>', '<Временной интервал>')`.
- ❑ `ADDTIME(<Время или дата и время>, <Добавляемое время>)` — возвращает сумму своих аргументов. Например, функция `ADDTIME('2007-12-12`

12:30:00', '15:50:00') добавляет 15 часов 50 минут к 12 часам 30 минутам 12 декабря 2007 г. и возвращает результат 2007-12-13 04:20:00.

- ❑ `SUBDATE('⟨Дата или дата и время⟩',⟨Количество дней⟩)` или `SUBDATE('⟨Дата или дата и время⟩',⟨Временной интервал⟩)` — функция аналогична функции `ADDDATE()`, только указанное количество дней или указанный временной интервал не добавляются к дате, а вычитаются из нее, иными словами, дата сдвигается в прошлое, а не в будущее. Например, функция `SUBDATE('2007-12-12', INTERVAL '28 12:30' DAY_MINUTE)` вычитает 28 дней 12 часов и 30 минут из 12 декабря 2007 г. и возвращает результат 2007-11-13 11:30:00.
- ❑ `DATE_SUB('⟨Дата или дата и время⟩',⟨Временной интервал⟩)` — синоним `SUBDATE('⟨Дата или дата и время⟩',⟨Временной интервал⟩)`.
- ❑ `SUBTIME(⟨Время или дата и время⟩,⟨Вычитаемое время⟩)` — возвращает разность своих аргументов. Например, функция `SUBTIME('2007-12-12 12:30:00', '15:50:00')` вычитает 15 часов 50 минут из 12 часов 30 минут 12 декабря 2007 г. и возвращает результат 2007-12-11 20:40:00.
- ❑ `DATEDIFF('⟨Дата или дата и время⟩',⟨Дата или дата и время⟩)` — функция возвращает разность в днях между первой и второй датами (время при этом не учитывается). Если первая дата предшествует второй, результат будет отрицательным. Например, `DATEDIFF('2007-12-12 12:30:00', '2007-12-31')` возвращает значение -19.
- ❑ `TIMEDIFF('⟨Время или дата и время⟩',⟨Время или дата и время⟩)` — возвращает разность своих аргументов в формате времени. Если первый момент предшествует второму, результат будет отрицательным. Например, функция `TIMEDIFF('2007-12-12 12:30:00', '2007-12-31 15:50:00')` возвращает значение -459:20:00. Это означает, что 12 декабря 2007 г. 12 часов 30 минут отстоит в прошлое от 31 декабря 2007 г. 15 часов 50 минут на 459 часов 20 минут.
- ❑ `PERIOD_ADD(⟨Период в формате YYMM или YYYYMM⟩,⟨Количество месяцев⟩)` — функция возвращает результат добавления к указанному периоду указанное количество месяцев. Обратите внимание, что оба аргумента этой функции — числа и возвращаемый результат также число. Например, `PERIOD_ADD(200712, 3)` возвращает значение 200803, поскольку через три месяца после декабря 2007 г. наступит март 2008 г.
- ❑ `PERIOD_DIFF(⟨Период в формате YYMM или YYYYMM⟩,⟨Период в формате YYMM или YYYYMM⟩)` — возвращает разность в месяцах между первым

и вторым периодами. Обратите внимание, что оба аргумента этой функции — числа. Например, `PERIOD_DIFF(200712, 200803)` возвращает значение `-3`.

- ❑ `TIMESTAMP('<Дата или дата время>', '<Время>')` — функция возвращает сумму своих аргументов в формате даты и времени. Например, функция `TIMESTAMP('2007-12-12 12:30', '15:50')` добавляет 15 часов 50 минут к 12 часам 30 минутам 12 декабря 2007 г. и возвращает результат `2007-12-13 04:20:00`.
- ❑ `TIMESTAMPADD(<Тип периода>, <Длина периода>, <Дата или дата и время>)` — возвращает дату или дату и время, сдвинутые относительно указанной даты на заданный период. Первым аргументом является тип периода:
 - `DAY` — число (номер дня в месяце);
 - `WEEK` — номер недели в году;
 - `MONTH` — номер месяца;
 - `QUARTER` — номер квартала;
 - `YEAR` — номер года;
 - `HOURL` — часы;
 - `MINUTE` — минуты;
 - `SECOND` — секунды.

Вторым аргументом является целое число — длина периода, то есть количество единиц измерения, заданных первым параметром. Если длина периода меньше 0, то дата, определяемая третьим параметром, будет сдвинута в прошлое.

Например, функция `TIMESTAMPADD(HOUR, 15, '2007-12-12 12:30:00')` добавляет 15 часов к 12 часам 30 минутам 12 декабря 2007 г. и возвращает результат `2007-12-13 03:30:00`.

- ❑ `TIMESTAMPDIFF(<Тип периода>, <Дата или дата и время>, <Дата или дата и время>)` — возвращает количество указанных периодов, прошедших между первым и вторым моментом. Первый аргумент может принимать те же значения, что и первый аргумент функции `TIMESTAMPADD()`. Если вторая дата предшествует первой, то результат будет отрицательным. Например, `TIMESTAMPDIFF(WEEK, '2007-12-12 12:30:00', '2007-12-31')` возвращает значение 2, и это означает, что 12 декабря и 31 декабря 2007 г. разделяют две недели.

В следующем подразделе мы рассмотрим функции, позволяющие переводить даты из одного формата в другой.

Функции преобразования форматов дат

Для преобразования дат из одного формата в другой вы можете использовать следующие основные функции.

□ `DATE_FORMAT ('<Дата или дата и время>', '<Формат>')` — возвращает строку, содержащую дату, преобразованную к указанному формату. Формат может включать следующие основные параметры.

- `%a` — сокращенное наименование дня недели (Sun, Mon и т. д.). О том, как настроить вывод дат на русском языке («Пнд», «Втр» и т. д.), рассказывалось при описании функции `DAYNAME()`.
- `%b` — сокращенное наименование месяца (Jan, Feb и т. д.). О том, как настроить вывод дат на русском языке («Янв», «Фев» и т. д.), рассказывалось при описании функции `DAYNAME()`.
- `%c` — номер месяца (0–12).
- `%D` — число (номер дня в месяце) с английским суффиксом (0th, 1st, 2nd, и т. д.).
- `%d` — число месяца (00–31).
- `%e` — число месяца (0–31).
- `%H` — часы (00–23).
- `%h`, `%I` — часы (01–12).
- `%i` — минуты (00–59).
- `%j` — номер дня в году (001–366).
- `%k` — часы (0–23).
- `%l` — часы (1–12).
- `%M` — наименование месяца (January, February и т. д.). О том, как настроить вывод дат на русском языке («Января», «Февраля» и т. д.), рассказывалось при описании функции `DAYNAME()`.
- `%m` — номер месяца (00–12).
- `%p` — АМ (обозначение первой половины суток) или РМ (обозначение второй половины суток).
- `%r` — время в 12-часовом формате (HH:MM:SS АМ или РМ).
- `%S`, `%s` — секунды (00–59).
- `%T` — время в 24-часовом формате (HH:MM:SS).

- %U — номер недели в году (00–53), первым днем недели считается воскресенье, первая неделя года — целиком находящаяся в этом году, для дней, предшествующих первой неделе, номер недели равен 0.
- %u — номер недели в году (00–53), первым днем недели считается понедельник, первая неделя года та, более трех дней которой находится в этом году, для дней, предшествующих первой неделе, номер недели равен 0.
- %V — номер недели в году (01–53), первым днем недели считается воскресенье, первая неделя года — целиком находящаяся в этом году, для дней, предшествующих первой неделе, номер недели равен номеру последней недели в предыдущем году.
- %v — номер недели в году (01–53), первым днем недели считается понедельник, первая неделя года та, более трех дней которой находится в этом году, для дней, предшествующих первой неделе, номер недели равен номеру последней недели в предыдущем году.
- %W — наименование дня недели (Sunday, Monday и т. д.). О том, как настроить вывод дат на русском языке («Понедельник», «Вторник» и т. д.), говорилось при описании функции DAYNAME () .
- %w — номер дня недели (0 — воскресенье, 1 — понедельник и т. д.).
- %X — номер года, к которому относится текущая неделя (первым днем недели считается воскресенье, первая неделя года — целиком находящаяся в этом году, неделя, предшествующая первой неделе года, относится к предыдущему году), в формате YYYY.
- %x — номер года, к которому относится текущая неделя (первым днем недели считается понедельник, первая неделя года та, более трех дней которой находится в этом году, неделя, предшествующая первой неделе года, относится к предыдущему году), в формате YYYY.
- %Y — номер года в формате YYYY.
- %y — номер года в формате YY.
- %% — знак процента.

Например, функция DATE_FORMAT('2007-12-12 12:30:00', '%e %M %Y г. %k часов %i минут') возвращает значение '12 December 2007 г. 12 часов 30 минут' или '12 Декабря 2007 г. 12 часов 30 минут', в зависимости от установленного языка вывода дат.

- ❑ TIME_FORMAT('<Время или дата и время>', '<Формат>') — функция возвращает время, преобразованное к указанному формату. Формат может

включать те из перечисленных выше параметров, которые предназначены для отображения часов, минут и секунд. Например, функция `TIME_FORMAT('2007-12-12 12:30:00', '%k часов %i минут')` возвращает значение '12 часов 30 минут'.

- ❑ `STR_TO_DATE(<Строка>, '<Формат>')` — получает в качестве аргумента строку, содержащую дату и/или время, и строку формата и возвращает дату и/или время, полученные из строки в соответствии с указанным форматом. Функции `STR_TO_DATE()` и `DATE_FORMAT()` взаимно обратны: если дата была преобразована в строку некоторого формата с помощью функции `DATE_FORMAT()`, то с помощью функции `STR_TO_DATE()`, указав тот же формат, можно получить исходную дату и наоборот. Например, `STR_TO_DATE('12 December 2007 г. 12 часов 30 минут', '%e %M %Y г. %k часов %i минут') = STR_TO_DATE(DATE_FORMAT('2007-12-12 12:30:00', '%e %M %Y г. %k часов %i минут'), '%e %M %Y г. %k часов %i минут') = 2007-12-12 12:30:00`. Однако названия месяцев и дней недели на русском языке функция `STR_TO_DATE()` обрабатывает некорректно.
- ❑ `GET_FORMAT(<DATE, TIME или DATETIME>, '<EUR', 'ISO', 'JIS', 'USA', или 'INTERNAL'>)` — возвращает строку формата даты и/или времени, которую затем можно использовать в функции `DATE_FORMAT()`. Первый аргумент функции указывает, какой формат нужно получить: формат даты, формат времени или формат даты и времени. Второй аргумент задает стандарт, которому соответствует возвращаемый формат. Например, функция `GET_FORMAT(DATETIME, 'EUR')` возвращает значение `%Y-%m-%d %H.%i.%s`.
- ❑ `MAKEDATE(<Номер года>, <Номер дня в году>)` — функция получает в качестве аргументов номер года и номер дня в году и возвращает соответствующую дату. Например, функция `MAKEDATE(2007, 346)` возвращает значение `2007-12-12`.
- ❑ `MAKETIME(<Часы>, <Минуты>, <Секунды>)` — получает в качестве аргументов час, минуту и секунду и возвращает соответствующее время. Например, функция `MAKETIME(12, 30, 0)` возвращает значение `12:30:00`.
- ❑ `FROM_DAYS(<Количество дней>)` — функция получает в качестве аргумента количество дней от Р. Х. и возвращает дату, соответствующую этому дню. Так, функция `FROM_DAYS(733387)` возвращает значение `2007-12-12`.
- ❑ `TO_DAYS(<Дата или дата и время>)` — получает в качестве аргумента дату или дату и время и возвращает количество дней от Р. Х., соответствующее этой

дате. Например, функция `TO_DAYS('2007-12-12 12:30:00')` возвращает значение 733387.

- ❑ `SEC_TO_TIME(<Количество секунд>)` — функция получает в качестве аргумента количество секунд и возвращает соответствующее количество часов, минут и секунд в формате времени. Например, функция `SEC_TO_TIME(45000)` возвращает значение 12:30:00.
- ❑ `TIME_TO_SEC(<Время или дата и время>)` — получает в качестве аргумента время или дату и время и возвращает количество секунд, соответствующее времени (дата при этом игнорируется). Например, функция `TIME_TO_SEC('2007-12-12 12:30:00')` возвращает значение 45000.
- ❑ `FROM_UNIXTIME(<Unix-время>[, '<Формат>'])` — функция получает в качестве аргумента UNIX-время — количество секунд, прошедших с 1 января 1970 г., и возвращает соответствующую дату и время. Например, функция `FROM_UNIXTIME(1197451800)` возвращает значение 2007-12-12 12:30:00.

При необходимости вы можете задать формат возвращаемой даты и/или времени, используя параметры, которые мы перечислили при описании функции `DATE_FORMAT()`.

- ❑ `UNIX_TIMESTAMP(<Дата>)` — если аргумент функции `UNIX_TIMESTAMP()` не задан, то она возвращает текущее UNIX-время. Если задан аргумент — дата, дата и время либо число в формате `YYYYMMDD`, `YYMMDD`, `YYYYMMDDHHMMSS` или `YYMMDDHHMMSS`, то функция `UNIX_TIMESTAMP()` возвращает UNIX-время, соответствующее указанной дате. Например, функция `UNIX_TIMESTAMP(20071212123000)` возвращает значение 1197451800.
- ❑ `TIMESTAMP('<Дата или дата время>')` — если задан только один аргумент функции `TIMESTAMP()`, то она возвращает этот аргумент, преобразованный в формат даты и времени. Например, `TIMESTAMP('2007-12-12')` возвращает значение 2007-12-12 00:00:00.

Итак, мы изучили функции, выполняющие операции с датами и временем. В следующем разделе мы рассмотрим некоторые функции, работающие с символьными значениями.

4.5. Символьные функции

В выражениях вы можете использовать символьные функции, описанные ниже.

- ❑ `BIT_LENGTH ('<Строка>')` — возвращает длину строки в битах. Например, функция `BIT_LENGTH ('Крылов')` возвращает значение 48 при использовании однобайтовой кодировки и значение 96 при использовании кодировки UTF-8.
- ❑ `CHAR_LENGTH ('<Строка>')`, `CHARACTER_LENGTH ('<Строка>')` — возвращают количество символов в строке. Например, функция `CHAR_LENGTH ('Крылов')` возвращает значение 6.
- ❑ `LENGTH ('<Строка>')`, `OCTET_LENGTH ('<Строка>')` — функции возвращают длину строки в байтах. Например, функция `LENGTH ('Крылов')` возвращает значение 6 при использовании однобайтовой кодировки и значение 12 при использовании кодировки UTF-8.
- ❑ `CHAR (<Код 1>, <Код 2>, ..., <Код N> [USING <Кодировка>])` — получает в качестве аргументов коды символов и возвращает строку, состоящую из этих символов. При необходимости можно явно указать кодировку, сопоставляющую коды с символами. Например, функции `CHAR (53402, 53632, 53643, 53435, 53438, 53426 USING utf8)` и `CHAR (138, 224, 235, 171, 174, 162 USING cp866)` возвращают значение 'Крылов'.
- ❑ `ORD ('<Строка>')` — возвращает числовой код первого символа в строке. Например, функция `ORD (CHAR (53402, 53632, 53643, 53435, 53438, 53426 USING utf8))` возвращает значение 53402.
- ❑ `CONCAT ('<Строка 1>', '<Строка 2>', ..., '<Строка N>')` — функция возвращает результат объединения своих аргументов. Например, функция `CONCAT ('ООО "Кускус"', 'Петров', 'Крылов')` возвращает значение 'ООО "Кускус"ПетровКрылов'.
- ❑ `CONCAT_WS ('Разделитель', '<Строка 1>', '<Строка 2>', ..., '<Строка N>')` — возвращает результат объединения своих аргументов, при этом первый аргумент используется как разделитель. Например, функция `CONCAT_WS ('', 'ООО "Кускус"', 'Петров', 'Крылов')` возвращает значение 'ООО "Кускус", Петров, Крылов'.
- ❑ `REPEAT ('<Строка>', 'Количество экземпляров, не менее 1')` — функция возвращает строку, в которую исходная строка входит указанное количество раз. Например, функция `REPEAT ('Трижды', 3)` возвращает значение 'ТриждыТриждыТрижды'.
- ❑ `REVERSE ('<Строка>')` — возвращает строку, в которой символы исходной строки расположены в обратном порядке. Например, функция `REVERSE ('наоборот')` возвращает значение 'торобоан'.

- ❑ `SPACE (<Количество пробелов>)` — функция возвращает строку, состоящую из указанного количества пробелов. Например, функция `SPACE (1)` возвращает значение ' '.
- ❑ `ELT (k, '<Строка 1>', '<Строка 2>', ..., '<Строка N>')` — возвращает строку с порядковым номером k . Например, функция `ELT (2, '000 "Кускус"', 'Петров', 'Крылов')` возвращает значение 'Петров'.
- ❑ `FIELD ('<Строка-образец>', '<Строка 1>', '<Строка 2>', ..., '<Строка N>')` — функция возвращает порядковый номер строки, совпадающей с образцом, и 0, если ни одна из строк 1, 2... N не совпадает с образцом. Например, функция `FIELD ('Петров', '000 "Кускус"', 'Петров', 'Крылов')` возвращает значение 2.
- ❑ `FIND_IN_SET ('<Строка-образец>', '<Строка-контейнер>')` — эта функция получает в качестве аргумента строку-образец (эта строка не должна содержать запятых) и строку-контейнер вида '`<Подстрока 1>, <Подстрока 2>, ..., <Подстрока N>`' и возвращает порядковый номер подстроки, совпадающей с образцом, и 0, если ни одна из подстрок 1, 2... N не совпадает с образцом или строка-контейнер пустая. Например, функция `FIND_IN_SET ('Петров', '000 "Кускус", Петров, Крылов')` возвращает значение 2.
- ❑ `EXPORT_SET (<Число>, '<Подстрока для бита 1>', '<Подстрока для бита 0>' [, '<Разделитель>' [, <Количество бит>]])` — преобразует число в строку, заменяя каждый бит (0 или 1) соответствующей подстрокой, и возвращает полученную строку. Биты рассматриваются в обратном порядке, то есть справа налево. При необходимости можно задать разделитель подстрок (если разделитель не задан, используется запятая), а также максимальное количество битов, которые будут обработаны (недостающие биты рассматриваются как нулевые, «лишние» биты игнорируются, а если количество битов не задано, используется значение 64). Например, функция `EXPORT_SET (25, 'Да', 'Нет', ':', 7)` возвращает значение 'Да:Нет:Нет:Да:Да:Нет:Нет', поскольку 25 записывается в двоичной системе счисления как 11001.
- ❑ `MAKE_SET (<Число>, '<Подстрока 0>', '<Подстрока 1>', ..., '<Подстрока N>')` — функция преобразует число в строку: подстрока с порядковым номером k добавляется в строку, если k -й бит равен 1. Биты рассматриваются в обратном порядке, то есть справа налево. Разделителем подстрок служит запятая. Например, функция `MAKE_SET (6, 'Я согласен получать новости`

компании', 'Я согласен участвовать в опросах', 'Я согласен участвовать в тестировании продукта') возвращает значение 'Я согласен участвовать в опросах, Я согласен участвовать в тестировании продукта', поскольку 6 записывается в двоичной системе счисления как 110.

- ❑ `INSERT(<Строка>', <Позиция>', <Длина подстроки>', <Замещающая подстрока>')` — возвращает строку, в которой подстрока, начинающаяся с указанной позиции и состоящая из указанного количества символов, заменена заданной подстрокой. Например, функция `INSERT('ООО "Кускус"', 6, 3, 'Кискис')` заменяет подстроку 'Кус' строки 'ООО "Кускус"' подстрокой 'Кискис' и возвращает значение 'ООО "Кискискус"'.
- ❑ `REPLACE(<Строка>', <Замещаемая подстрока>', <Замещающая подстрока>')` — возвращает строку, в которой вместо замещаемой подстроки подставлена замещающая. Например, функция `REPLACE('Не имей сто рублей, а имей сто друзей', 'сто', 'тысячу')` возвращает значение 'Не имей тысячу рублей, а имей тысячу друзей'.
- ❑ `SUBSTR(<Строка>', <Позиция>[, <Длина>])`, `SUBSTRING(<Строка>', <Позиция>[, <Длина>])`, `SUBSTR(<Строка>' FROM <Позиция>[FOR <Длина>])`, `SUBSTRING(<Строка>' FROM <Позиция>[FOR <Длина>])` — возвращают подстроку исходной строки, начинающуюся с указанной позиции. При необходимости можно указать длину получаемой подстроки. Если номер позиции меньше 0, то позиция отсчитывается не от начала строки, а от конца. Например:
 - функция `SUBSTR('Семь чудес света', 6)` возвращает значение 'чудес света';
 - функция `SUBSTR('Семь чудес света', 6, 5)` возвращает значение 'чудес';
 - функция `SUBSTR('Семь чудес света', -5)` возвращает значение 'света'.
- ❑ `MID(<Строка>', <Позиция>', <Длина>)` — синоним функции `SUBSTRING(<Строка>', <Позиция>', <Длина>)`.
- ❑ `SUBSTRING_INDEX(<Строка>', <Подстрока>', <Порядковый номер вхождения>)` — если заданный порядковый номер вхождения больше 0, то функция `SUBSTRING_INDEX()` находит в исходной строке вхождение указанной подстроки с этим порядковым номером (ведя отсчет от начала строки)

и возвращает часть исходной строки, предшествующую этому вхождению. Если же заданный порядковый номер вхождения меньше 0, то вхождения отсчитываются от конца строки и возвращается часть исходной строки, которая следует за этим вхождением.

Например, функция `SUBSTRING_INDEX('Семь чудес света', ' ', 2)` возвращает значение `'Семь чудес'` (подстроку, предшествующую второму пробелу), а функция `SUBSTRING_INDEX('Семь чудес света', ' ', -2)` возвращает значение `'чудес света'` (подстроку, следующую за вторым пробелом).

- ❑ `LEFT('<Строка>', <Длина подстроки>)` — возвращает начальную подстроку исходной строки, состоящую из указанного количества символов. Например, функция `LEFT('Генератор', 3)` возвращает значение `'Ген'`.
- ❑ `RIGHT('<Строка>', <Длина подстроки>)` — функция возвращает подстроку, состоящую из указанного количества последних символов исходной строки. Например, функция `RIGHT('Генератор', 3)` возвращает значение `'тор'`.
- ❑ `LOCATE('<Подстрока>', '<Строка>' [, <Позиция>])` — возвращает позицию, с которой начинается первое вхождение подстроки в строку, или 0, если строка не содержит такой подстроки. При необходимости можно указать позицию в исходной строке, начиная с которой нужно искать вхождение подстроки. Например, функция `LOCATE('сто', 'Не имей сто рублей, а имей сто друзей')` возвращает значение 9, а функция `LOCATE('сто', 'Не имей сто рублей, а имей сто друзей', 20)` возвращает значение 28. Регистр символов учитывается только в том случае, если хотя бы одна из строк байтовая (бинарная).
- ❑ `INSTR('<Строка>', '<Подстрока>'), POSITION('<Подстрока>' IN '<Строка>')` — синонимы функции `LOCATE('<Подстрока>', '<Строка>')`. Обратите внимание, что порядок аргументов у функций `INSTR` и `LOCATE` разный.
- ❑ `LCASE('<Строка>'), LOWER('<Строка>')` — возвращают строку, приведенную к нижнему регистру. Например, функция `LCASE('Крылов')` возвращает значение `'крылов'`.
- ❑ `UCASE('<Строка>'), UPPER('<Строка>')` — возвращают строку, приведенную к верхнему регистру. Например, функция `UCASE('Крылов')` возвращает значение `'КРЫЛОВ'`.

- ❑ `LOAD_FILE('<Путь и имя файла>')` — получает в качестве аргумента полный путь и имя файла, расположенного на компьютере, где работает сервер MySQL, и возвращает в виде строки данные, содержащиеся в этом файле. В пути к файлу необходимо использовать прямую косую черту вместо принятой в Windows обратной косой черты.
- ❑ `LPAD('<Строка>', <Длина>, '<Символы заполнения>')` — возвращает строку указанной длины, полученную из исходной строки путем добавления символов заполнения в начале строки (если количество символов в исходной строке меньше указанного) или отбрасывания «лишних» символов (если количество символов в исходной строке больше указанного). Например, функция `LPAD('Крылов', 9, '+-')` возвращает значение `'++Крылов'`, а функция `LPAD('ООО "Кускус"', 9, '+-')` — значение `'ООО "Куск"'`.
- ❑ `RPAD('<Строка>', <Длина>, '<Символы заполнения>')` — аналогична функции `LPAD()`, только символы заполнения добавляются в конце строки. Например, функция `RPAD('Крылов', 9, '+-')` возвращает значение `'Крылов++'`, а функция `RPAD('ООО "Кускус"', 9, '+-')` возвращает значение `'ООО "Куск"'`.
- ❑ `LTRIM('<Строка>')` — возвращает строку, полученную из исходной путем удаления начальных пробелов. Например, функция `LTRIM(' Крылов ')` возвращает значение `'Крылов '`.
- ❑ `RTRIM('<Строка>')` — функция возвращает строку, полученную из исходной путем удаления пробелов в конце строки. Например, функция `RTRIM(' Крылов ')` возвращает значение `' Крылов'`.
- ❑ `TRIM([[LEADING, TRAILING или BOTH] ['<Символы заполнения>'] FROM] '<Строка>')` — возвращает строку, полученную из исходной путем удаления начальных пробелов и пробелов в конце строки. Если необходимо удалить другой символ или последовательность символов, их нужно указать в параметре `<Символы заполнения>`. Если требуется удалить символы заполнения только в начале строки, укажем параметр `LEADING`, если только в конце — параметр `TRAILING`. Например, функция `TRIM('+-' FROM '++Крылов-+-')` возвращает значение `'+Крылов-`'.
- ❑ `FORMAT(<Число>, <Количество цифр после запятой>)` — округляет число до заданного количества цифр после десятичного разделителя или, наоборот, дополняет число нулями справа до заданного количества цифр после десятичного разделителя и возвращает это число в виде строки, используя запятую как разделитель тысяч. Например, функция `FORMAT(12345.6789, 2)` возвращает строку `'12,345.68'`.

Итак, мы рассмотрели функции, оперирующие символьными значениями. Подведем теперь итоги главы.

4.6. Резюме

В этой главе вы научились применять операторы и функции для поиска нужных данных в таблицах, а также для вычислений на основе этих данных, в том числе для получения обобщающей информации из таблиц. Из следующей главы вы узнаете, как использовать базу данных MySQL в различных веб-приложениях: подключать веб-приложения к базе данных, передавать данные из базы в веб-приложения и сохранять в базе данные из веб-приложений.

Глава 5

Работа с базами данных и их администрирование из веб-приложений

Эта глава посвящена использованию базы данных MySQL в веб-приложениях, написанных на языке PHP. В первую очередь мы поговорим о совместной работе MySQL и приложений PHP.

5.1. Интерфейс с PHP

В этом разделе мы рассмотрим процесс создания веб-приложения, взаимодействующего с базой данных MySQL. Для иллюстрации работы со сценариями PHP будем использовать пакет XAMPP, который обсуждали в главе 1. Надеюсь, что вы его успешно установили и готовы к созданию PHP-приложений. Об этом и пойдет речь в следующем разделе.

Подготовительные действия

Прежде чем выполнять операции с данными в базе, необходимо подключиться к работающему серверу MySQL. Подключение выполняется с помощью функции:

```
mysql_connect("<Имя хоста>",  
    "<Имя пользователя>", "<Пароль>");
```

Эта функция возвращает *указатель на соединение* либо значение `false`, если установить соединение не удалось.

В качестве примера рассмотрим PHP-приложение, которое подключается к серверу MySQL и выводит диагностическое сообщение. Создадим в папке `htdocs` корневой папки XAMPP файл `output.php`, содержащий следующий код (листинг 5.1).

Листинг 5.1. Подключение к серверу MySQL

```
<html>  
<head>  
<title>Работа с MySQL</title>  
</head>  
<body>  
<h1>Подключение к базе данных</h1>  
<?php  
//Соединяемся с сервером MySQL  
$connection = mysql_connect("localhost","username","userpassword");  
if(!$connection) die("Ошибка доступа к базе данных.  
    Приносим свои извинения");  
print("Подключение выполнено успешно");  
?>  
</body>
```

Для запуска этого приложения наберем в адресной строке браузера адрес `http://localhost/output.php`. При открытии этой страницы приложение осуществляет подключение к серверу с использованием имени пользователя `username` и паролем `userpassword`; возвращаемый функцией указатель на соединение сохраняется в переменной `$connection`. При успешном подключении на веб-странице появится соответствующее сообщение (рис. 5.1).

Если подключиться к серверу не удалось, то на веб-странице появится сообщение **Ошибка доступа к базе данных**. В этом случае необходимо убедиться, что сервер MySQL запущен (например, с помощью утилиты `phpMyAdmin`, см. главу 1), а также проверить правильность написания имени пользователя и пароля.

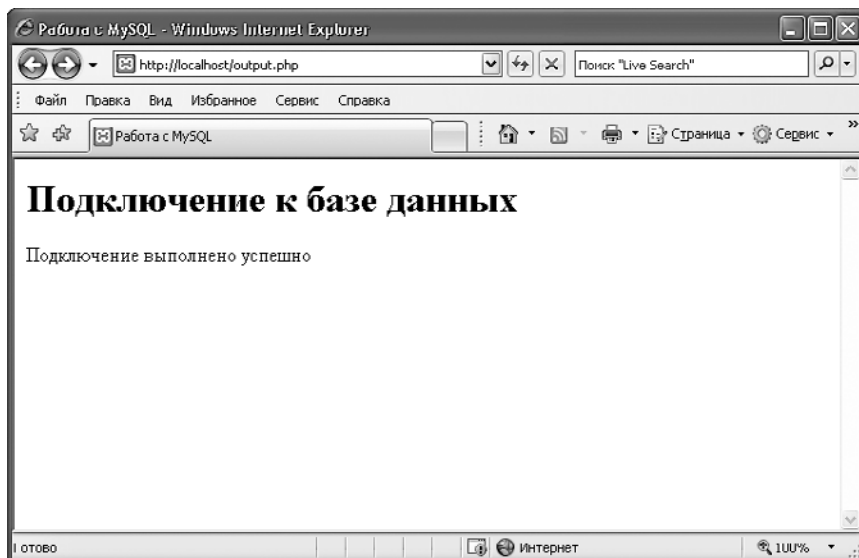


Рис. 5.1. Результат подключения к серверу MySQL

СОВЕТ



Для защиты от несанкционированного доступа при подключении к базе данных рекомендуется указывать не пользователя root, а специально созданного пользователя с минимально необходимыми правами доступа. О регистрации пользователей и настройке прав мы подробно расскажем чуть ниже.

Следующий шаг — выбор текущей базы данных с помощью функции:

```
mysql_select_db("<Имя базы данных>"[,  
    <Указатель на соединение>]);
```

ПРИМЕЧАНИЕ



Для этой функции, а также для всех остальных функций, описанных далее в этом разделе, указатель на соединение является необязательным параметром. Если этот параметр не указан, то подразумевается последнее открытое соединение. Таким образом, если ваше приложение использует только одно соединение с базой данных, то задавать этот параметр нет необходимости.

Функция `mysql_select_db()` аналогична команде `USE <Имя базы данных>`, о которой рассказывалось в главе 3. Добавим в сценарий `output.php` вызов этой функции (листинг 5.2).

Листинг 5.2. Выбор текущей базы данных

```
<html>  
<head>
```

```
<title>Работа с MySQL</title>
</head>
<body>
<h1>Подключение к базе данных</h1>
<?php
//Соединяемся с сервером MySQL
$connection = mysql_connect("localhost","username","userpassword");
if(!$connection) die("Ошибка доступа к базе данных.
    Приносим свои извинения");
//Выбираем базу данных SalesDept (Отдел продаж)
if(!mysql_select_db("SalesDept"))
    die("База данных отсутствует. Приносим свои извинения");
print("База данных выбрана успешно");
?>
</body>
```

После обновления страницы <http://localhost/output.php> вы увидите сообщение Операция выполнена успешно либо База данных отсутствует. Приносим свои извинения. В последнем случае необходимо проверить, существует ли база данных с таким именем на сервере MySQL и есть ли у пользователя `username` право доступа к этой базе.

После подключения к серверу MySQL и выбора текущей базы данных можно приступать к работе с данными. И в первую очередь необходимо установить кодировку, чтобы избежать проблем с отображением символов русского алфавита. Как вы знаете из главы 3, указать серверу кодировку, которую использует клиентское приложение, можно с помощью команды `SET NAMES <Кодировка>`; . Поскольку файл `output.php` мы сохранили в кодировке Windows (CP-1251), именно ее и нужно будет установить.

Для отправки SQL-команды на сервер MySQL используется функция:

```
mysql_query("<Текст команды>"[, <Указатель на соединение>]);
```

Добавим в сценарий `output.php` вызов этой функции (листинг 5.3).

Листинг 5.3. Установка кодировки

```
<html>
<head>
<title>Работа с MySQL</title>
```



```
</head>
<body>
<h1>Подключение к базе данных</h1>
<?php
//Соединяемся с сервером MySQL
$connection = mysql_connect("localhost","username","userpassword");
if(!$connection) die("Ошибка доступа к базе данных.
    Приносим свои извинения");
//Выбираем базу данных SalesDept (Отдел продаж)
if(!mysql_select_db("SalesDept"))
    die("База данных отсутствует. Приносим свои извинения");
//Устанавливаем кодировку CP-1251
mysql_query("SET NAMES cp1251");
print("Операция выполнена успешно");
?>
</body>
```

В следующем подразделе мы поговорим о том, как получить из базы данных необходимую информацию.

Выполнение запроса к базе данных

В этом подразделе вы узнаете, как создать PHP-сценарий, формирующий динамическую веб-страницу на основе данных, полученных из базы.

Как я упоминал выше, для выполнения SQL-команды PHP-приложением предназначена функция:

```
mysql_query("<Текст команды>"[, <Указатель на соединение>])
```

Если SQL-команда предполагает получение информации из базы данных, то функция `mysql_query()` возвращает указатель на полученный массив данных (либо значение `false`, если при выполнении запроса произошла ошибка).

После того как запрос выполнен, извлечь из полученного массива конкретные значения можно с помощью функции:

```
mysql_fetch_assoc(<Указатель на результат запроса>)
```

или

```
mysql_fetch_array(<Указатель на результат запроса>)
```

Функция `mysql_fetch_assoc()` получает очередную строку из результата запроса и возвращает ее в виде ассоциативного массива. Иными словами, вы получаете возможность работать со значениями в строке, используя для доступа к ним названия столбцов.

Проиллюстрирую работу этой функции на примере вывода списка товаров, то есть данных из таблицы `Products`. Добавим в сценарий `output.php` вызов функций `mysql_query()` и `mysql_fetch_assoc()` (листинг 5.4).

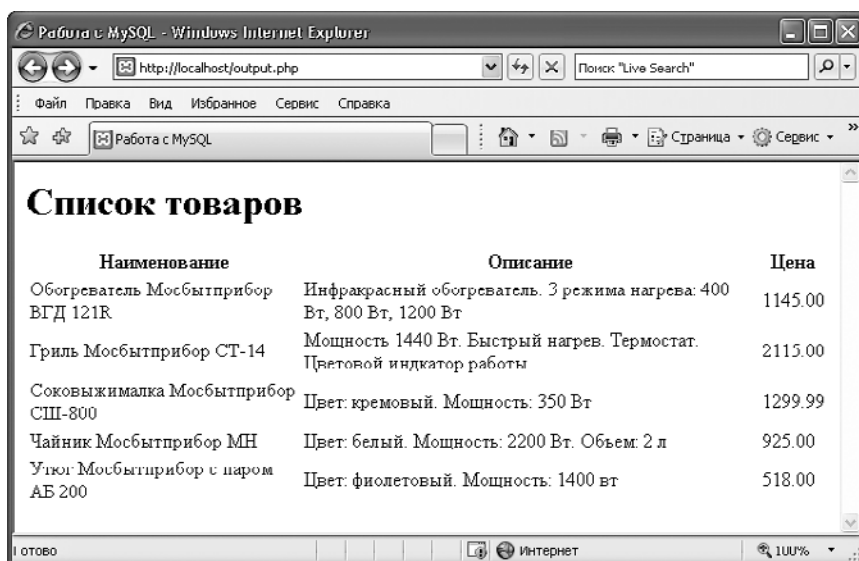
Листинг 5.4. Получение информации и отображение ее на странице

```
<html>
<head>
<title>Работа с MySQL</title>
</head>
<body>
<h1>Список товаров</h1>
<!-- Выводим список товаров -->
  <table>
<!-- Выводим заголовок списка товаров -->
    <tr>
      <th>Наименование</th>
      <th>Описание</th>
      <th>Цена</th>
    </tr>
<?php
//Соединяемся с сервером MySQL
$connection = mysql_connect("localhost","username","userpassword");
if(!$connection) die("Ошибка доступа к базе данных.
    Приносим свои извинения");
//Выбираем базу данных SalesDept (Отдел продаж)
if(!mysql_select_db("SalesDept"))
    die("База данных отсутствует. Приносим свои извинения");
//Устанавливаем кодировку CP-1251
mysql_query("SET NAMES cp1251");
//Получаем список товаров
$qresult = mysql_query("SELECT * FROM Products");
if(!$qresult) die("Ошибка доступа к базе данных.
    Приносим свои извинения");
```

```
//Очередную строку из результата запроса (информацию о товаре)
// записываем в ассоциативный массив $product
while ($product=mysql_fetch_assoc($qresult))
{
//выводим элементы массива $product с именами description (наименование),
//details (описание) и price (цена)
print "\n<tr><td>{$product["description"]}</td>
<td>{$product["details"]}</td>
<td>{$product["price"]}</td></tr>\n";
}
?>
</table>
</body>
```

В этом примере функция `mysql_query()` выполняет запрос `SELECT * FROM Products;`, а функция `mysql_fetch_assoc()` в цикле `while` преобразует каждую из строк, полученных запросом, в массив `$product`. Затем мы извлекаем элементы массива `$product` с именами `description`, `details` и `price`, что соответствует значениям в столбцах `description`, `details` и `price` таблицы `Products`.

После обновления страницы `http://localhost/output.php` вы увидите следующий результат (рис. 5.2).



Наименование	Описание	Цена
Обогреватель Мосбытприбор ВГД 121R	Инфракрасный обогреватель. 3 режима нагрева: 400 Вт, 800 Вт, 1200 Вт	1145.00
Гриль Мосбытприбор СТ-14	Мощность 1440 Вт. Быстрый нагрев. Термостат. Цветовой индикатор работы	2115.00
Соковыжималка Мосбытприбор СШ-800	Цвет: кремовый. Мощность: 350 Вт	1299.99
Чайник Мосбытприбор МН	Цвет: белый. Мощность: 2200 Вт. Объем: 2 л	925.00
Утюг Мосбытприбор с паром АБ 200	Цвет: фиолетовый. Мощность: 1400 Вт	518.00

Рис. 5.2. Отображение на странице полученной информации

Полезно также изучить HTML-код этой страницы (в Internet Explorer для этого нужно нажать кнопку Страница и в появившемся меню выбрать пункт Просмотр HTML-кода). Вы увидите результат работы цикла while и, в частности, значения элементов массива (рис. 5.3).

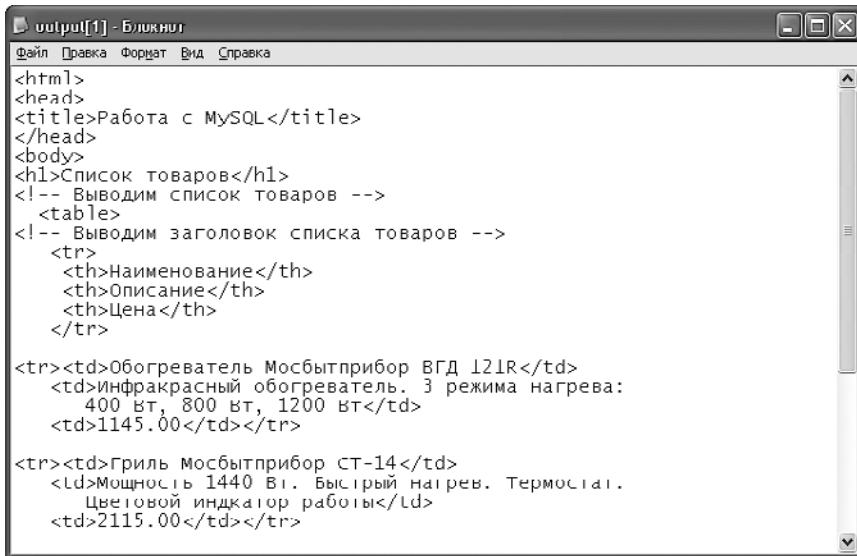


Рис. 5.3. HTML-код страницы

В некоторых случаях вместо названий элементов массива удобнее использовать числовые индексы. Например, если в результате запроса присутствуют несколько столбцов с одинаковыми именами (это возможно при получении информации из нескольких таблиц), то функция `mysql_fetch_assoc()` предоставляет доступ только к последнему из этих столбцов. Избежать этого позволяет функция `mysql_fetch_array()`. Она возвращает как ассоциативный массив, так и массив с числовыми индексами. Таким образом, вы можете обращаться к значению, хранящемуся в строке, используя либо имя столбца, либо порядковый номер столбца в результате запроса. Например, код, представленный в листинге 5.4, можно переписать следующим образом (листинг 5.5).

Листинг 5.5. Получение информации и отображение ее на странице

```

<html>
<head>
<title>Работа с MySQL</title>
</head>
<body>

```

```
<h1>Список товаров</h1>
<!-- Выводим список товаров -->
<table>
<!-- Выводим заголовок списка товаров -->
  <tr>
    <th>Наименование</th>
    <th>Описание</th>
    <th>Цена</th>
  </tr>
<?php
//Соединяемся с сервером MySQL
$conn = mysql_connect("localhost","username","userpassword");
if(!$conn) die("Ошибка доступа к базе данных.
    Приносим свои извинения");
//Выбираем базу данных SalesDept (Отдел продаж)
if(!mysql_select_db("SalesDept"))
    die("База данных отсутствует. Приносим свои извинения");
//Устанавливаем кодировку CP-1251
mysql_query("SET NAMES cp1251");
//Получаем список товаров
$qresult = mysql_query("SELECT * FROM Products");
if(!$qresult) die("Ошибка доступа к базе данных.
    Приносим свои извинения");
//Очередную строку из результата запроса (информацию о товаре)
// записываем в массив $product
while ($product=mysql_fetch_array($qresult))
{
//выводим элементы массива $product с номерами 1, 2, 3
    print "\n<tr><td>{$product[1]}</td>
        <td>{$product[2]}</td>
        <td>{$product[3]}</td></tr>\n";
}
?>
</table>
</body>
```

Результат выполнения приложения при этом не изменится. Итак, мы рассмотрели функции, обеспечивающие получение данных из базы и работу с этими данными. Однако при выполнении запросов к базе данных возможно возникновение ошибок. Чтобы устранить эти ошибки, необходимо иметь подробную информацию о них. В следующем подразделе мы остановимся на этом вопросе подробнее.

Обработка ошибок

Для получения сведений о возникшей ошибке взаимодействия с базой данных предназначены функции:

```
mysql_error([<Указатель на соединение>])
```

и

```
mysql_errno([<Указатель на соединение>])
```

Функция `mysql_error()` возвращает описание ошибки, произошедшей при выполнении последней SQL-команды (или пустую строку, если команда была выполнена успешно). Функция `mysql_errno()` возвращает код ошибки, произошедшей при выполнении последней SQL-команды (или 0, если команда была выполнена успешно).

Значения, возвращаемые функциями `mysql_error()` и `mysql_errno()`, как и системные сообщения об ошибках, нежелательно отображать на веб-странице, чтобы не раскрывать информацию об архитектуре приложения. Вместо этого рекомендуется записывать сведения об ошибке в файл или отправлять по электронной почте разработчику или администратору приложения.

Чтобы отключить вывод системных сообщений об ошибках, добавим в код приложения вызов функции:

```
error_reporting(0)
```

При возникновении ошибки сформируем собственное сообщение, содержащее дату и время, номер и описание ошибки. Записать это сообщение в log-файл или отправить по электронной почте позволяет функция:

```
error_log("<Текст сообщения>",  
        "<Тип сообщения>", "<Адрес доставки>")
```

Дополним сценарий `output.php` обработкой ошибок (листинг 5.6).

Листинг 5.6. Обработка ошибок

```
<html>
<head>
<title>Работа с MySQL</title>
</head>
<body>
<h1>Список товаров</h1>
<!-- Выводим список товаров -->
  <table>
<!-- Выводим заголовок списка товаров -->
    <tr>
      <th>Наименование</th>
      <th>Описание</th>
      <th>Цена</th>
    </tr>
<?php
//Отключаем вывод системных сообщений об ошибках
error_reporting(0);
//Соединяемся с сервером MySQL
$connection = mysql_connect("localhost","username","userpassword");
if(!$connection) die("Ошибка доступа к базе данных.
Приносим свои извинения");
//Выбираем базу данных SalesDept (Отдел продаж)
//В случае ошибки формируем сообщение, записываем его в файл
//и отправляем по электронной почте
if(!mysql_select_db("SalesDept"))
{
  $err_message=date("Y.m.d H:i:s")."
  ".mysql_errno()." ".mysql_error()."\r\n";
  error_log($err_message,3,"/mysqlerror.log");
  error_log($err_message,1,"admin@somedomain.ru");
  die("Ошибка доступа к базе данных. Приносим свои извинения");
}
//Устанавливаем кодировку CP-1251
mysql_query("SET NAMES cp1251");
```

```
//Получаем список товаров
$qresult = mysql_query("SELECT * FROM Products");
//Проверяем результат выполнения запроса; в случае ошибки формируем
//сообщение, записываем его в файл и отправляем по электронной почте
if(!$qresult)
{
    $err_message=date("Y.m.d H:i:s")."
    ".mysql_errno()." ".mysql_error()."\r\n";
    error_log($err_message,3,"/mysqlerror.log");
    error_log($err_message,1,"admin@somedomain.ru");
    die("Ошибка доступа к базе данных. Приносим свои извинения");
}
//Очередную строку из результата запроса (информацию о товаре)
// записываем в ассоциативный массив $product
while ($product=mysql_fetch_assoc($qresult))
{
    //выводим элементы массива $product с именами description (наименование),
    //details (описание) и price (цена)
    print "\n<tr><td>{$product["description"]}</td>
    <td>{$product["details"]}</td>
    <td>{$product["price"]}</td></tr>\n";
}
?>
</table>
</body>
```

Если при выполнении запроса произойдет ошибка, например, окажется, что таблица `Products` была удалена, то сообщение вида `2008.06.15 14:22:53 1146 Table 'salesdept.products' doesn't exist` будет записано в файл `mysqlerror.log`, находящийся в папке `htdocs` корневой папки `XAMPP`, и отправлено на адрес `admin@somedomain.ru` (тип сообщения 3 соответствует записи в файл, тип 1 — отправке по электронной почте). На веб-странице при этом отобразится нейтральное сообщение: `Ошибка доступа к базе данных. Приносим свои извинения.`

Итак, мы завершили создание приложения, которое получает информацию из базы данных. В следующем подразделе мы рассмотрим обратный пример — приложение, которое записывает в базу данные, введенные пользователем на веб-странице.

Ввод данных в базу

В этом подразделе мы поговорим о том, как создать PHP-приложение для ввода данных в базу. Такие приложения обычно состоят из двух взаимосвязанных страниц. Первая страница представляет собой форму, в которую пользователь может ввести данные. Вторая — собственно PHP-сценарий, обрабатывающий эти данные.

В качестве примера рассмотрим форму саморегистрации нового клиента, где клиент указывает свои имя, телефон и адрес. Создадим в папке `htdocs` корневой папки XAMPP файл `input.php`, содержащий следующий код (листинг 5.7).

Листинг 5.7. Форма ввода данных

```
<html>
<head>
  <title>Работа с MySQL</title>
</head>
<body>
<h1>Пожалуйста, заполните следующие поля:</h1>
<!-- Создаем форму для ввода данных -->
<!-- Обрабатывать введенные данные будет сценарий save.php -->
  <form method="post" action="save.php">
    <table>
<!-- Создаем поле для ввода имени заказчика -->
      <tr>
        <td>Ваше имя:</td>
        <td><input type="text" name="CustomerName" value=""></td>
      </tr>
<!-- Создаем поле для ввода телефона заказчика -->
      <tr>
        <td>Телефон:</td>
        <td><input type="text" name="CustomerPhone" value="(495)"></td>
      </tr>
<!-- Создаем поле для ввода адреса заказчика -->
      <tr>
        <td>Адрес:</td>
        <td><input type="text" name="CustomerAddress" value=""></td>
      </tr>
    </table>
  </form>
</body>
</html>
```

```
</tr>
</table>
<br>
<!-- Создаем кнопку для подтверждения данных -->
<input type="submit" value="Отправить">
</form>
</body>
</html>
```

На рис. 5.4 показана веб-страница, сгенерированная этим кодом.

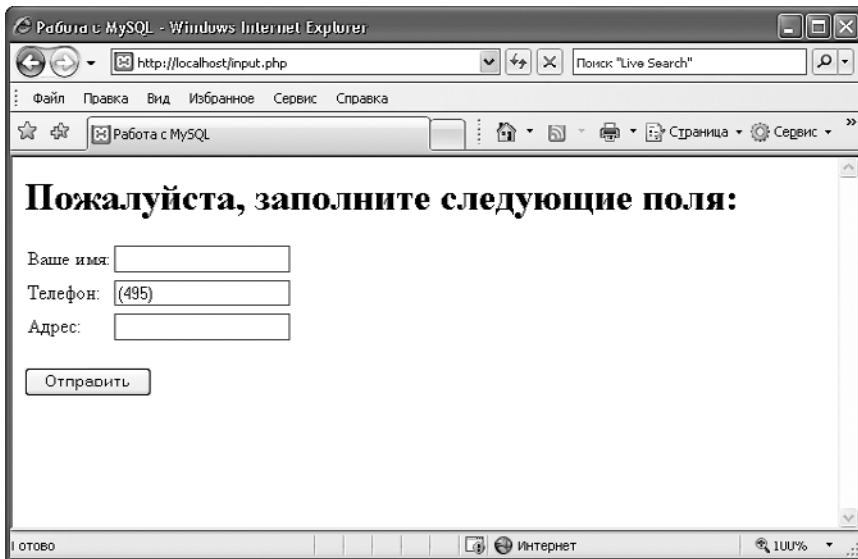


Рис. 5.4. Форма ввода данных

Создавая эту форму, мы указали, что при ее подтверждении (то есть при нажатии кнопки **Отправить**) введенные пользователем значения полей будут переданы в сценарий `save.php` по методу POST. В сценарии `save.php` мы будем использовать уже известные PHP-функции:

- ❑ функции подключения к серверу MySQL и выбора базы данных, которые мы рассматривали выше в этой главе;
- ❑ функцию `mysql_query()`, которая обеспечивает выполнение SQL-команды на сервере MySQL. Если SQL-команда не предполагает получение данных из базы (такими командами являются, например, INSERT, UPDATE, DELETE), то

функция возвращает значение `true` при успешном выполнении команды и значение `false` при ошибке;

❑ функции обработки ошибок, о которых мы рассказывали выше.

Итак, создадим в папке `htdocs` корневой папки **ХАМРР** файл `save.php`, содержащий следующий код (листинг 5.8).

Листинг 5.8. Сохранение данных в базе

```
<html>
<head>
<title>Работа с MySQL</title>
</head>
<body>
<?php
//Отключаем вывод системных сообщений об ошибках
error_reporting(0);
//Получаем данные из формы input.php
$phone=$_POST["CustomerPhone"];
//Если номер телефона не введен, то связаться с клиентом невозможно.
//Предлагаем клиенту вернуться к заполнению формы
if(empty($phone) or ($phone == "(495)"))
{
    print "<h3>Пожалуйста, введите номер телефона</h3>";
    print "<input type='button' value='Вернуться к редактированию данных'
        onClick='history.go(-1)''";
}
//Если номер телефона введен, продолжаем обработку данных
else
{
    //Получаем из формы имя и адрес клиента
    $name=$_POST["CustomerName"];
    $address=$_POST["CustomerAddress"];
    //Соединяемся с сервером MySQL
    $connection = mysql_connect("localhost","username","userpassword");
    if(!$connection) die("Ошибка доступа к базе данных.
        Приносим свои извинения");
```

```

//Выбираем базу данных SalesDept (Отдел продаж)
//В случае ошибки формируем сообщение, записываем его в файл
//и отправляем по электронной почте
if(!mysql_select_db("SalesDept"))
{
    $err_message=date("Y.m.d H:i:s")."
    ".mysql_errno()." ".mysql_error()."\r\n";
    error_log($err_message,3,"/mysqlerror.log");
    error_log($err_message,1,"admin@somedomain.ru");
    die("Ошибка доступа к базе данных. Приносим свои извинения");
}
//Устанавливаем кодировку CP-1251
mysql_query("SET NAMES cp1251");
//Записываем данные о заказчике в таблицу Customers
$qresult = mysql_query("INSERT INTO Customers (name,phone,address)
VALUES
('".$name."','".$phone."','".$address."')");
//Проверяем результат выполнения команды; в случае ошибки формируем
//сообщение, записываем его в файл и отправляем по электронной почте
if(!$qresult)
{
    $err_message=date("Y.m.d H:i:s")."
    ".mysql_errno()." ".mysql_error()."\r\n";
    error_log($err_message,3,"/mysqlerror.log");
    error_log($err_message,1,"admin@somedomain.ru");
    die("Ошибка при сохранении данных. Приносим свои извинения");
}
print "<h3>Поздравляем! Регистрация завершена успешно</h3>";
}
?>
</body>
</html>

```

Если, например, в форме были введены значения полей Иванов, 157400 и Москва, а/я 255, то вызов функции:

```
$qresult = mysql_query("INSERT INTO Customers  
  (name,phone,address)  
VALUES  
  ('".$name."','".$phone."','".$address."')");
```

после подстановки значений переменных `$name`, `$phone` и `$address` будет выглядеть следующим образом:

```
$qresult = mysql_query("INSERT INTO Customers  
  (name,phone,address)  
VALUES  
  ('Иванов','157400','Москва, а/я 225')");
```

Если команда `INSERT` была выполнена успешно, то сценарий `save.php` выведет на странице соответствующее сообщение (рис. 5.5).

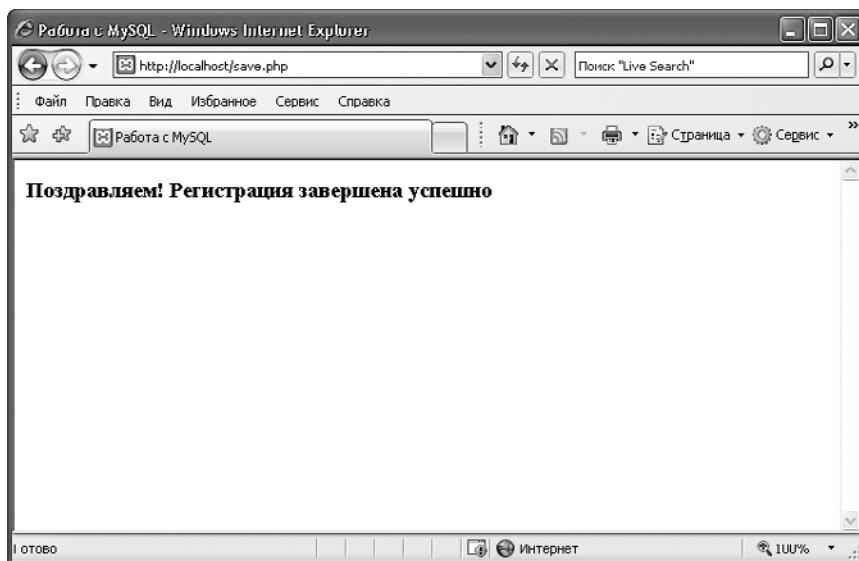


Рис. 5.5. Результат сохранения данных

Мы почти завершили создание приложения, которое записывает информацию в базу данных. Однако нужно сделать важное дополнение, которое связано с некорректным сохранением значений, содержащих спецсимволы.

Предположим, например, что пользователь вводит на веб-странице в поле **Ваше имя** значение `д'Артаньян`. Тогда вызов PHP-функции:

```
mysql_query("INSERT INTO Customers (name,phone,address)
```

```
VALUES
('".$name."' , '".$phone."' , '".$address."'");
```

приведет к попытке выполнения некорректной SQL-команды:

```
INSERT INTO Customers
(name,phone,address)
VALUES ('д'Артаньян','Телефон','Адрес');
```

В результате произойдет ошибка и сохранить в базе введенные пользователем данные не удастся.

Избежать этой ошибки позволяет функция:

```
mysql_real_escape_string("<Строка>"[,
<Указатель на соединение>]);
```

Функция `mysql_real_escape_string()` экранирует строку, полученную в качестве аргумента, то есть перед каждым специальным символом в этой строке (например, перед одинарной кавычкой) помещает обратную косую черту. Например, если в качестве аргумента передана строка д'Артаньян, то функция возвращает значение д\ 'Артаньян. Таким образом, при вызове PHP-функции:

```
mysql_query("INSERT INTO Customers (name,phone,address)
VALUES
('".$mysql_real_escape_string($name)."',
'".$mysql_real_escape_string($phone)."',
'".$mysql_real_escape_string($address)."')");
```

будет выполнена корректная SQL-команда:

```
INSERT INTO Customers
(name,phone,address)
VALUES ('д\ 'Артаньян','Телефон','Адрес');
```

Кроме того, в отдельных случаях функция `mysql_real_escape_string()` позволяет обезопасить PHP-приложение от *SQL-инъекций*, то есть предотвратить выполнение SQL-команд, которые недобросовестный пользователь может ввести в текстовые поля на веб-странице.

Исправим сценарий `save.php`, добавив вызов функции `mysql_real_escape_string()` (листинг 5.9).

Листинг 5.9. Сохранение данных в базе

```
<html>
<head>
```

```
<title>Работа с MySQL</title>
</head>
<body>
<?php
//Отключаем вывод системных сообщений об ошибках
error_reporting(0);
//Получаем данные из формы input.php
$phone=$_POST["CustomerPhone"];
//Если номер телефона не введен, то связаться с клиентом невозможно.
//Предлагаем клиенту вернуться к заполнению формы
if(empty($phone) or ($phone == "(495)"))
{
    print "<h3>Пожалуйста, введите номер телефона</h3>";
    print "<input type='button' value='Вернуться к редактированию данных'
        onClick='history.go(-1)''";
}
//Если номер телефона введен, продолжаем обработку данных
else
{
    //Получаем из формы имя и адрес клиента
    $name=$_POST["CustomerName"];
    $address=$_POST["CustomerAddress"];
    //Соединяемся с сервером MySQL
    $connection = mysql_connect("localhost","username","userpassword");
    if(!$connection) die("Ошибка доступа к базе данных.
        Приносим свои извинения");
    //Выбираем базу данных SalesDept (Отдел продаж)
    //В случае ошибки формируем сообщение, записываем его в файл
    //и отправляем по электронной почте
    if(!mysql_select_db("SalesDept"))
    {
        $err_message=date("Y.m.d H:i:s")."
            ".mysql_errno()." ".mysql_error()."\r\n";
        error_log($err_message,3,"/mysqlerror.log");
    }
}
```

```

        error_log($err_message,1,"admin@somedomain.ru");
        die("Ошибка доступа к базе данных. Приносим свои извинения");
    }
//Устанавливаем кодировку CP-1251
mysql_query("SET NAMES cp1251");
//Записываем данные о заказчике в таблицу Customers
$result = mysql_query("INSERT INTO Customers (name,phone,address)
VALUES
('".mysql_real_escape_string($name)."',
'".mysql_real_escape_string($phone)."',
'".mysql_real_escape_string($address)."'");
//Проверяем результат выполнения команды; в случае ошибки формируем
//сообщение, записываем его в файл и отправляем по электронной почте
if(!$result)
{
    $err_message=date("Y.m.d H:i:s")."
    ".mysql_errno()." ".mysql_error()."\r\n";
    error_log($err_message,3,"/mysqlerror.log");
    error_log($err_message,1,"admin@somedomain.ru");
    die("Ошибка при сохранении данных. Приносим свои извинения");
}
print "<h3>Поздравляем! Регистрация завершена успешно</h3>";
}
?>
</body>
</html>

```

На этом мы завершаем изучение PHP-функций, позволяющих организовать обмен данными с MySQL. Вы ознакомились с примерами PHP-приложений, использующих базу данных MySQL. Все они имеют сходную структуру.

1. Подключение к серверу MySQL.
2. Выбор базы данных.
3. Установка кодировки.
4. Выполнение SQL-команды (ввод, изменение или получение данных).
5. Обработка ошибки.

В итоге мы изучили все функциональные возможности СУБД MySQL по работе с данными, которые необходимы для решения большинства практических задач. Теперь мы рассмотрим вспомогательные, но не менее важные процессы: управление доступом пользователей к базе данных и предотвращение потерь данных в случае сбоев. Будет рассказано, как выполнять операции администрирования с помощью специальных SQL-команд.

5.2. Администрирование и безопасность баз данных MySQL

Начнем наше обсуждение с вопросов разграничения доступа пользователей к базам данных MySQL. Контроль действий пользователей включает два этапа.

- ❑ Когда пользователь пытается подключиться к серверу баз данных, программа MySQL проверяет, разрешено ли ему подключение, то есть проверяет его *учетную запись*.
- ❑ Когда пользователь пытается выполнить какую-либо операцию, программа MySQL проверяет, имеет ли пользователь право исполнять эту операцию. Для этого сервер MySQL поддерживает систему *привилегий*, запрещающих или разрешающих пользователю определенный вид доступа.

Следующий раздел посвящен операциям с учетными записями пользователей MySQL, после чего перейдем к обсуждению системы привилегий доступа.

Учетные записи пользователей

В этом разделе мы поговорим о настройке учетных записей: вы узнаете, как зарегистрировать или удалить пользователя MySQL, изменить его пароль, получить информацию о зарегистрированных пользователях.

Общие сведения

Под учетной записью пользователя MySQL мы подразумеваем строку в таблице `user` системной базы данных `mysql`. Первичным ключом в этой таблице служат столбцы `Host` и `User`. Таким образом, в MySQL идентификация пользователя основана не на имени пользователя, а на комбинации имени пользователя и хоста, с которого подключается пользователь. Это означает, что вы не просто можете ограничить круг хостов, с которых разрешено подключаться данному пользователю; а можете, например, создать *разные* учетные записи (а следовательно, назначить

разные привилегии доступа) для пользователя `anna`, подключающегося с компьютера `localhost`, и для пользователя `anna`, подключающегося с компьютера `somedomain.com`.

Столбец `User` допускает значения длиной не более 16 символов. Значение этого столбца может быть пустым, что соответствует анонимному пользователю, но в этой книге мы не будем рассматривать такую возможность.

Столбец `Host` допускает такие значения, как:

- ☐ конкретное имя компьютера или IP-адрес;
- ☐ маска подсети (например, `192.168.1.0/255.255.255.0`);
- ☐ маска имени компьютера или маска IP-адреса, которая может содержать подстановочные символы:
 - `' % '` — на месте знака процента может быть любое количество произвольных символов;
 - `' _ '` — на месте знака подчеркивания может быть ровно один произвольный символ.



ПРИМЕЧАНИЕ

Если маска начинается с цифры или точки, то она рассматривается как маска IP-адреса. Поэтому значение `122.somedomain.com` не соответствует маске `'122.%'`.

В командах, управляющих учетными записями пользователей MySQL, используется *идентификатор пользователя* — значение первичного ключа учетной записи в формате:

```
'<Значение столбца User>'['@'<Значение столбца Host>']
```

Например, `'anna'@'localhost'`. Если значение столбца `Host` не указано, подразумевается маска `' % '`, так что идентификаторы `'anna'` и `'anna'@' % '` эквивалентны.

При подключении пользователя к серверу MySQL происходит идентификация пользователя — поиск соответствующей ему учетной записи. Поиск начинается с тех строк таблицы `user`, в которых значение столбца `Host` не содержит подстановочных символов. Поэтому, например, если в таблице зарегистрированы две учетные записи с идентификаторами `'anna'@' % '` и `'anna'@'localhost'` соответственно, то при подключении пользователя с именем `anna` с локального компьютера будет выбрана вторая учетная запись.

После определения учетной записи выполняется аутентификация (проверка подлинности) пользователя, которая заключается в сравнении введенного пользователем пароля с паролем учетной записи, хранящимся в столбце `Password` таблицы `user` (обратите внимание, что пароли хранятся и передаются только в зашифрованном виде). Если пароль указан правильно, то первый этап контроля доступа завершается успешно и устанавливается соединение клиентского приложения с сервером.



ПРИМЕЧАНИЕ

Имена пользователей и пароли чувствительны к регистру символов, а имена хостов — не чувствительны.

Теперь, изучив особенности идентификации и аутентификации пользователей MySQL, вы можете переходить к регистрации новых пользователей.

Регистрация пользователя

Чтобы создать учетную запись пользователя, выполним команду:

```
CREATE USER <Идентификатор пользователя>  
[IDENTIFIED BY [PASSWORD] '<Пароль>'];
```

Обязательным параметром этой команды является идентификатор нового пользователя. Если не задан параметр `IDENTIFIED BY`, то будет использоваться пустой пароль.

Параметр `PASSWORD` необходимо указать в том случае, если вы вводите не реальный, а зашифрованный пароль (что позволяет избежать передачи незашифрованного пароля при отправке на сервер команды `CREATE USER`). Получить зашифрованное значение из реального пароля вы можете с помощью функции:

```
PASSWORD('<Реальный пароль>')
```

Например, команда:

```
CREATE USER 'anna' IDENTIFIED BY 'annapassword';
```

создает учетную запись для пользователя с именем `anna`, подключающегося с любого компьютера, и устанавливает для этой учетной записи пароль `annapassword`. Команда:

```
CREATE USER 'anna'@'localhost' IDENTIFIED BY PASSWORD  
'*3C7F72EAE78BC95AAFBFD21F8741C24A0056C04B';
```

создает учетную запись для пользователя `anna`, подключающегося с локального компьютера, и устанавливает в качестве пароля значение `annalocpassword` (поскольку функция `PASSWORD('annalocpassword')` возвращает значение `'*3C7F72EAE78BC95AAFBFD21F8741C24A0056C04B'`).

Сразу после выполнения команды `CREATE USER` новый пользователь может подключаться к серверу MySQL.

В следующем подразделе мы обсудим, как изменить пароль пользователя, а также как восстановить забытый пароль пользователя `root`.

Установка пароля

Для установки пароля предназначена команда:

```
SET PASSWORD [FOR <Идентификатор пользователя>]  
= PASSWORD(<Пароль>);
```

Параметрами этой команды являются идентификатор учетной записи пользователя и новый пароль для этой записи. Если вы не укажете идентификатор пользователя, то измените свой пароль.

Вместо функции `PASSWORD()`, зашифровывающей реальный пароль, можно сразу ввести зашифрованный пароль. Например, команды:

```
SET PASSWORD FOR 'anna'@'%' =  
PASSWORD('newannapassword');
```

и

```
SET PASSWORD FOR 'anna'@'%' =  
'*006B99DE1BDA1BE6E1FFF714E764A8FAB0E614DF';
```

устанавливают пароль `newannapassword` для пользователя `anna`, подключающегося с любого компьютера. Эти команды никак не влияют на другие учетные записи, например, пароль учетной записи с идентификатором `'anna'@'localhost'` не изменится.

Итак, вы научились работать с паролями пользователей. Теперь рассмотрим операцию удаления пользователя.

Удаление пользователя

Удалить учетную запись вы можете с помощью команды:

```
DROP USER <Идентификатор пользователя>;
```

После удаления пользователь лишается возможности подключаться к серверу MySQL. Однако если на момент удаления он был подключен к серверу, то соединение не прерывается.

Вместе с учетной записью удаляются все привилегии доступа для этой записи.

Наконец, рассмотрим поиск информации о пользователях, зарегистрированных в системе.

Просмотр учетных записей

Для получения информации о зарегистрированных пользователях выполним запрос к таблице `user` системной базы данных `mysql`, например:

```
SELECT * FROM mysql.user;
```

Первые три столбца таблицы `user` нам уже знакомы — это `Host`, `User` и `Password`. Далее следуют *столбцы глобальных привилегий*, о которых я расскажу чуть ниже, и, наконец, столбцы, в которых содержатся параметры безопасности соединения и сведения о ресурсах, предоставляемых соединению (эти столбцы остаются за рамками нашего обсуждения).

Итак, вы научились выполнять все основные операции с учетными записями пользователей MySQL. Далее вы узнаете, как назначить учетным записям те или иные привилегии доступа.

Система привилегий доступа

Этот раздел посвящен второму этапу контроля доступа пользователей — проверке привилегий доступа при выполнении каждой операции в базе данных. Я расскажу, какие привилегии предусмотрены в MySQL и как предоставить эти привилегии пользователям.

Общие сведения

Создание привилегии доступа в MySQL подразумевает определение следующих параметров:

- ❑ идентификатор учетной записи пользователя, которому предоставляется привилегия;
- ❑ тип привилегии, то есть тип операций, которые будут разрешены пользователю;
- ❑ область действия привилегии.

Перечислю основные типы привилегий, используемых в MySQL:

- ☐ ALL [PRIVILEGES] — предоставляет все привилегии, кроме GRANT OPTION, для указанной области действия;
- ☐ ALTER — разрешает выполнение команд ALTER DATABASE и ALTER TABLE;
- ☐ CREATE — разрешает выполнение команд CREATE DATABASE и CREATE TABLE;
- ☐ CREATE USER — разрешает выполнение команд CREATE USER, DROP USER, RENAME USER;
- ☐ DELETE — разрешает выполнение команды DELETE;
- ☐ DROP — разрешает выполнение команд DROP DATABASE и DROP TABLE;
- ☐ FILE — разрешает чтение и создание файлов на сервере с помощью команд SELECT ... INTO OUTFILE и LOAD DATA INFILE;
- ☐ INDEX — разрешает выполнение команд CREATE INDEX и DROP INDEX;
- ☐ INSERT — разрешает выполнение команды INSERT;
- ☐ SELECT — разрешает выполнение команды SELECT;
- ☐ LOCK TABLES — разрешает выполнение команды LOCK TABLES при наличии привилегии SELECT для блокируемых таблиц;
- ☐ SHOW DATABASES — разрешает отображение всех баз данных при выполнении команды SHOW DATABASES (если эта привилегия отсутствует, то в списке будут отображены только те базы данных, по отношению к которым у пользователя есть какая-либо привилегия);
- ☐ RELOAD — разрешает выполнение команды FLUSH;
- ☐ SUPER — привилегия администратора сервера; в частности, разрешает выполнение команды SET GLOBAL;
- ☐ UPDATE — разрешает выполнение команды UPDATE;
- ☐ GRANT OPTION — разрешает назначать и отменять привилегии другим пользователям (эта возможность распространяется только на те привилегии, которые есть у самого пользователя для указанной области действия).



ПРИМЕЧАНИЕ

Здесь приведены только те типы привилегий, которые требуются для выполнения операций, описанных в данной книге. Полный список типов привилегий вы можете найти в документации компании-разработчика на веб-странице <http://dev.mysql.com/doc/refman/5.0/en/privileges-provided.html>.

Областью действия привилегии могут быть:

- ☐ все базы данных (такие привилегии называются глобальными);
- ☐ отдельная база данных;
- ☐ таблица;
- ☐ столбец таблицы.

Каждый тип привилегии имеет свои допустимые области действия. Так, привилегии `FILE`, `SHOW DATABASES`, `RELOAD`, `SUPER` и `CREATE USER` могут быть только глобальными. Привилегия `LOCK TABLES` может применяться глобально или к отдельным базам данных, но не к отдельным таблицам. К отдельным столбцам таблицы применимы только привилегии `SELECT`, `INSERT` и `UPDATE`.

Чтобы получить разрешение на выполнение операции с каким-либо объектом базы данных, пользователю достаточно иметь привилегию соответствующего типа для какой-либо области действия, содержащей этот объект. Например, пользователь сможет выполнить запрос данных из столбца `description` таблицы `Products` базы данных `SalesDept`, если у него есть хотя бы одна из следующих привилегий:

- ☐ глобальная привилегия `SELECT`;
- ☐ привилегия `SELECT` для базы данных `SalesDept`;
- ☐ привилегия `SELECT` для таблицы `Products`;
- ☐ привилегия `SELECT` для столбца `description`.

Для выполнения некоторых операций может потребоваться несколько типов привилегий. Например, команда:

```
UPDATE SalesDept.Products SET price='548.00' WHERE id=5;
```

доступна пользователю, если у него одновременно есть привилегия `SELECT` для таблицы `Products` (или для базы данных `SalesDept`, или глобальная) и привилегия `UPDATE` для столбца `price` (или для таблицы `Products`, или для базы данных `SalesDept`, или глобальная).

Теперь, получив общее представление о привилегиях доступа в MySQL, вы можете переходить к назначению привилегий пользователям.

Предоставление привилегии

Для предоставления привилегий пользователям используется команда:

```
GRANT <Тип привилегии>  
[((<Список столбцов>)] ON <Область действия>
```

```
TO <Идентификатор пользователя>  
[WITH GRANT OPTION];
```

В качестве области действия вы можете указать одно из следующих значений:

- ❑ `*.*` — привилегия будет действовать глобально;
- ❑ `<Имя базы данных>.*` — привилегия будет действовать для указанной базы данных;
- ❑ `*` — привилегия будет действовать для базы данных, которая в момент выполнения команды `GRANT` являлась текущей;
- ❑ `<Имя базы данных>.<Имя таблицы>` или `<Имя таблицы>` — привилегия будет действовать для указанной таблицы (если имя базы данных не указано, то подразумевается текущая база данных); если требуется создать привилегию не для всей таблицы, а только для отдельных столбцов, необходимо перечислить эти столбцы в скобках перед ключевым словом `ON`.

Приведу несколько примеров.

- ❑ `GRANT CREATE ON *.* TO 'anna'@'localhost';`

Команда предоставляет пользователю `'anna'@'localhost'` привилегию на создание баз данных и таблиц в любой базе данных.

- ❑ `GRANT DROP ON SalesDept.* TO 'anna'@'localhost';`

Команда предоставляет пользователю `'anna'@'localhost'` привилегию на удаление таблиц в базе данных `SalesDept`, а также на удаление самой базы данных `SalesDept`.

- ❑ `GRANT SELECT ON SalesDept.Products TO 'anna'@'localhost';`

Команда предоставляет пользователю `'anna'@'localhost'` привилегию на получение данных из таблицы `Products` базы данных `SalesDept`.

- ❑ `GRANT UPDATE (price) ON SalesDept.Products TO 'anna'@'localhost';`

Команда предоставляет пользователю `'anna'@'localhost'` привилегию на изменение данных в столбце `price` таблицы `Products`.

При назначении привилегий необходимо иметь в виду следующие особенности.

- ❑ Если учетная запись с указанным идентификатором не существует, то команда `GRANT` может создать такую запись. Отключить автоматическое создание учетной записи позволяет ключевое слово `NO_AUTO_CREATE_USER` в значении переменной `sql_mode` (напомню, что настройку режима взаимодействия с сервером MySQL мы обсуждали в разделе 3.4 главы 3).

- ❑ Привилегию ALTER рекомендуется предоставлять с осторожностью: путем переименования таблиц и столбцов пользователь может изменить настройки системы привилегий.
- ❑ Если при создании привилегии вы указываете параметр WITH GRANT OPTION, то пользователь получает возможность «делиться» не только созданной привилегией, но и *другими* своими привилегиями в рамках данной области действия.

Например, после выполнения команд:

```
GRANT SELECT ON *.* TO 'marina';
```

```
GRANT INSERT ON SalesDept.* TO 'marina' WITH GRANT OPTION;
```

```
GRANT DELETE ON SalesDept.Customers TO 'marina';
```

пользователь **marina** может предоставлять другим пользователям следующие привилегии:

- привилегии SELECT, INSERT и GRANT OPTION на уровне базы данных SalesDept; хотя сам пользователь **marina** имеет глобальную привилегию SELECT, его возможности делегирования привилегий ограничены базой данных SalesDept;
 - привилегию DELETE на уровне таблицы Customers базы данных SalesDept, так как область действия этой привилегии входит в область действия привилегии GRANT OPTION.
- ❑ Если вы предоставляете привилегию GRANT OPTION нескольким пользователям, то они могут «обменяться» привилегиями, то есть объединить свои наборы привилегий.
 - ❑ Привилегии, областью действия которых является таблица или столбец, вступают в силу немедленно — пользователь может сразу же начать выполнять SQL-команды, разрешенные ему новой привилегией. Привилегии, относящиеся к базе данных, начинают действовать после выполнения команды USE <имя базы данных>, то есть после выбора какой-либо базы данных в качестве текущей. Глобальные привилегии начинают применяться при следующем подключении пользователя к серверу MySQL.

Итак, вы ознакомились с командой добавления привилегий. В следующем подразделе мы рассмотрим команду отмены привилегий.

Отмена привилегий

Чтобы удалить привилегию, ранее назначенную пользователю, используется команда:

```
REVOKE <Тип привилегии>
```

```
[(<Список столбцов>)] ON <Область действия>
```

FROM <Идентификатор пользователя>;

Например:

❑ REVOKE CREATE ON *.* FROM 'anna'@'localhost';

Команда отменяет глобальную привилегию пользователя 'anna'@'localhost', разрешавшую создание баз данных и таблиц;

❑ REVOKE DROP ON SalesDept.* FROM 'anna'@'localhost';

Команда отменяет привилегию пользователя 'anna'@'localhost' на удаление базы данных SalesDept и таблиц в этой базе данных;

❑ REVOKE SELECT ON SalesDept.Products FROM 'anna'@'localhost';

Команда отменяет привилегию пользователя 'anna'@'localhost' на получение данных из таблицы Products базы данных SalesDept;

❑ REVOKE UPDATE (price) ON SalesDept.Products FROM 'anna'@'localhost';

Команда отменяет привилегию пользователя 'anna'@'localhost' на изменение данных в столбце price таблицы Products.

Параметры команды REVOKE имеют тот же смысл, что и параметры команды GRANT. Аналогичны и правила вступления изменений в силу.

Надо отметить, что в MySQL при удалении баз данных, таблиц и столбцов связанные с ними привилегии не удаляются автоматически; для удаления таких привилегий требуется выполнить команду REVOKE.

Теперь вы знаете, как добавлять и удалять привилегии доступа. В следующем разделе мы поговорим о том, как получить информацию о зарегистрированных привилегиях.

Просмотр привилегий

Сведения о привилегиях доступа содержатся в следующих таблицах системной базы данных mysql.

- ❑ Глобальные привилегии хранятся в таблице user, которая уже знакома вам из предыдущего раздела. Каждому типу привилегии соответствует отдельный столбец, допускающий значения 'Y' (операция разрешена) и 'N' (операция не разрешена).
- ❑ Привилегии, областью действия которых является отдельная база данных, хранятся в таблице db (база данных). Первичный ключ в этой таблице образу-

ют столбцы `Host`, `Db` и `User`. Таким образом, каждая строка таблицы определяет привилегии одного пользователя по отношению к одной базе данных. Как и в таблице `user`, каждой привилегии соответствует отдельный столбец, возможными значениями которого являются 'Y' и 'N'.



ПРИМЕЧАНИЕ

При проверке доступа пользователей к базе данных используется также таблица `host`. Однако команды `GRANT` и `REVOKE` не затрагивают эту таблицу, и обычно она остается пустой.

- ❑ Привилегии, относящиеся к отдельным таблицам, хранятся в таблице `tables_priv`. Первичным ключом в ней служат столбцы `Host`, `Db`, `User` и `Table_name` (Имя таблицы). Таким образом, каждая строка таблицы `tables_priv` определяет привилегии доступа конкретного пользователя к конкретной таблице. Типы привилегий, которыми обладает пользователь, перечислены в столбце `Table_priv` (Привилегии доступа к таблице), имеющем тип данных `SET`. Кроме того, в таблице `tables_priv` имеется столбец `Column_priv` (Привилегии доступа к столбцу) с типом данных `SET`, значение которого указывает, что у пользователя имеются привилегии доступа к отдельным столбцам таблицы.
- ❑ Привилегии для отдельных столбцов хранятся в таблице `columns_priv`. Первичный ключ этой таблицы состоит из столбцов, идентифицирующих пользователя (`Host` и `User`), и столбцов, идентифицирующих столбец (`Db`, `Table_name` и `Column_name`). Типы привилегий, которыми обладает пользователь по отношению к столбцу, перечислены в столбце `Column_priv` (Привилегии доступа к столбцу) с типом данных `SET`.

Таблицы `user`, `db`, `tables_priv` и `columns_priv` можно использовать для получения информации о пользователях, обладающих привилегиями доступа к интересующему вас объекту базы данных. Если же требуется, наоборот, узнать, к каким объектам имеет доступ конкретный пользователь, выполните команду:

```
SHOW GRANTS [FOR <Идентификатор пользователя>];
```

Команда `SHOW GRANTS` выводит сведения о привилегиях пользователя в виде набора команд `GRANT`, с помощью которых можно сформировать текущий набор привилегий пользователя. Если идентификатор пользователя не задан, вы увидите свои привилегии.

Итак, вы научились добавлять, удалять и просматривать привилегии с помощью специальных команд. Примите к сведению, что выполнять те же самые операции можно в наглядном интерфейсе графической утилиты `phpMyAdmin`, которая широко используется различными хостинг-провайдерами как инструмент администрирования баз данных MySQL.

5.3. Резюме

В текущей главе вы получили базовые сведения об управлении пользователями и их привилегиями, о резервировании базы данных и ее восстановлении в случае сбоя, об исправлении повреждений в таблицах с типом `MyISAM`, а также о просмотре журналов, создаваемых в процессе функционирования сервера `MySQL`. Эти сведения помогут вам успешно использовать базы данных и минимизировать возможные проблемы.

При этом мы рассмотрели только самые необходимые для взаимодействия с `MySQL` функции языка `PHP`. Их полный список вы можете найти в Руководстве по `PHP` на странице <http://www.php.net/manual/ru/ref.mysql.php>. Но этим возможности `PHP` не исчерпываются. В следующей главе мы обсудим технику асинхронных запросов (`Ajax`), которая специально разработана для получения клиентом информации большого объема в процессе работы с сайтом, причем так, чтобы посетитель сайта не замечал никаких неудобств. Это совершенно необходимая вещь, без которой не обходится ни один мало-мальски полноценный интернет-магазин, предлагающий своим покупателям обширный каталог своих товаров. Средства `Ajax` позволят вам так построить обмен данными с сайтом, что пользователь даже не заметит и не почувствует каких-либо задержек в работе сайта, связанных с загрузкой данных большого объема, что чрезвычайно важно для работы коммерческого сервиса.

Глава 6

Технология AJAX

Ранее мы рассмотрели несколько веб-приложений, созданных на основе языка PHP. Это были простые учебные примеры, все назначение которых состоит в демонстрации технических приемов, применяемых при написании сценариев PHP. Если же вы займетесь разработкой «настоящего», имеющего практическое назначение веб-приложения, то перед вами сразу встанет задача, которую можно назвать важнейшей из всех: обеспечение удобства пользователя (на сленге «юзабилити», от английского usability — «удобство», «практичность», «простота использования»), работающего с вашим приложением.

Веб-приложение работает при участии серверного сценария, исполняемого на веб-сервере (например, Apache), и клиента, работающего на локальном компьютере. Последним чаще всего используется браузер, подключенный к соответствующему веб-сервису. В этой связке клиент обеспечивает интерфейс работы с пользователем, а сервер обрабатывает данные, пересылаемые ему клиентом. Например, если вы работаете с поисковой машиной, то с помощью браузера посылаете серверу запрос на поиск данных определенного вида, а сервер выполняет этот поиск в своей базе данных и передает вам информацию. Все это взаимодействие вы наблюдали при исполнении примеров из предыдущей главы.

Но что будет, если объем передаваемых клиенту данных окажется велик? Или поиск в базе данных будет достаточно длительным? Тогда, если мы останемся в рамках той технологии общения клиента с сервером, которую применяли в своих примерах, пользователь вашего веб-приложения будет испытывать большие

неудобства, связанные с длительными задержками в передаче по сети данных большого объема или с загруженностью сервера.

Эта проблема особо остро встала буквально в последние годы, когда наметилась явная тенденция использования Интернета в тех сферах, для которых он не был предназначен изначально. В самом деле, поначалу Сеть создавалась как средство для извлечения данных небольшого объема и передачи их клиентам по маломощным линиям связи (кто не помнит эпоху «диал-апа», когда все мы подключались к Глобальной сети через телефонные провода, проложенные в те времена, когда понятия «Интернет» не было и в помине?). Нынешние веб-приложения стали претендовать на те же функции, которыми ранее оснащались «обычные» приложения, работающие или на пользовательском компьютере, или в локальной сети офиса.

И вот, чтобы обеспечить юзабилити своих веб-приложений, позволяющих работать с большими объемами данных или значительно нагружающих сервер, была придумана технология AJAX, которая расшифровывается как *Asynchronous JavaScript and XML* — *асинхронный JavaScript и XML*. И сейчас мы приступим к ее освоению.

6.1. Как работает AJAX

Суть технологии AJAX, как это и следует из ее названия, состоит в отправке запросов серверу, которые исполняются в асинхронном режиме. Это значит, что серверная часть веб-приложения, использующего AJAX, работает независимо от клиентской. После того как клиент послал запрос серверу, сервер начинает его обработку, а клиент продолжает свою работу, не вынуждая пользователя дожидаться ответа. Например, пользователь может вводить данные в форму, отображенную в окне клиента (в данном случае браузера), а клиент, не дожидаясь окончания ввода и нажатия кнопки подтверждения формы, отправляет данные на сервер, который скрытно от пользователя обрабатывает данные и отправляет их результаты клиенту.

Что это дает? А вот что. В наших простых примерах мы получали ответ от сервера практически мгновенно и никакого AJAX нам не требовалось. Но в случае, допустим, решения задачи поиска, при котором сервер должен выполнить большую работу и переслать объемные данные, мы сможем снизить время ожидания пользователем результатов, если сервер будет выполнять поиск параллельно с вводом данных в форму поиска.

Яркий пример тому — поиск по каталогу. По мере того как пользователь открывает щелчками кнопкой мыши новые разделы, веб-приложение с помощью AJAX загружает информацию раздела, предполагая ее использование при последующих

шагах выбора. Это очень удобно при реализации интернет-магазинов, когда посетитель ищет какой-то товар в каталоге, скажем автомобиль, и сначала выбирает производителя, затем модель, потом какие-то модификации и параметры поставки, комплектацию, детали дизайна: цвет корпуса, салона и т. д. Так вот, AJAX очень сильно поможет вам удержать такого потенциального покупателя от бегства на другой сайт. Ведь посетитель не захочет ждать на каждом шаге долгого отклика сервера, ему нужно, чтобы на его действия следовала мгновенная (или быстрая) реакция приложения. Что тут может сделать AJAX? Да все! Пока ваш покупатель разглядывает каталог автомобилей одного производителя, раздумывает над вариантами выбора, указывает нужные параметры покупки, вы с помощью AJAX загружаете на его компьютер всю информацию об автомобилях данного типа. И когда покупатель решится на выбор модели, у вас на клиентском компьютере уже все будет готово для быстрого ответа.

Примеров можно привести множество. Надо только упомянуть, что технологию AJAX используют крупнейшие поисковики (Yahoo!, Google), известные интернет-магазины и многие веб-сервисы.

На чем же основана технология AJAX? Опять-таки, соответственно названию, это языки JavaScript и XML. Далее я предполагаю, что вам знаком язык JavaScript. Если это не так, рекомендую изучить хотя бы его основы. Мы же начнем знакомство с технологией AJAX с краткого введения в язык XML.

Знакомство с XML

Аббревиатура XML означает *eXtensible Markup Language*, что переводится как *расширяемый язык разметки*. Более развернутое значение аббревиатуры — расширяемая спецификация языка, предназначенного для создания веб-страниц. В чем состоит назначение языка XML? Язык HTML, который мы обсуждали в главе 1, представляет собой некий необязательный и нестрогий поддерживаемый свод правил оформления гипертекстовых документов. Множество браузеров интерпретируют его теги и синтаксические конструкции на свой лад, вводят в язык свои дополнения и расширения и даже нарушают его требования. Кроме того, появилось множество устройств, работающих с Сетью, например смартфоны, коммуникаторы, для которых требуются средства, не включенные в классический HTML.

Все это было учтено организацией W3C при определении пути развития языка HTML, которое преодолело бы указанные недостатки. Этот путь состоит в следующем: вместо жесткого ограничения и пресечения различных вольностей в трактовке языка HTML или создания универсального, всеобъемлющего языка гипертекстовой разметки было

решено создать метаязык, предназначенный для создания собственного языка гипертекстовой разметки или его модификации. В результате был создан XML.

Язык XML содержит средства, позволяющие создавать теги для конкретного языка разметки, определять их структуру, вложенность, набор атрибутов и их значений. Соответствующие конструкции XML реализуют так называемые правила *DTD* (*Document Type Definition, определение типа документа*), которым должен удовлетворять XML-документ, подчиняющийся созданному языку. Для того чтобы браузер или любая другая программа могли обрабатывать XML-документы, для каждого из них указывается соответствующий набор правил DTD.

Вот как это выглядит на практике. Ниже приведен пример документа XML, который, несмотря на простоту, содержит все необходимые компоненты XML.

```
<?XML version="1.0"?>
<hello>
<title>Это документ XML</title>
<body>Наш первый XML-документ</body>
</hello>
```

Первый тег содержит указание браузеру, что этот документ подготовлен согласно языку XML версии 1.0. Знак ? после угловой скобки сообщает браузеру, что далее следуют инструкции по обработке документа, которыми браузер должен руководствоваться. Эти инструкции завершаются знаком вопроса (?) перед угловой скобкой, то есть все инструкции браузеру помещаются между записями <? и ?>.

Саму гипертекстовую разметку задают дескрипторами, аналогично языку HTML. Дескрипторы могут быть одиночными и парными, вкладываться друг в друга и обладать набором атрибутов. Все это определяется набором правил DTD документа, подготовленного средствами XML. Для примера напишем правила DTD для вышеприведенного документа XML.

```
<!ELEMENT HELLO (TITLE, BODY)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT BODY (#PCDATA)>
```

Эти правила указывают, что в нашем документе XML могут использоваться дескрипторы HELLO, TITLE и BODY. При этом TITLE и BODY обязательно должны лежать внутри HELLO и содержать символьные данные #PCDATA.

Для того чтобы связать наш документ XML с соответствующим набором правил DTD, следует указывать ключевое слово DOCTYPE. Вот как это делается при использовании внешнего, то есть не включенного в сам документ, набора правил DTD.

Сначала сохраним наш набор правил DTD в файле `hello.dtd`. Тогда документ XML будет таков.

```
<?XML version="1.0"?>
<!DOCTYPE HELLO SYSTEM "hello.dtd">
<hello>
<title>Это документ XML</title>
<body>Наш первый XML-документ</body>
</hello>
```

Здесь дескриптор `<!DOCTYPE` содержит ссылку на файл правил `hello.dtd`, а ключевое слово `SYSTEM` означает, что этот набор является системным, то есть недоступным для общего использования (в противном случае указывается ключевое слово `PUBLIC`).

Набор правил DTD можно просто включить в документ XML, написав его следующим образом:

```
<?XML version="1.0"?>
<!DOCTYPE HELLO [
<!ELEMENT HELLO (TITLE, BODY)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT BODY (#PCDATA)>
]>
<hello>
<title>Это документ XML</title>
<body>Наш первый XML-документ</body>
</hello>
```

Теперь браузер или другая программа будут знать, что им делать при обработке XML-документа. Придется лишь найти корневой элемент (в нашем случае `HELLO`) и выстроить иерархическую структуру из вложенных дескрипторов. В нашем примере все потомки корневого дескриптора ограничиваются двумя тегами `TITLE` и `BODY`, но более сложные конструкции DTD позволяют определять иерархическую структуру любой глубины. Это очень удобно при использовании структурированных данных, передаваемых для обработки в различные программы, не обязательно связанные с отображением ресурсов Сети. Поэтому язык XML получил широкое распространение в других сферах, в частности, его используют в технологии AJAX для обмена данными между сервером и клиентом в асинхронном режиме.

Перейдем теперь к рассмотрению техники программирования веб-приложений средствами AJAX.

Первое веб-приложение с использованием AJAX

Создадим и сохраним в папке `examples` файл `ajax.html` следующего документа HTML (листинг 6.1).

Листинг 6.1. Веб-страница с запросами AJAX

```
<html>
  <head>
    <title>Пример AJAX</title>
    <script type="text/javascript" src="ajax.js"></script>
  </head>
  <body onload='request()'>
    Enter your login:
    <input type="text" id="myLogin" />
    <div id="Message" />
  </body>
</html>
```

Как видите, он содержит дескриптор загрузки сценария JavaScript, сохраненного в файле `ajax.js` (листинг 6.2). Этот сценарий содержит операторы функции JavaScript для инициации запросов AJAX и их исполнения.

Листинг 6.2. Сценарий инициации и обработки запросов AJAX

```
if(window.ActiveXObject)
  var xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
else
  var xmlhttp = new XMLHttpRequest();
if (!xmlhttp)
  alert("Error creating the XMLHttpRequest object.");
function request() {
  if (xmlhttp.readyState == 4 || xmlhttp.status == 0) {
    name = document.all("myLogin").value;
    xmlhttp.open("GET", "ajax.php?login=" + name, true);
    xmlhttp.onreadystatechange = ResponseHandler;
```

```

        xmlHttp.send(null);
    }
    else
        setTimeout('request()', 1000);
}
function ResponseHandler() {
    if (xmlHttp.readyState == 4) {
        if (xmlHttp.status == 200) {
            xmlResponse = xmlHttp.responseXML;
            xmlDocumentElement = xmlResponse.documentElement;
            Response = xmlDocumentElement.firstChild.data;
            document.all("Message").innerHTML =
                '<i>' + Response + '</i>';
            setTimeout('request()', 1000);
        }
        else {
            alert("Ошибка доступа к серверу");
        }
    }
}

```

Сохраним файл этого сценария в папке **examples** под именем **ajax.js**. Далее создадим сценарий PHP и сохраним его в папке **examples** под именем **ajax.php** (листинг 6.3).

Листинг 6.3. Серверная часть приложения

```

<?php
header('Content-Type: text/xml');
echo '<?xml version="1.0" ?>';
echo '<response>';
$login = $_GET['login'];
$userNames = array('PETYA', 'SERGEY', 'IVAN', 'MASHA', 'LENA');
if (in_array(strtoupper($login), $userNames))
    echo 'Hello, user ' . $login . '!';
else if (trim($login) == '')
    echo 'Empty login';
else

```

```
echo $login, ', - unregistered user';
echo '</response>';
?>
```

Откроем документ `ajax.html`, введя в строку браузера `localhost/examples/ajax.html`. В окне браузера отобразится поле `Enter your login` для ввода входного имени пользователя. Начав вводить в него какой-либо текст, вы заметите, как параллельно с вводом в строке под полем отображается сообщение сервера о результатах обработки введенных данных (рис. 6.1).

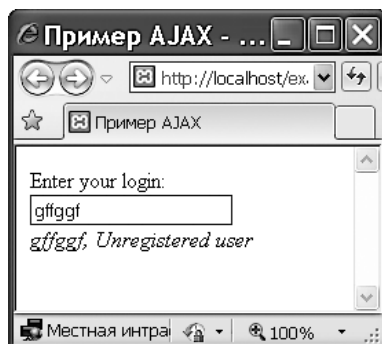


Рис. 6.1. Вводимые в поле данные обрабатываются сервером по запросам AJAX

Обратите внимание, что вам не требуется нажимать кнопку подтверждения формы, как это приходилось делать в примерах ранее. Сервер сам, независимо от ваших действий, считывает введенные данные и сравнивает их со списком известных ему имен пользователей. Если введенный текст не совпадает с каким-либо известным серверу именем, то отображается сообщение `Unregistered user` (Незарегистрированный пользователь). Если же ввести имя, известное серверу, то отобразится приветствие (рис. 6.2).

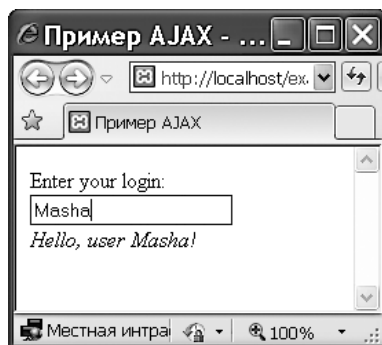


Рис. 6.2. Введенное имя распознано сервером

Теперь обсудим, как работает это веб-приложение. При загрузке страницы вместе с кодом HTML из листинга 6.1 загружается код программы JavaScript, представленный в листинге 6.2. При этом, согласно правилам обработки кода HTML-документа, будет выполнена часть программы JavaScript, расположенная в начале листинга 6.2:

```
if(window.ActiveXObject)
    var xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
else
    var xmlhttp = new XMLHttpRequest();
if (!xmlhttp)
    alert("Error creating the XMLHttpRequest object.");
```

Здесь создается объект `XMLHttpRequest`, отвечающий за выполнение запросов AJAX. В зависимости от того, какой браузер используется, объект создается вызовом либо функции `ActiveObject()`, либо `XMLHttpRequest()`. При успешном создании объекта `XMLHttpRequest` переменной `xmlhttp` присваивается значение ссылки на объект, при неудаче создания объекта генерируется сообщение.

Теперь обратимся к следующей строке кода HTML из листинга 6.1:

```
<body onload='request()'
```

Она означает, что при загрузке страницы вызывается функция `request()` из сценария `ajax.js`:

```
function request() {
    if (xmlhttp.readyState == 4 || xmlhttp.readyState == 0) {
        name = document.all("myLogin").value;
        xmlhttp.open("GET", "ajax.php?login=" + name, true);
        xmlhttp.onreadystatechange = ResponseHandler;
        xmlhttp.send(null);
    }
    else
        setTimeout('request()', 1000);
}
```

Работа этой функции состоит из двух частей: подготовки запросов AJAX к исполнению и запуска механизма этих запросов. Сначала проверяется состояние готовности объекта `XMLHttpRequest`. Это делается благодаря параметру объекта `readyState`, который может иметь такие значения:

- ❑ 0 — запрос не инициализирован;
- ❑ 1 — идет отправка запроса;

- ❑ 2 — запрос отправлен;
- ❑ 3 — происходит обмен данными;
- ❑ 4 — запрос завершен.

Итак, мы проверяем, был ли запрос завершен или не инициализирован, и в любом из этих случаев начинаем его исполнение. В противном случае с помощью функции `setTimeout()` устанавливается задержка повторного вызова функции `request()`:

```
setTimeout('request()', 1000);
```

На первом шаге подготовки запросов AJAX с помощью коллекции `all` мы определяем значение элемента `myLogin` в нашем документе HTML, то есть поля, содержащего введенный пользователем текст:

```
name = document.all("myLogin").value;
```

Далее с помощью метода `open` объекта `XMLHttpRequest` мы вызываем исполнение серверного сценария PHP:

```
xmlHttp.open("GET", "ajax.php?login=" + name, true);
```

Как это видно из строки вызова сценария:

```
"ajax.php?login=" + name
```

ему передается текст, введенный пользователем в поле нашего HTML-документа. Мы также задаем асинхронный режим обмена информацией с сервером, поскольку второй параметр метода равен `true`.

После этого запускается механизм AJAX асинхронных запросов: клиент будет автоматически, без всякого вмешательства пользователя посылать запросы серверу и в случае отклика передавать полученные данные на обработку специальной функцией. Эта функция назначается следующим оператором:

```
xmlHttp.onreadystatechange = ResponseHandler;
```

Здесь мы просто присвоили параметру `onreadystatechange` объекта `XMLHttpRequest` ссылку на функцию `ResponseHandler()`.

Теперь все готово для запросов AJAX. Вызываем метод `send()` объекта `XMLHttpRequest`:

```
xmlHttp.send(null);
```

и отправляем запрос на сервер.

На стороне сервера выполняется сценарий PHP (см. листинг 6.3). Он очень прост. Сначала мы отправляем в ответ на запрос заголовок XML-сообщения:

```
header('Content-Type: text/xml');  
echo '<?xml version="1.0" ?>';
```

Первая строка содержит важный элемент заголовка, говорящего о том, что мы передаем данные в формате XML. Далее мы отправляем инструкцию браузеру, указывающую на используемую в сообщении версию языка XML, и отправляем первый тег этого сообщения:

```
echo '<response>';
```

Остальная часть кода выполняет сборку текста ответного сообщения в формате XML. Чтобы посмотреть, что получается в результате, мы вызовем наш сценарий PHP, введя в адресную строку браузера такое значение:

```
http://localhost/ajax.php?login=
```

Ответное XML-сообщение представлено на рис. 6.3.

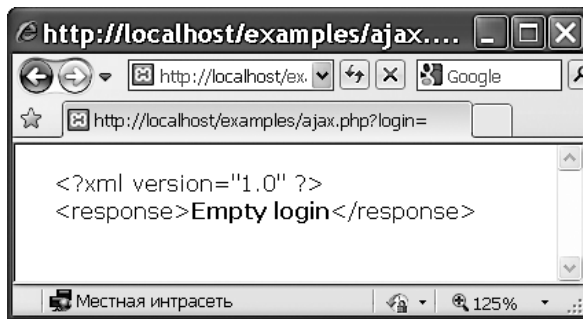


Рис. 6.3. Ответное XML-сообщение нашего сервера

Формирование этого сообщения начинается с извлечения из строки вызова сценария переданного ему параметра:

```
$login = $_GET['login'];
```

Далее мы просто сравниваем его со значениями в массиве `$userNames`, используя для этого встроенную функцию PHP `in_array()`, и, если совпадение имеется, передаем сообщение:

```
echo 'Hello, user ' . $login . '!';
```

В противном случае мы проверяем, не пустая ли строка была передана серверу. Это нелишняя проверка, поскольку именно с такого варианта начинаются запросы к серверу, когда пользователь еще не успел ввести никаких данных. Чтобы проверка была корректной, предварительно с помощью функции `trim()` из переданной переменной `trim($login)` удаляются все пробелы, которые могут находиться в начале и конце строки.

Если переданный параметр пуст, отображается сообщение о пустом поле (рис. 6.4).

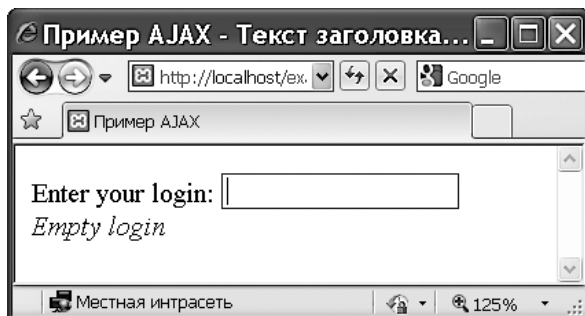


Рис. 6.4. Поле для ввода имени пустое

Если же ввести имя, известное серверу, то отображается приветствие (рис. 6.5).

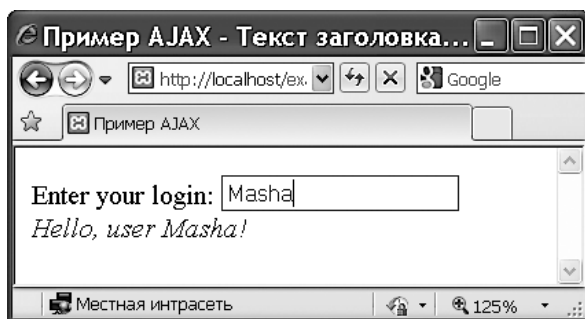


Рис. 6.5. Сервер узнал введенное имя

Но как же все эти сообщения выводятся на экране браузера? Ведь сервер передает свои сообщения в формате XML, непонятном браузеру. Ответ прост — это делает программа — обработчик прерываний:

```
function ResponseHandler() {
    if (xmlHttp.readyState == 4) {
        if (xmlHttp.status == 200) {
            xmlResponse = xmlHttp.responseXML;
            xmlDocumentElement = xmlResponse.documentElement;
            Response = xmlDocumentElement.firstChild.data;
            document.all("Message").innerHTML = '<i>' + Response + '</i>';
            setTimeout('request()', 1000);
        }
    }
}
```



```
else {  
    alert("Ошибка доступа к серверу");  
} } }
```

Этот сценарий автоматически вызывается клиентом при каждом изменении в статусе отправленного запроса AJAX. После вызова на первом шаге проверяется состояние запроса согласно значению уже упомянутого параметра `readyState` объекта `XMLHttpRequest`:

```
if (xmlHttp.readyState == 4)
```

Если запрос завершен, то проверяется другой параметр `status` объекта `XMLHttpRequest`:

```
if (xmlHttp.status == 200)
```

Значение 200 означает успешное завершение запроса. Если запрос успешно завершен, то сценарий приступает к разборке полученного сообщения XML с целью извлечения переданного ответа сервера. Для этого используются параметры объекта `XMLHttpRequest`:

- ❑ `xmlHttp.responseXML` — извлекает полученное сообщение XML;
- ❑ `xmlDocumentElement = xmlResponse.documentElement` — этот параметр извлекает корневой элемент в полученном сообщении;
- ❑ `xmlDocumentElement.firstChild.data` — извлекает содержимое корневого элемента XML.

В объекте `XMLHttpRequest` содержатся и другие средства для работы с запросами XML, но для нашего простого примера достаточно перечисленных выше.

Извлеченное сообщение сервера присваивается переменной `Response`, с помощью которой составляется ответ серверу:

```
'<i>' + Response + '</i>'
```

Наконец, результат всех этих действий отображается на экране браузера в элементе `<div id="Message" />`.

А теперь, детально разобрав код нашего веб-приложения, повторим шаги его работы по выполнению запросов AJAX.

1. Посетитель сайта загружает страницу `ajax.html` с запросом AJAX и начинает ввод данных.
2. Браузер запускает программу `request()` для генерации запроса AJAX.

3. Программа `request()` проверяет готовность объекта `XMLHttpRequest` к отправке запросов.
4. Если объект `XMLHttpRequest` не готов, назначается интервал ожидания 1 с и выполняется повторный вызов `request()`.
5. Если механизм AJAX готов к работе, то `request()` готовит запрос AJAX:
 - извлекает содержимое поля ввода и помещает его в строку вызова серверного сценария;
 - назначает программу `ResponseHandler()` обработчиком запроса AJAX.
6. Программа `request()` запускает механизм асинхронных запросов серверу.
7. Серверный сценарий PHP обрабатывает запросы и генерирует ответные сообщения XML.
8. Клиент при любом изменении состояния переданного запроса вызывает обработчик `ResponseHandler()`.
9. Если запрос выполнен успешно, то `ResponseHandler()` обрабатывает XML-сообщение и отображает ответ сервера:
 - если поле ввода пустое, в ответ отобразится сообщение о пустом логине (см. рис. 6.4);
 - если поле ввода содержит неизвестное имя, то будет выведено сообщение о незарегистрированном пользователе (см. рис. 6.1);
 - если введенное имя совпадает с известным серверу, отображается приветствие (см. рис. 6.5).
10. Обработчик назначает задержку исполнения программы `request()` в 1 с и вызывает программу по истечении срока. Все повторяется.

Таким образом, с помощью объекта `XMLHttpRequest` мы сумели передать запрос серверу, получить ответ и обработать его. Написанное нами веб-приложение реализует общепринятую последовательность действий, состоящую в создании объекта `XMLHttpRequest`, создании асинхронного запроса, назначении функции автоматической обработки этого запроса и повторном создании асинхронного запроса. Самое главное преимущество технологии AJAX состоит в том, что выполнение асинхронных запросов никак не влияет на работу пользователя, что важно при обработке больших объемов информации.

Для обмена данными между сервером и клиентом мы использовали сообщения в формате XML. Это вовсе не обязательное требование к приложениям, создаваемым на основе AJAX. Мы можем использовать любые структуры данных, в зави-

симости от решаемой задачи. Но применение XML имеет множество преимуществ, поскольку, во-первых, это позволяет структурировать данные с помощью тегов и, во-вторых, далее работать с такими данными, пользуясь множеством методов, предоставляемых AJAX. Рассмотрим это на примере.

6.2. Работа с данными XML

Напишем новое приложение, которое будет обрабатывать данные, сохраняемые в файле в формате XML. Сначала создадим сам файл данных (листинг 6.4).

Листинг 6.4. Данные о пользователях в формате XML

```
<?xml version="1.0" ?>
<response>
  <users>
    <user>
      <name>
        Petia
      </name>
      <age>
        25
      </age>
    </user>
    <user>
      <name>
        Vasia
      </name>
      <age>
        20
      </age>
    </user>
  </users>
</response>
```

Назовем файл `users.xml` и сохраним в папке `example`. Как видите, он содержит имена пользователей и их возраст в тегах `<user>...</user>`, `<name>...</name>` и `<age>...</age>`.

Теперь создадим HTML-файл страницы обращения к серверу (листинг 6.5).

Листинг 6.5. Страница запроса сервера

```
<html>
  <head>
    <title>Пример работы со структурой XML</title>
    <script type="text/javascript" src="users.js"></script>
  </head>
  <body onload="request()">
    Сервер, кто твои пользователи?
    <br/>
    <div id="DivOutput" />
  </body>
</html>
```

Здесь, кроме вопроса к серверу, содержится элемент `<div id="DivOutput" />`, в который мы будем помещать ответы сервера так, как делали это в предыдущем примере. Сохраним этот код в файле `users.html` в папке `examples`.

Наконец, создадим сценарий JavaScript для выполнения запросов к серверу (листинг 6.6).

Листинг 6.6. Сценарий запросов серверу

```
if(window.ActiveXObject)
    var xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
else
    var xmlHttp = new XMLHttpRequest();
function request() {
    if (xmlHttp)
    {
        // Пытаемся подключиться к серверу
        try
        {
            xmlHttp.open("GET", "http://localhost/examples/users.xml", true);
            xmlHttp.onreadystatechange = ResponseHandler;
            xmlHttp.send(null);
        }
    }
}
```

```

// Сообщение об ошибке подключения в случае неудачи
catch (e)
{
    alert("Ошибка подключения к серверу");
}
}
}
// Функция-обработчик запросов AJAX
function ResponseHandler() {
    if (xmlHttp.readyState == 4)
    {
        if (xmlHttp.status == 200)
        {
            var xmlResponse = xmlHttp.responseXML;
            xmlRoot = xmlResponse.documentElement;
            nameArray = xmlRoot.getElementsByTagName("name");
            ageArray = xmlRoot.getElementsByTagName("age");
            var html = "";
            for (var i=0; i<nameArray.length; i++)
                html += "<b> Пользователь: " + nameArray.item(i).firstChild.data +
                    ", Возраст: " + ageArray.item(i).firstChild.data + "
</b><br/>";
            myDiv = document.all("DivOutput");
            myDiv.innerHTML = "<i>Сервер ответил - вот они: </i><br />" +
html;
        }
        else
        {
            alert("Ошибка доступа к серверу");
        }
    }
}
}

```

Сохраним его в файле `users.js` в папке `examples`.

Откроем наш HTML-документ в окне браузера, введя в адресную строку `localhost/examples/users.html`. Сервер выдаст ответ, представленный на рис. 6.6.

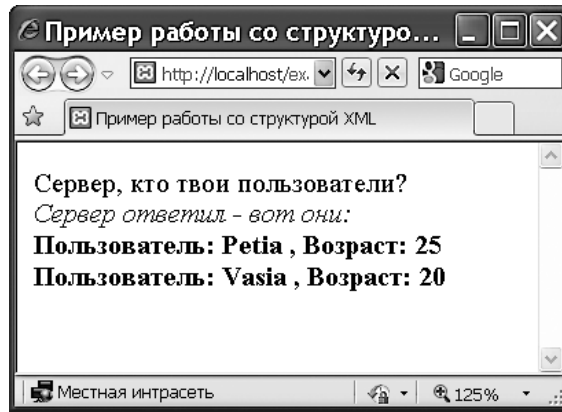


Рис. 6.6. Список пользователей сервера

Рассмотрим, как работает код сценария `users.js`. В самом начале создается объект `XMLHttpRequest`, с использованием методов, описанных в предыдущем примере. При загрузке страницы вызывается функция `request()`.

На первых шагах проверяется успешность создания объекта `XMLHttpRequest`, а по полученным результатам мы применяем механизм обработки исключительных состояний `try/catch`, предусмотренный в JavaScript. Суть его состоит в перехвате возникших ошибок (то есть исключительных состояний) и их обработке. Синтаксис конструкции `try/catch` таков:

```
try
{
    код, способный породить ошибку
}
catch
{
    код, исполняемый при возникновении ошибки
}
```

Такая конструкция очень полезна, поскольку позволяет нам контролировать ход выполнения запросов. При возникновении ошибок, вполне возможных при работе с сервером, мы сможем перехватить исключительное состояние и отобразить пользователю понятный ему и дружелюбный ответ.

В остальном программа `request()` действует аналогично предыдущему примеру: сначала готовится запрос AJAX, затем он отправляется на сервер. Серверу передается имя файла с данными XML:

```
xmlHttp.open("GET", "users.xml", true);
```

после чего назначается функция, вызываемая при любом изменении состояния запроса `ResponseHandler()`. Эта функция представляет для нас особый интерес, поскольку именно она занята обработкой полученных XML-данных.

Рассмотрим код, решающий эту задачу. Сначала выполняется обычная проверка успешного завершения запроса, после чего из запроса извлекаются переданные XML-данные:

```
var xmlResponse = xmlHttp.responseXML;
```

Эти данные сохраняются в переменной `xmlResponse`, а затем начинается разборка полученных сведений. Сначала извлекаем корневой элемент XML-структуры:

```
xmlRoot = xmlResponse.documentElement;
```

Далее с помощью методов объекта `XMLHttpRequest` извлекаем и сохраняем в массивах имена и возраст пользователей:

```
nameArray = xmlRoot.getElementsByTagName("name");
```

```
ageArray = xmlRoot.getElementsByTagName("age");
```

Теперь нам остается только собрать ответное сообщение, которое помещается в содержимое элемента `<div>` конкатенацией строк из содержимого массивов с данными по имени и возрасту.

Как видите, в этом примере не используются какие-либо серверные сценарии PHP, мы просто получаем от сервера файл данных и далее обрабатываем их клиентом.

Однако хранение данных на сервере в виде файла нельзя назвать оптимальным решением. Более удобное средство — использование баз данных, в частности MySQL. Посмотрим, как в этом случае можно применять технологию AJAX.

6.3. Работа с MySQL

Сейчас мы продемонстрируем работу с сервером MySQL средствами, включающими в себя асинхронные запросы к серверному сценарию PHP, который будет извлекать нужную информацию из базы данных MySQL и передавать ее клиенту с помощью XML-сообщений. Для простоты мы используем наше самое первое

веб-приложение с применением AJAX, описанное в разделе 6.2. Мы оставим без изменений коды из листингов 6.1 и 6.2, которые реализуют клиентскую часть приложения, и перепишем сценарий PHP, снабдив его средствами работы с MySQL.

Для начала создадим базу данных MySQL с таблицей, содержащей имена пользователей, которые в сценарии в листинге 6.3 хранились в массиве `$UserNames`: 'PETYA', 'SERGEY', 'IVAN', 'MASHA', 'LENA'. Это можно сделать запросами MySQL или из панели администрирования сервером MySQL методами, описанными в главе 3. Назовем эту базу данных `users`, таблицу — `UsrLogin`.

Теперь подготовим код нашего веб-приложения. Сначала оформляем страницу ввода входного имени пользователя (листинг 6.7).

Листинг 6.7. Страница ввода логина

```
<html>
  <head>
    <title>Пример AJAX</title>
    <script type="text/javascript" src="mysql.js"></script>
  </head>
  <body onload='request()'>
    Enter your login:
    <input type="text" id="myLogin" />
    <div id="Message" />
  </body>
</html>
```

Сохраним этот код в файле `mysql.html` в папке `examples`. Теперь напишем сценарий JavaScript для организации запросов MySQL. Он отличается от предложенного в листинге 6.2 только использованием механизма перехвата исключительных состояний (листинг 6.8).

Листинг 6.8. Сценарий отправки и обработки асинхронных запросов

```
if(window.ActiveXObject)
  var xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
else
  var xmlHttp = new XMLHttpRequest();
if (!xmlHttp)
  alert("Error creating the XMLHttpRequest object.");
```



```

function request() {
    if (xmlHttp)
    {
        // Пытаемся подключиться к серверу
        try
        {
            name = document.all("myLogin").value;
            xmlHttp.open("GET", "mysql.php?login=" + name, true);
            xmlHttp.onreadystatechange = ResponseHandler;
            xmlHttp.send(null);
        }
        // Сообщение об ошибке подключения в случае неудачи
        catch (e)
        {
            alert("Ошибка подключения к серверу");
        }
    }
}

function ResponseHandler() {
    if (xmlHttp.readyState == 4) {
        if (xmlHttp.status == 200) {
            xmlResponse = xmlHttp.responseXML;
            xmlDocumentElement = xmlResponse.documentElement;
            Response = xmlDocumentElement.firstChild.data;
            document.all("Message").innerHTML =
                '<i>' + Response + '
        </i>';
            setTimeout('request()', 1000);
        }
        else {
            alert("Ошибка доступа к серверу");
        }
    }
}

```

Здесь, как вы можете заметить, используется конструкция `try/catch` перехвата ошибок и их обработки. Сохраним этот сценарий в файле `mysql.js` в папке `example`. Теперь напишем сценарий PHP работы с базой данных MySQL (листинг 6.9).

Листинг 6.9. Серверный сценарий выбора входных имен из MySQL

```
<?php
require_once('handle_err.php');
header('Content-Type: text/xml');
echo '<?xml version="1.0" ?>';
echo '<response>';
$login = $_GET['login'];
$connection=mysql_connect("localhost", "user", "password");
$my_db=mysql_select_db("users", $connection);
$query="SELECT * FROM UsrLogin";
$result=mysql_query($query);
$userNames=mysql_fetch_array($result);
if (in_array(strtoupper($login), $userNames))
    echo 'Hello, user ' . htmlentities($login) . '!';
else if (trim($login) == '')
    echo 'Empty login';
else
    echo htmlentities($login) . ', Unregistered user';
echo '</response>';
?>
```

Сохраним этот сценарий в файле `mysql.php` в папке `example` и обратим внимание на одну деталь: в начале сценария находится такой оператор:

```
require_once('handle_err.php');
```

Он выполняет однократную загрузку файла `handle_err.php` со сценарием функции обработки ошибок PHP (листинг 6.10). Этот файл мы также создадим и запишем в папку `example`.

Листинг 6.10. Функция обработки ошибок

```
<?php
set_error_handler('handle_err');
function handle_err($errNo, $errStr, $errFile, $errLine)
{
```

```
if(ob_get_length()) ob_clean();
$message = 'Номер ошибки: ' . $errNo .
           'Сообщение: ' . $errStr .
           'Расположение: ' . $errFile .
           ', Строка ' . $errLine;
echo $message;
exit;
}
?>
```

Если теперь загрузить в браузер страницу `mysql.html`, введя в адресную строку `localhost/examples/mysql.html`, то на экране будут отображаться те же сообщения, что и на рис. 6.1 и 6.2. Однако теперь все данные о пользователях сервер извлекает из базы данных MySQL. Если потребуется, мы всегда можем дополнить наш сценарий инструментами пополнения этой базы данных, редактирования хранимых данных, добавления новых таблиц — все в наших руках. Принцип построения веб-приложения, использующего технологию AJAX для обмена данными с сервером PHP и MySQL, остается неизменным. Конечно, в нашем случае, когда мы просто извлекаем несколько имен пользователей из базы данных и отображаем их на экране, применение AJAX не так и обязательно. Но если при обращениях к базе данных вы намереваетесь извлекать информацию большого объема, причем уточнять характер получаемых данных вы будете вводом параметров в сложную форму, то применение AJAX — единственный способ оптимизировать время ожидания отклика сервера.

Но рассмотрим, как работают наши сценарии PHP. Содержимое листинга 6.9 достаточно тривиально — в самом начале мы подключаемся к серверу MySQL, после чего выбираем нашу базу данных `users` и извлекаем из таблицы `UsrLogin` сведения с помощью запроса:

```
$query="SELECT * FROM UsrLogin";
```

Далее полученная таблица обрабатывается аналогично примеру сценария в листинге 6.3. Наш сценарий проверяет, содержится ли введенное имя в списке имен, хранимых на сервере, и, в зависимости от результата, генерирует ответ в формате XML-сообщения.

Наибольший интерес представляет сценарий обработки исключительных состояний, представленный в листинге 6.10. В нем в самом начале с помощью функции `set_error_handler()` определяется пользовательский обработчик прерываний:

```
set_error_handler('handle_err');
```

Здесь `handle_err` — название пользовательской функции. Пользовательская функция должна принимать следующие параметры: код ошибки (`$errNo`), строку с описанием ошибки (`$errStr`), имя файла, в котором появилась ошибка (`$errFile`), номер строки (`$errLine`):

```
function handle_err($errNo, $errStr, $errFile, $errLine)
```

В начале работы программы с помощью функции `ob_clean()` очищается буфер вывода. Затем генерируется сообщение, собираемое конкатенацией параметров нашей функции — обработчика ошибок. По завершении работа сценария прерывается, поскольку при использовании пользовательской функции обработки исключений мы должны сами управлять остановкой работы программы.

Имейте в виду, что такая функция не в состоянии предотвратить ошибки синтаксического анализа кода, которые прерывают работу интерпретатора еще до выполнения сценария. Однако пользовательские программы обработки ошибок чрезвычайно полезны, поскольку позволяют создавать осмысленные сообщения о проблемах в работе веб-приложения.

6.4. Резюме

Итак, вы познакомились с основами технологии AJAX, научившись создавать асинхронные запросы, генерировать ответы сервера в виде XML-сообщений и обрабатывать такие сообщения средствами разборки структуры XML. Далее вы освоили средства обработки исключительных состояний с помощью конструкции `try/catch` для сценариев JavaScript и назначения пользовательских функций обработки ошибок сценариев PHP. Кроме того, было показано, как веб-приложения, созданные с использованием AJAX, могут работать с базами данных MySQL, и это проиллюстрировано на примере простого приложения.

Этим возможности AJAX не ограничиваются, вне нашего обсуждения остались многие важные вопросы. Однако теперь вы готовы писать простые сценарии, использующие асинхронные запросы, что чрезвычайно важно при создании интернет-магазинов, обеспечивающих выбор покупки из огромных баз данных.

Глава 7

Концепция объектно-ориентированного программирования

Объектно-ориентированное программирование (ООП) представляет собой следующий шаг в развитии методов разработки программ, весьма полезный при разработке сложных проектов. Большинство веб-приложений сравнительно невелики по размеру и состоят из набора страниц, связанных ссылками. Для таких приложений приемлемы методы процедурно-ориентированного программирования, которые мы обсуждали до сих пор.

В процедурно-ориентированной программе, как вы могли заметить, данные и подпрограммы их обработки никак не связаны, никаких формальных ограничений на работу подпрограмм с данными язык не накладывает. Все отдано в руки программиста, и, если проект простой, вы сможете проконтролировать работу с данными благодаря такому подходу. Но если проект усложняется, над ним работает множество людей, то начинаются проблемы, когда данные, предназначенные для работы одного сегмента приложения, подвергаются изменениям со стороны другого сегмента, который использует свои алгоритмы и методы, не обязательно совместимые.

По этим причинам и возникла концепция объектно-ориентированного программирования, которая ограничивает обработку данных определенного вида строго определенными наборами методов.

7.1. Классы и объекты

В объектно-ориентированном программировании все данные определенного вида и все операции их обработки сводятся в некое единое целое, называемое *объектом*, который включает в себя набор переменных, называемых *свойствами* (или атрибутами) для хранения данных, и функций, называемых *методами*, предназначенными для обработки данных объекта. К свойствам объекта можно получить доступ только через его методы, которые все вместе образуют *интерфейс объекта*. Такое сокрытие, или, более научно, *инкапсуляция*, данных, является основным достоинством объектно-ориентированного программирования. Действия, выполняемые объектом, распространяются только на используемые им данные, что позволяет разбить общую задачу на независимые части.

В объектно-ориентированной программе каждый объект идентифицируется отдельной переменной, с помощью которой становится возможным обращение к нему из программы. Например, в программе могут существовать два объекта, которые реализуют диалоговые окна, отличающиеся только размерами. Чтобы программа могла обратиться к этим объектам, при их создании каждому объекту присваивается уникальный идентификатор, хранящийся в отдельной переменной программы.

Объекты могут группироваться в классы, которые включают в себя объекты со сходными признаками. В этих классах содержатся объекты, у которых одинаковые методы в каждом объекте действуют одинаково, а одинаковые переменные объекта представляют одни и те же свойства. Однако каждый объект класса может иметь различный набор переменных. Скажем, класс диалоговых окон может содержать объекты, отличающиеся набором элементов интерфейса (кнопками, меню и пр.). Но методы изменения размеров и перемещения окон должны совпадать.

Таким образом, объектно-ориентированное программирование состоит в описании структуры и поведения проектируемой программы, отвечающем на вопросы:

- ☐ из каких частей состоит программа;
- ☐ какие функции реализует каждая часть.

Каждая часть должна иметь минимальный и точно определенный набор функций и быть как можно меньше связанной с остальными частями. Начав с крупных частей, нужно последовательно уточнить и выделить более мелкие объекты и т. д., составив иерархическую структуру классов. После этого каждый компонент может реализовываться независимо от остальных.

Объектно-ориентированный язык программирования должен поддерживать *полиморфизм*, который представляет собой возможность исполнять одну и ту же операцию в различных классах по-разному. Например, в классе геометрических фигур объекты эллипс, круг могут подвергаться деформации одной и той же функцией, переопределяемой в зависимости от конкретной ситуации, что значительно упрощает написание кода.

Другая отличительная особенность объектно-ориентированной программы — *наследование*, которое позволяет создавать из одного класса, называемого *родительским*, другой подкласс — *дочерний*, образуя иерархические взаимосвязи между ними. Дочерний класс наследует свойства и методы своего родительского класса. Используя наследование, можно расширять и дополнять существующие классы. По мере необходимости из простого базового класса можно получать сложные и специализированные производные классы. Это делает код более пригодным для повторного использования, что является одним из важных преимуществ объектно-ориентированного подхода.

Создание классов, свойств и методов

Теперь перейдем к обсуждению практической реализации всего вышесказанного в языке РНР. Для создания класса в РНР используется ключевое слово `class`. Вот как выполняется объявление класса:

```
class имя_класса
```

Запись *имя_класса* обозначает имя создаваемого класса.

Каждый класс должен иметь методы и свойства. Свойства определяются внутри класса с помощью ключевого слова `var`:

```
class MyClass
{
var $parameter1;
var $parameter2;
}
```

А сейчас добавим в класс методы:

```
class MyClass
{
    var $parameter1;
    var $parameter2;

    function metod1($parameter2)
    {
    }

    function metod2 ($parameter1, $parameter2)
    {
    }
}
```

Здесь в классе `MyClass` создаются два метода `metod1` и `metod2`, которые принимают один и два параметра соответственно.

В большинстве классов используются особые методы, называемые *конструкторами*. Они автоматически исполняются при создании класса и служат для задания начальных значений свойствам класса. В PHP 5 конструктор объявляется так же, как другие методы, но с именем `_construct()`:

```
class MyClass
{
    function _construct ($param)
    {
        $param=10;
    }
}
```



ПРИМЕЧАНИЕ

В предыдущих версиях языка (PHP 4 и старше) для объявления конструктора использовали функцию-метод с именем самого класса. В версии PHP 5 для обеспечения обратной совместимости при отсутствии объявления конструктора с помощью специального имени `_construct()` выполняется поиск функции с именем класса, которая будет использована как конструктор.

Если по ходу исполнения программы все ссылки на объект — экземпляр класса удаляются или выходят из области видимости, то автоматически выполняется функция — *деструктор*, которая по своим действиям противоположна конструктору и отменяет все сделанные им операции.

Создание объектов — экземпляров класса

Чтобы начать работу со средствами, предоставляемыми классом, следует создать объект, представляющий собой конкретную реализацию свойств и методов класса, так называемый экземпляр класса. Образно говоря, класс можно сравнить с неким шаблоном, штампом, «применением» которого к заготовке «штампуется», то есть создается, конкретная «реализация» шаблона — нужная вам деталь. Так и в программе, при создании объекта в области памяти формируется структура данных, хранящая свойства и методы класса, к которым вы можете обращаться из программы. Как и для механического штампа, для каждого класса может быть создано множество «деталей» — экземпляров класса, независимых друг от друга. Вы сможете работать с каждым из созданных объектов по отдельности, не затрагивая остальные.

Для создания объекта используется ключевое слово `new`, в котором указывается класс, на основе которого создается объект, и параметры исполнения конструктора класса. Вот пример создания трех объектов одного и того же класса `MyClass`:

```
class MyClass
{
    function MyClass($param)
    {
        echo "<p> Конструктору класса передано значение параметра $param
</p>";
    }
}

$a = new MyClass ("Class_1") ;
$b = new MyClass ("Class_2") ;
$c = new MyClass ("Class_3") ;
```

При каждой операции вызова класса автоматически вызывается конструктор класса `MyClass`, которому передается значение параметра `$param`: `Class_1`, `Class_2` и `Class_3` соответственно, которые будут отображаться при каждом исполнении функции.

Определения функции могут включить указание типа класса, передаваемого в качестве параметра. Если функция будет вызвана с неправильным типом, то произойдет ошибка.

```
function receiveMyClass(MyClass $object)
{
}
```

Функция может также возвращать объект.

Обращение к свойствам и методам

Итак, у нас есть класс, содержащий свойства и методы — программы, которые обрабатывают данные, хранимые в свойствах. Чтобы обратиться к свойству класса из метода этого класса, следует использовать специальную переменную `$this` следующим образом:

```
class MyClass
{
    var $myclass_parameter;
    function MyClassFunction ($param)
    {
        $this->myclass_parameter = $param;
        echo $this->myclass_parameter;
    }
}
```

Здесь свойству `$myclass_parameter` класса `MyClass` внутри функции `MyClassFunction` присваивается значение параметра функции `$param`, что указывается в записи `$this->myclass_parameter`.

Доступ к свойствам и методам класса из кода вне класса в РНР 5 определяется специальными модификаторами доступа:

- ❑ `public` (общедоступны) — открывает доступ к свойствам и методам класса;
- ❑ `private` (приватный) — разрешает доступ к свойствам и методам класса только изнутри и запрещает наследование;
- ❑ `protected` (защищенный) — разрешает доступ к свойствам и методам класса только изнутри и разрешает наследование.

Что такое *наследование*, мы обсудим в следующем разделе, а сейчас просто запомните, что по умолчанию всем свойствам класса присваивается модификатор `public`, после чего к его свойствам и методам можно обращаться таким образом:

```
class MyClass
{
    public $myclass_parameter;
}

$object = new MyClass ();
$object->myclass_parameter = "value";
```

В этом коде мы сначала создали объект класса `MyClass` и обратились к свойству объекта, записав:

```
$object->myclass_parameter
```

Тем самым мы изменили свойство класса извне кода класса.

Для обращения к методам класса используются очень похожие конструкции языка. Например, пусть у нас имеется класс:

```
class MyClass
{
    function MyClassFunction ($param);
    {
    }
}
```

Тогда для вызова функции `MyClassFunction` извне класса следует писать так:

```
$MyClassObject = new MyClass();
$MyClassObject->MyClassFunction("value");
```

Как видите, запись почти аналогична обращению к свойству класса, только вместо имени свойства используется название метода класса.

Для получения данных, возвращаемых методом класса, следует писать так:

```
$return_value=$MyClassObject->MyClassFunction("value");
```

Таким образом, вы поняли, как в PHP реализуется работа со свойствами и методами класса. Но этим возможности PHP отнюдь не исчерпываются.

7.2. Реализация наследования в PHP

Из классов PHP можно строить иерархические структуры, пользуясь для их создания механизмом наследования, который мы сейчас и обсудим. Класс-потомок наследует все методы и свойства класса-родителя плюс добавляет к ним свои, расширяя возможности базового класса. Это очень полезное свойство при построении сложно структурированных данных, которое позволяет избежать множества проблем и ошибок.

Допустим, у нас имеется такой класс:

```
class ClassParent
```

```

{
    var $ClassParent_parameter ;
    function ClassParentFunction()
    {
    }
}

```

Чтобы создать класс `ClassChild`, наследующий другой класс `ClassParent`, следует написать такой код:

```

class ClassChild extends ClassParent
{
    var $ClassChild_parameter;
    function ClassChildFunction();
    {
    }
}

```

Ключевое слово `extend` означает, что класс `ClassChild` расширяет класс `ClassParent`, является его *потомком*, или *дочерним* классом. Соответственно `ClassParent` называется *родительским* классом. К методам и свойствам родительского и дочернего классов можно обращаться так:

```

$ObjectChild = new ClassChild();
$ObjectChild->ClassChildFunction();
$ObjectChild->ClassChild_parameter = 100;
$ObjectChild->ClassParentFunction();
$ObjectChild->ClassParent_parameter = 30;

```

Заметьте, что к свойствам и методам родительского класса мы обращаемся так же, как и к свойствам и методам его потомка. В этом и заключается преимущество наследования, при котором потомок получает в свое распоряжение все возможности родительского класса и дополняет их своими средствами. Однако обратное наследование недопустимо. В приведенном ниже примере два последних оператора *некорректны*:

```

$ObjectParent = new ObjectParent();
$ObjectParent->ClassChildFunction();
$ObjectParent->ClassChild_parameter = 100;

```

Таким образом, класс-родитель *не имеет* доступа к возможностям класса-потомка. Да, но что, если мы в классе-потомке определим свойства и методы с именами,

совпадающими с использованными в родительском классе? Такая возможность существует и определяется механизмом PHP, называемым *перекрытием*, использование которого бывает весьма полезным.

Перекрытие

Операция перекрытия применяется в том случае, если в программе требуется модифицировать какие-то характеристики свойств и методов класса, изменить, скажем, значение свойства по умолчанию или дополнить метод новыми функциональными возможностями. Проиллюстрируем сказанное примером.

Пусть у нас имеется класс `ClassA` такого вида:

```
class ClassA
{
    var $ClassA_parameter = 10;
    function ClassA_function()
    {
        echo "Значение по умолчанию \$ClassA_parameter равно $this->ClassA_
parameter<br>" ;
    }
}
```

В этом классе свойству `$ClassA_parameter` присваивается значение по умолчанию 10, которое и отображается при обращении к методу класса `ClassA_function`.

Если по ходу исполнения программы нам потребуется, чтобы значением по умолчанию этого свойства было другое число, скажем 100, то мы можем его *перекрыть*, создав класс-потомок, функции и методы которого перекрывают родительский:

```
class ClassB extend ClassA
{
    var $ClassA_parameter = 100;
    function ClassA_function()
    {
        echo "Новое значение по умолчанию \$ClassA_parameter равно $this->
ClassA_parameter<br>" $
    }
}
```

Как видите, новый класс содержит свойство и метод с именами, совпадающими с родительским классом, перекрывающими их, но имеющими новое значение по умолчанию и новый код метода класса соответственно. Если теперь написать такой код:

```
$ObjectA = new ClassA();  
$ObjectA->ClassA_function();
```

то при его исполнении будет выведено прежнее значение по умолчанию свойства `$ClassA_parameter` — 10:

Значение по умолчанию `$ClassA_parameter` равно 10

Таким образом, на методы и свойства класса-родителя объявление нового класса-потомка не повлияло. Но если написать так:

```
$ObjectB = new ClassB();  
$ObjectB->ClassA_function();
```

то отобразится такая строка:

Новое значение по умолчанию `$ClassA_parameter` равно 100

Таким образом, в классе-потомке все функции родительского класса перекрыты, при обращении к его свойствам и методам срабатывает перекрытие и класс-потомок получает доступ к новым возможностям.

Наследование может быть многоуровневым, каждый дочерний класс наследует свойства и методы родительского класса и перекрывает их при необходимости. Но РНР не поддерживает *множественное наследование*, то есть каждый класс может наследовать свойства и методы только одного-единственного родительского класса и количество классов-потомков одного родителя не ограничено.

Иногда бывает необходимо обратиться к перекрытым свойствам и методам класса-родителя. Для этого используют такую запись:

```
ClassA::ClassA_parameter
```

Здесь префикс `ClassA::` указывает на обращение к перекрытому свойству.

Статические методы класса

Определения классов могут теперь включить статические методы, доступ к которым не требует создания экземпляра класса, а реализуется непосредственно через класс. Вы можете определить какой-либо метод класса статическим, используя

ключевое слово `static`, и далее обращаться к нему напрямую. Вот пример объявления статического метода класса и обращения к нему:

```
class MyClass {
    static function static_function() {
        print "Это статическая функция";
    }
}
MyClass::static_function();
```

Вы можете теперь определить методы как статические, разрешая им быть вызванными без создания объекта — экземпляра класса. К статическим методам нельзя обращаться через переменную `$this`, поскольку для обращения к ним не требуется создавать объект — экземпляр класса, то есть ссылка на объект может отсутствовать.

Константы класса

В определения классов можно включить константы и ссылаться на них, используя объект, как вне класса, так и в его пределах. При объявлении констант и обращениях к ним не используется символ `$`. Как и свойства и методы, значения констант, объявленных внутри класса, не могут быть получены через переменную, содержащую экземпляр этого класса.

```
class MyClass {
    const TEMPERATURE = 36.6;
}
echo 'Нормальная температура = '.MyClass::TEMPERATURE."\n";
```

Доступ к константам выполняется с помощью префикса *ИмяКласса::* и не требует создания экземпляра класса.

Обращение к элементам классов

Используя оператор `::`, можно обращаться к константам, статическим или перекрытым свойствам или методам класса. При обращении к этим элементам извне класса программист должен указывать имя класса.

Оператор `::` вне объявления класса применяется таким образом:

```
<?php
```

```
class MyClass {  
    const CONST_VALUE = 'Значение константы';  
}  
echo MyClass::CONST_VALUE;  
?>
```

Для обращения к свойствам и методам в объявлении класса задаются ключевые слова `self` и `parent` вместе с оператором `::`, например, так:

```
<?php  
class OtherClass extends MyClass {  
    public static $my_static = 'статическая переменная';  
    public static function doubleColon() {  
        echo parent::CONST_VALUE . "\n";  
        echo self::$my_static . "\n";  
    }  
}  
OtherClass::doubleColon();  
?>
```

Когда дочерний класс перекрывает методы родительского класса, а вам требуется обратиться к методу в родительском классе, нужно написать так:

```
<?php  
class MyClass {  
    protected function myFunc() {  
        echo "MyClass::myFunc()\n";  
    }  
}  
class OtherClass extends MyClass {  
    public function myFunc() {  
        parent::myFunc();  
        echo "OtherClass::myFunc()\n";  
    }  
}  
$class = new OtherClass();  
$class->myFunc();  
?>
```


В сценарии выполняется перекрытие родительских методов, но будет вызван родительский метод и отобразится строка:

```
MyClass::myFunc() OtherClass::myFunc()
```

Проверка типа объекта

В PHP поддерживается проверка типа объекта на предмет того, является он экземпляром интересующего вас класса или унаследован от указанного класса. Для этого используется специальная конструкция языка с ключевым словом `instanceof`:

```
class Square {  
    ...  
}  
$myobject = new Square();  
if ($myobject instanceof of Square) {  
    echo '$myobject является квадратом';  
}
```

Здесь создается объект `myobject` — экземпляр класса `Square`, и далее мы проверяем его происхождение операцией сравнения (`$myobject instanceof of Square`). При подтверждении значение этого выражения будет `true`, иначе — `false`.

Клонирование объекта

Язык PHP поддерживает копирование, или, как чаще говорят, *клонирование*, существующих объектов. Клон объекта создается при указании ключевого слова `clone`:

```
$clone_myobject = clone $myobject;
```

При создании клона объекта средства PHP определяют, был ли для класса этого объекта объявлен метод `_clone()`.

- ❑ Если НЕТ, то при создании клона будет вызван метод `_clone()`, объявленный по умолчанию и выполняющий копирование всех свойств объекта.
- ❑ Если ДА — будет выполнен объявленный в классе метод `_clone()`.

Вот пример использования `_clone()` при клонировании объекта:

```
class MyClass {
```

```
function _clone() {  
    print "Объект был клонирован ";  
}  
  
$myobject = new MyClass();  
$clone_myobject = clone $myobject;
```

В функцию `_clone()` обычно включают средства проверки корректности клонирования для свойств класса и его ссылок на другие объекты (если это необходимо). Или обновляют базу данных на основе информации в копируемом объекте.

7.3. Абстрактные классы

Язык PHP 5 поддерживает абстрактные классы и методы, для которых нельзя создавать объекты — экземпляры класса. Абстрактные классы и методы не включают в себя какие-то функциональные средства и служат лишь для описательных целей. Однако абстрактные классы могут наследоваться, и их используют при создании сложных иерархических конструкций из классов. В этом случае абстрактные методы перекрываются дочерними классами, получая конкретное функциональное наполнение. Класс, в котором объявлен хотя бы один абстрактный метод, должен также быть объявлен абстрактным.

Для объявления абстрактного класса или метода используют ключевое слово `abstract`:

```
abstract class MyAbstractClass {  
    abstract protected function getValue();  
    public function print() {  
        print $this->getValue();  
    }  
}  
  
class MyClass1 extends MyAbstractClass {  
    protected function getValue() {  
        return "MyClass1";  
    }  
}  
  
class MyClass2 extends MyAbstractClass {  
    protected function getValue() {
```

```
        return "MyClass2";
    }
}
$class1 = new MyClass1;
$class1->print();
$class2 = new MyClass2;
$class2->print();
```

При исполнении сценария отобразится следующее:

```
MyClass1
MyClass2
```

Как видите, метод, объявленный как `abstract`, далее определяется дочерними классами, получая функциональные средства классов-потомков.

Интерфейсы

Интерфейсы объектов позволяют программисту создавать код, который указывает, какие методы и свойства должен включать класс, без необходимости описания их функционала.

Интерфейсы объявляются так же, как и обычные классы, но с использованием ключевого слова `interface`; тела методов интерфейсов должны быть пустыми. Для включения интерфейса в класс программист должен использовать ключевое слово `implements` и реализовать методы во включаемом интерфейсе. При необходимости классы могут включать более одного интерфейса, тогда их перечисляют через пробел. Пример интерфейса:

```
interface Template
{
    function setTemplate();
    function getTemplate();
}
class MyTemplate implements Template
{
    function setTemplate()
    {
        // ...
    }
}
```

```
function getTemplate()
{
// ...
}
}
```

Здесь класс `MyTemplate` реализует функционал шаблонного интерфейса `Template`, который содержит метод `setTemplate()` применения шаблона к некоей заготовке веб-страницы и метод `getTemplate()` получения готового к применению кода. Это может быть, например, шаблон дизайна веб-страницы, который можно изменить заданием нового шаблона. Такие операции широко распространены в инструментальных средствах веб-дизайна, и использование интерфейсов позволяет разработчикам заранее, до конкретной реализации, определять функционал создаваемых средств. По мере разработки каждый интерфейс реализуется в определенном классе, и, если метод интерфейса не получает конкретного наполнения, отображается фатальная ошибка.

Предотвращение перекрытия — `final`

Ключевое слово `final` позволяет пометить методы, чтобы наследующий класс не мог перекрыть их. Разместив перед объявлениями методов или свойств класса ключевое слово `final`, вы можете предотвратить их переопределение в дочерних классах, например:

```
class MyClass {
    public function funct() {
        echo "Вызван метод MyClass::FirstFunc()\n";
    }
    final public function SecondFunc() {
        echo "Вызван метод MyClass::SecondFunc()\n";
    }
}
class ChildClass extends MyClass {
    public function SecondFunc() {
        echo "Вызван метод ChildClass::SecondFunc()\n";
    }
}
```

Как видите, здесь дочерний класс пытается перекрыть метод родительского класса `SecondFunc()`, помеченного `final`. Выполнение заканчивается фатальной ошибкой:

Fatal error: Cannot override final method MyClass::SecondFunct()

Здесь сообщается, что финальный метод `MyClass::SecondFunct()` не может быть перекрыт.

Итераторы

PHP 5 предоставляет механизм итераторов для получения списка всех свойств какого-либо объекта, например, для использования совместно с оператором `foreach`. По умолчанию в итерации будут участвовать все свойства, объявленные как `public`. Пример использования итераторов:

```
class MyClass {
    public $var1 = 'public 1';
    public $var2 = 'public 2';
    public $var3 = 'public 3';
    protected $protected = 'protected';
    private $private = 'private';
}
$MyObject = new MyClass();
foreach($MyObject as $key => $value) {
    echo "$key => $value". "<br/>";
}
```

Результат исполнения сценария таков:

```
var1 => public 1
var2 => public 2
var3 => public 3
```

Таким образом, оператор `foreach` извлек все принадлежащие объекту свойства типа `public`.

7.4. Функции для работы с классами и объектами

В PHP существует множество стандартных функций для работы с классами и объектами. Здесь мы рассмотрим основные из них, а для ознакомления с полным перечнем таких функций вы можете обратиться к справочнику функций PHP.

get_class_methods()

Функция `get_class_methods()` возвращает массив имен общедоступных (типа `public`) методов указанного ей класса. Синтаксис функции таков:

```
array get_class_methods (string имя_класса)
```

Вот пример ее использования для получения списка методов класса:

```
class Auto {
    var $model;
    var $speed;
    var $weight;
    protected function setModel($model) {
        $this->model = $model;
    }
    private function setSpeed($speed) {
        $this->speed = $speed;
    }
    public function setWeight($weight) {
        $this->weight = $weight;
    }
}

$class_methods = get_class_methods("Auto");
foreach($class_methods as $value)
    echo "$value <br/>";
```

После исполнения сценария отобразится:

```
setWeight
```

Как видите, функция `get_class_methods()` извлекла имя единственного общедоступного метода класса `Auto`.

get_class_vars()

Функция `get_class_vars()` возвращает массив имен общедоступных (типа `public`) свойств указанного класса. Синтаксис функции таков:

```
array get_class_vars (string имя_класса)
```

Приведу пример использования функции `get_class_vars()`:

```
<?php
class Auto {
private $model;
var $speed;
}
class MiniCars extends Auto {
protected $weight;
}
$MyClass = "MiniCars";
$attrs = get_class_vars($MyClass);
while (list($value) = each($attrs)):
print "$value <br>";
endwhile;
?>
```

В результате отобразится:

speed

Таким образом, в рассмотренном примере массив `$attrs` содержит имя единственного общедоступного свойства `$speed` класса `MiniCars`.

get_object_vars()

Функция `get_object_vars()` возвращает ассоциативный массив с информацией обо всех атрибутах объекта с заданным именем. Синтаксис функции `get_object_vars()`:

`array get_object_vars (object имя_объекта)`

Пример использования функции `get_object_vars()`:

```
<?php
class Auto {
var $speed;
var $weight;
}
class Cars extends Auto {
var $wheels;
}
```

```

class MiniCars extends Cars {
    var $doors;
    function MiniCars($speed, $weight, $wheels, $doors) {
        $this->speed = $speed;
        $this->weight = $weight;
        $this->wheels = $wheels;
        $this->doors = $doors;
    }
    function get_weight() {
        return $this->weight;
    }
}

$lada = new MiniCars(159,1000,4,4);
$attribs = get_object_vars($lada);
while (list($key, $value) = each($attribs)) :
    print "$key => $value <br>";
endwhile;
?>

```

В результате исполнения этого сценария отобразится:

```

doors => 4
wheels => 4
speed => 159
weight => 1000

```

Функция `get_object_vars()` позволяет быстро получить всю информацию о свойствах конкретного объекта и их значениях в виде ассоциативного массива.

method_exists()

Функция `method_exists()` проверяет, имеется ли в указанном объекте метод с определенным именем, и при его наличии возвращает `true`, в противном случае — `false`. Синтаксис функции таков:

```
bool method_exists (object имя_объекта, string имя_метода)
```

Пример использования метода `method_exists()`:

```
<?php
```



```
class Auto {
// ...
}
class MiniCars extends Auto {
var $fourDoors;
function setFourDoors() {
$this->fourDoors = 1;
}
}
$car = new MiniCars;
if (method_exists($car, "setFourDoors")) :
print "Это 4-дверное авто";
else :
print " Это не 4-дверное авто";
endif;
?>
```

В рассмотренном примере функция `method_exists()` проверяет, поддерживается ли объектом `$car` метод с именем `setFourDoore()`. Если метод поддерживается, то функция возвращает `true` и выводит соответствующее сообщение. В противном случае возвращается `false` и выводится другое сообщение.

get_class()

Функция `get_class()` возвращает имя класса, к которому относится объект с указанным именем. Синтаксис функции:

```
string get_class(object имя_объекта);
```

Приведем пример использования функции `get_class()`:

```
<?php
class Auto {
}
class MiniCars extends Auto {
}
$car = new MiniCars;
$MyClass = get_class($car);
echo $MyClass;
?>
```

В этом сценарии переменной `$MyClass` присваивается имя класса, на основе которого был создан объект `$car`. В результате исполнения сценария отобразится `MiniCars`.

get_parent_class()

Функция `get_parent_class()` возвращает имя родительского класса (если он есть) для объекта с указанным именем. Синтаксис функции:

```
string get_parent_class (object имя_объекта);
```

Пример использования функции `get_parent_class()`:

```
<?php
class Auto {
//...
}
class MiniCars extends Auto {
//...
}
$car = new MiniCars;
$parent = get_parent_class($car);
echo $parent;
?>
```

Здесь создается объект `$car` и при вызове `get_parent_class()` переменной `$parent` будет присвоена строка "Auto", которая и отобразится при исполнении сценария.

is_subclass_of()

Функция `is_subclass_of()` проверяет, был ли объект создан на базе класса, имеющего родительский класс с заданным именем. Функция возвращает `true`, если проверка дает положительный результат, и `false` в противном случае. Синтаксис функции:

```
bool is_subclass_of (object объект, string имя_класса)
```

Пример использования функции `is_subclass_of()`:

```
<?php
```

```
class Auto {  
    //...  
}  
class MiniCars extends Auto {  
    //...  
}  
$car = new MiniCars;  
$MySubClass = is_subclass_of($car, "Auto");  
echo $MySubClass;  
?>
```

В рассмотренном примере переменной `$is_subclass()` присваивается признак того, принадлежит ли объект `$car` потомку родительского класса `Auto`. В приведенном фрагменте `$car` относится к классу `Auto`; следовательно, `$is_subclass()` будет присвоено значение `true` (отобразится 1).

7.5. Обработка исключительных ситуаций

Приведу пример использования средств ООП в сценариях. Во время работы любой программы при исполнении различных операций могут появляться ошибки, связанные с возникновением исключительной ситуации. Например, может быть выполнена попытка деления на нуль или открытия несуществующего файла. Корректно написанный сценарий должен предусмотреть все подобные ситуации и реагировать на их появление так, чтобы, во-первых, работа программы не была нарушена и, во-вторых, ее пользователь получил уведомление о встреченной ошибке (если это требуется), причем в понятной для неспециалиста форме.

Все это не так просто сделать, как может показаться. Например, если программа во время работы вызывает цепочку подпрограмм, каждая из которых открывает файлы или базы данных, считывает нужную информацию, выполняет ее обработку, то внезапное прерывание ее работы требует грамотного завершения. Следует прервать текущую операцию, вернуться обратно в цепочке вызванных подпрограмм, закрыть открытые ресурсы, сохранить результаты обработки и завершить работу программы так, чтобы не возникли проблемы при следующем ее запуске. Для этого в РНР реализован общепринятый метод обработки исключительных ситуаций, использующий конструкцию языка, называемую блоком `try/throw/catch`, и класс исключений `Exception`.

Благодаря им реализуется генерация исключения с помощью оператора `throw` и его перехват оператором `catch`. Сам код программы, генерирующий исключение, помещается в блок `try`. Все генерируемые кодом внутри блока исключения «ловятся» операторами `catch`, расположенными сразу после блока `try`. Вот пример такой конструкции.

```
<?php
function invert($val)
{
    if (!$val)
    {
        throw new Exception('Деление на ноль.');
```

```
    }
    else
        return 1/$val;
}
try
{
    echo invert(4) . "<br>";
    echo invert(0) . "<br>";
}
catch (Exception $e) {
    echo 'Сгенерировано исключение: ', $e->getMessage(), "<br>";
}
echo 'Здравствуй, мир!';
?>
```

В этом примере мы поместили в блок `try` функцию `invert()`, которая делает всего одну операцию — вычисляет обратное значение переданного ей параметра `$var` и, если значение `$var` равно нулю, генерирует исключение с помощью такого оператора:

```
throw new Exception('Деление на ноль.');
```

В этом операторе объявляется объект класса `Exception`, конструктору которого передается параметр с текстом сообщения об ошибке. Сгенерированное этим оператором исключение перехватывается в блоке `catch (Exception $e)`, которое в качестве параметра получает созданный при этом объект класса `Exception`. Он и выдает сообщение о возникновении исключительной ситуации.

```
echo 'Сгенерировано исключение: ', $e->getMessage(), "<br>";
```

Здесь `$e->getMessage()` вызывает метод класса `Exception`, который возвращает текст сообщения, полученного экземпляром класса `Exception` при его создании оператором `throw`.

При исполнении кода отобразится результат:

```
0.25
```

```
Сгенерировано исключение: Деление на ноль.
```

```
Здравствуй, мир!
```

Каждый блок `try` должен иметь как минимум один соответствующий ему блок `catch`, а каждый сгенерированный исключительной ситуацией объект должен принадлежать классу `Exception` либо его расширению. В противном случае отобразится сообщение об ошибке. При генерации исключения конструктору объекта передается два необязательных параметра — сообщение об ошибке и ее код (или номер):

```
throw new Exception(['Сообщение_об_ошибке'][, Код_ошибки]);
```

Исполнение кода блока `try/throw/catch` подчиняется следующим правилам.

- ❑ Если исполнение кода в блоке `try` пройдет нормально, то будет продолжено выполнение программы сразу за последним блоком `catch`.
- ❑ Если в блоке `try` будет сгенерировано исключение, то все расположенные после `throw` операторы не будут исполнены и РНР предпримет попытку найти блок `catch`, в котором объявлен класс, совпадающий с классом сгенерированного исключения.
- ❑ Если такой блок `catch` не будет найден, отобразится сообщение об ошибке.
- ❑ Исключения также могут быть сгенерированы (или вызваны еще раз) оператором `throw` внутри блока `catch`.
- ❑ В РНР новейшей на момент написания книги версии 5.5 стало возможным использование блока `finally` после блока `catch`. Перед тем как продолжится выполнение программы, всегда будет исполняться код в блоке `finally`, независимо от того, было сгенерировано исключение или нет.
- ❑ Генерируемый исключением объект должен принадлежать классу `Exception` или наследоваться от `Exception`. Попытка сгенерировать исключение другого класса приведет к неисправимой ошибке.
- ❑ Блоки `try/throw/catch` можно вкладывать друг в друга.

Рассмотрим более сложный пример генерирования и перехвата исключения.

```
<?php
class MyException extends Exception { }
class Test {
    public function testing() {
        try {
            try {
                throw new MyException('Ошибка!');
            }
            catch (MyException $e) {
                /* повторная генерация исключения */
                throw $e;
            }
        }
        catch (Exception $e) {
            echo $e->getMessage();
        }
    }
}

$error = new Test;
$error->testing();
?>
```

Здесь определяется расширение класса исключения `MyException`, а блок `try` с этим исключением вложен в другой, внешний блок `try`. При исполнении кода сгенерированное во внутреннем блоке `try` исключение перехватывается в блоке `catch`, который повторно генерирует исключение, перехватываемое внешним блоком `try`. В результате отобразится сообщение "Ошибка!".

Остановимся на использовании ресурсов класса `Exception`. Очевидно, что преимущество таких конструкций `try/throw/catch` как раз и состоит в возможности определить, какое нарушение нормального хода работы программы произошло, что случилось в программе, и сообщить детали произошедшего события пользователю в понятной форме. Для этого в классе `Exception` содержатся такие методы:

- ❑ `getCode()` — возвращает код ошибки, заданный конструктором объекта исключения;
- ❑ `getMessage()` — выводит сообщение об ошибке, заданное конструктором объекта исключения;

- ❑ `getFile()` — возвращает полный путь к файлу программы, сгенерировавшей исключение;
- ❑ `getLine()` — выводит номер строки в файле программы, в которой была сгенерирована ошибка;
- ❑ `getTrace()` — возвращает массив с полными данными о местонахождении источника исключения;
- ❑ `getTraceAsString()` — выдает ту же информацию, что и `getTrace()`, но в виде строки наподобие следующей:

Ошибка!#0 D:\xampp\htdocs\examples\hello.php(129): Test->testing() #1 {main}

- ❑ `__toString()` — отображает всю информацию в объекте исключения, включая все, что выводят перечисленные выше методы:

Ошибка!exception 'MyException' with message 'Ошибка!' in D:\xampp\htdocs\examples\hello.php:116 Stack trace: #0 D:\xampp\htdocs\examples\hello.php(129): Test->testing() #1 {main}

Чтобы использовать механизм генерирования исключений в полной мере, следует прибегать к механизму перекрытия для определения собственных методов класса исключений. Это позволит максимально точно устанавливать источник ошибки и, если потребуется, выдавать понятные пользователю сообщения.

7.6. Резюме

Итак, в этой главе вы познакомились с основными средствами PHP, поддерживающими методы объектно-ориентированного программирования. Вы научились создавать классы, расширять их и перекрывать свойства и методы, освоили функции PHP для работы с классами. Основное предназначение классов состоит в инкапсуляции данных и возможности строить на их основе сложные приложения, состоящие из многих частей, каждая из которых использует определенный набор данных без рисков нарушения или повреждения информации, предназначенной для остальной части приложения.

В классах PHP содержится много полезного. На официальном сайте PHP (php.net) предоставляется доступ к открытому коду PEAR (*PHP Extension and Application Repository — Репозиторий приложений и модулей PHP*), представляющему собой структурированную библиотеку кода PHP. В ней содержится множество полезных программ, так что, если вы хотите решить какую-либо задачу, сначала поинтересуйтесь, нет ли готового решения в PEAR. Легче всего это сделать на сайте PEAR по адресу pear.php.net, где представлены каталог готовых пакетов PEAR и указания по их установке.

Жадаев Александр Геннадьевич
РНР для начинающих

Заведующий редакцией
Ведущий редактор
Художник
Корректор
Верстка

*Д. Виноцкий
Н. Гринчик
Л. Адуевская
Е. Павлович
О. Богданович*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Подписано в печать 23.09.13. Формат 70×100/16. Усл. п. л. 23,220. Тираж 2000. Заказ 0000.

Отпечатано в полном соответствии с качеством предоставленных издательством материалов
в ГППО «Псковская областная типография». 180004, Псков, ул. Ротная, 34.