АНДРЕЙ ОВЧАРЕНКО

Ajax

НА ПРИМЕРАХ

ОСНОВЫ JavaScript, HTML, CSS И DOM

ОБМЕН ДАННЫМИ С СЕРВЕРОМ В ФОРМАТЕ XML И JSON

РАБОТА С APACHE, PHP И MySQL

ЗАЩИТА ПРИЛОЖЕНИЙ, АУТЕНТИФИКАЦИЯ И АВТОРИЗАЦИЯ

СООТВЕТСТВИЕ СТАНДАРТАМ И КРОССБРАУЗЕРНОСТЬ

+CD

bhv®

Андрей Овчаренко

Ajax
НА ПРИМЕРАХ

Санкт-Петербург

«БХВ-Петербург»

2009

УДК 681.3.068
ББК 32.973.26-018.1
О-35

Овчаренко А. В.

О-35 Ajax на примерах. — СПб.: БХВ-Петербург, 2009. — 432 с.: ил. + CD-ROM

ISBN 978-5-9775-0299-3

На практических примерах рассмотрены эффективные приемы разработки динамических Web-приложений, построенных по технологии Ajax. Каждая глава посвящена разработке законченного компонента пользовательского интерфейса Web-приложения. Даны необходимые для быстрого старта сведения по HTML и CSS, XML и DOM Level 1, PHP и MySQL, а также примеры совместного их применения. Большое внимание уделено программированию на языке JavaScript и асинхронному обмену данными между клиентом и сервером при помощи объекта XMLHttpRequest в формате XML и JSON. Подробно описан процесс разработки компонентов пользовательского интерфейса: "Аккордеон", панель с закладками, слайд-шоу, выпадающее меню, плавающие окна и др. Рассмотрены вопросы доступа к базам данных MySQL, управления учетными записями пользователей, защиты данных, аутентификации и авторизации, кроссбраузерности разрабатываемых приложений и др. На прилагаемом к книге компакт-диске содержится полный программный код компонентов и приложений, рассмотренных в книге.

Для Web-программистов

УДК 681.3.068
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.09.08.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 34,83.
Тираж 2500 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0299-3

Оглавление

Введение

Уважаемые читатели. Вы держите в руках книгу, которая полностью посвящена разработке Web-приложений с использованием технологии *Ajax*. Возможно, вы уже сталкивались с этой технологией или даже являетесь гуру в области Ajax. Возможно, вы никогда даже не слышали о существовании технологии Ajax. Автор сделал все, чтобы предлагаемая вашему вниманию книга была интересной, а главное — полезной. Автор книги — практический программист, который уже не один год использует технологию Ajax в своей ежедневной деятельности. Поэтому все примеры программного кода книги обкатаны в реальных приложениях и совмещают простоту, надежность и практическую направленность.

Для тех, кто уже работал с технологией Ajax, вводная часть может быть на этом закончена, и можно переходить к непосредственно интересующей вас главе книги. Главы книги относительно независимы, хотя ключевой является *глава 2*, и ее автор рекомендует прочитать вне зависимости от уровня вашей подготовки. Такое пожелание продиктовано тем, что в последующих главах используются утилитные функции, которые разработаны в *главе 2*. Для тех же, кто впервые сталкивается с технологией Ajax, будет удобнее изучать главы в последовательном порядке, не пропуская, между прочим, и введения. Итак, мы — начинаем.

Ajax — это аббревиатура, которая расшифровывается как *асинхронный JavaScript и XML*. Второе (вернее, истинное) значение этого слова — имя древнегреческого героя, о чем уже не раз писали авторы литературы по Ajax. Но постараюсь и тут дать вам немного новой информации. На самом деле Аяксов было двое — *Большой Аякс* и *Малый Аякс*. Согласно мифу, они были друзьями. Поэтому лет, этак, 100 назад, когда гимназисты получали классическое образование, выражение "два Аякса" было у всех на слуху и означало примерно то же, что сейчас означают выражения неразлейвода, Фома и Ерема, Лель и Полель, два сапога пара. В этом (в том, что Аяксов было двое) можно усмотреть очень глубокий смысл, который и не снился авторам самой

технологии и термина Ajax. До появления технологии Ajax Web-приложение выполнялось преимущественно на стороне Web-сервера. Web-браузер в таком приложении играл роль пассивного монитора, который отображал полученный с Web-сервера HTML-документ. В противоположность классическому Web-приложению, Ajax-приложение выполняется как на Web-сервере, так и на Web-клиенте, т. е. в Web-браузере. Говоря образно, Web-сервер можно ассоциировать с Большим Аяксом, а Web-браузер — с Малым Аяксом.

Надо сказать, что наименование (аббревиатура) Ajax носит не столько смысловой, сколько маркетинговый характер. Когда я впервые увидел книгу, в названии которой присутствовало слово Ajax, то если меня и привлекло это название, то благодаря наличию букв **j** и **x**. Первая из них ассоциировалась с передовыми интернет-технологиями, а вторая — со всем хорошим, широким (и, в то же время, немного таинственным), что только может придумать человек. Наличие в названии сразу двух букв **a** (заметьте, как и Аяксы, одна большая и одна малая) только усиливало положительный эффект, показывая, что технология активная и находится на первом месте среди равных. Но только когда я пролистал (благо была возможность) полкниги, я, наконец, понял, что технология Ajax — то самое, что мне необходимо прямо сейчас и чего мне все это время не хватало при разработке Web-приложений.

На самом деле Ajax-технология — это не столько новая технология, сколько новый подход к разработке Web-приложений, использующий возможности, которые присутствуют в Web-браузере с конца 90-х годов прошлого века. Средства Web-браузера, которые используются в Ajax-технологии, принято называть инфраструктурой Ajax. Инфраструктуру Ajax составляют:

- документы HTML;
- объектная модель документа DOM (Document Object Model);
- каскадные таблицы стилей CSS (Cascade Stylesheets);
- язык программирования JavaScript;
- объект `XMLHttpRequest` Web-браузера.

Первые четыре составляющие инфраструктуры Ajax широко известны как *DHTML* (динамический HTML). Несмотря на широкие возможности технологии DHTML и публикацию близких к этой технологии спецификаций *DOM Level 1* и *CSS*, технология DHTML некоторое время оставалась в тени и не имела той популярности среди разработчиков Web-приложений, которая была бы достойна ее богатым возможностям и которую она начинает приобретать в настоящее время. Для этого есть, по крайней мере, две причины. Во-первых, достаточно долго Web-браузеры обеспечивали весьма слабую совместимость, позволяя кросс-браузерно отображать только самые простые

HTML-документы. Код более сложных HTML-документов и Web-приложений, которые работали с объектной моделью документа DOM при помощи JavaScript, разрабатывался или для конкретной версии Web-браузера, или использовал различные приемы, похожие на трюки, для достижения кросс-браузерной работы приложения.

В настоящее время, опубликованы спецификации *HTML 4.01*, *DOM Level 1*, *CSS2.1*, которые поддерживаются всеми основными производителями Web-браузеров. Со времени публикации этих спецификаций в конце 90-х годов прошлого столетия должно было пройти еще время, чтобы разработчики Web-браузеров обеспечили поддержку этих спецификаций, и большинство пользователей Web-приложений установили у себя на компьютерах новые версии Web-браузеров. На момент написания книги можно быть практически уверенным в том, что большинство пользователей имеют на своих компьютерах Web-браузеры, работающие с технологией DHTML и Ajax. Хотя для разработки кросс-браузерных приложений по-прежнему необходимо прикладывать дополнительные усилия, можно сказать, что прогресс в обеспечении кросс-браузерности достигнут колоссальный. И это создало предпосылки для широкого использования технологии DHTML разработчиками Web-приложений.

> ***Замечание***
>
> В настоящее время опубликованы и более новые спецификации HTML, DOM и CSS. Есть и проект новой спецификации JavaScript (ECMAScript), который, по мнению авторов, должен стать языком строго типизированным, поддерживать пакеты, классы и пространства имен.

Но была и вторая (после несовместимости Web-браузеров) причина, которая мешала широкому использованию технологии DHTML. Дело в том, что технология DHTML не предусматривала средств для обмена данными между Web-браузером и Web-сервером. Как уже стало ясно сейчас, с высоты достижений технологии Ajax, только совместное использование DHTML и фонового (без перезагрузки основного HTML-документа) обмена данными между Web-браузером и Web-сервером позволяют создавать по-настоящему динамические Web-приложения. Может показаться почти фантастическим, что вместе с появлением технологии DHTML в тот же период, был разработан объект `XMLHttpRequest` (пятый элемент в инфраструктуре Ajax), который также стал доступен для разработчиков. При помощи объекта `XMLHttpRequest` как раз и можно организовать в DHTML-приложении фоновый обмен данными между Web-браузером и Web-сервером. Но все составляющие инфраструктуры Ajax какое-то время не могли встретиться в рамках одного приложения и существовали сами по себе. В это время довольно часто для фонового обмена данными Web-браузера с Web-сервером использовались HTML-элементы

IFRAME или SCRIPT. Но по-настоящему удобным обмен данными между Web-браузером и Web-сервером стал с использованием асинхронных запросов при помощи объекта XMLHttpRequest.

Давайте разберемся, почему возникла технология Ajax и почему она возникла именно сейчас? Сравнительно длительный период (для нашего стремительного времени) развитие Web-приложений шло в направлении усложнения серверного программирования. Сначала Web-сервер просто доставлял HTML-документ в Web-браузер пользователя. Потом появилась возможность отсылать с Web-браузера на Web-сервер простые данные при помощи технологии CGI. Вскоре оказалось, что CGI-приложение может не только принимать и обрабатывать данные, отправленные Web-браузером, но и возвращать в Web-браузер динамически сгенерированный HTML-документ. Затем появилась возможность управлять сессиями, создавать серверные объекты. За этим великолепием серверных технологий в тени остался пользователь, который работал с Web-браузером. Независимо от того, какая технология работала на Web-сервере, пользователю доставлялась все та же HTML-страница. И работа пользователя с Web-браузером практически не изменялась при очередном переходе на все более и более прогрессивную серверную технологию. И это выглядело довольно неестественно, потому что Web-браузер уже в конце 90-х годов прошлого века позволял создавать Web-приложения, которые активно моги бы использовать средства программирования на стороне Web-браузера. Налицо было противоречие между все усложняющимися серверными технологиями и их результатом, который, с точки зрения пользователя, раскрывшего Web-браузер, застыл на уровне середины 90-х годов.

Программирование на стороне Web-браузера не только не развивалось, но даже считалось недопустимым при разработке серьезного Web-приложения. Отчасти, такое мнение имеет под собой почву. Выполнение ответственных бизнес-операций с использованием программирования на стороне клиента Web-браузера было бы достаточно рискованным решением. Это связано с тем, что Web-сервер не может контролировать среду выполнения клиентского приложения, как это могут делать некоторые серверы приложений, и этим может воспользоваться злоумышленник. Выполнение некоторых бизнес-операций, таких как вычисление стоимости заказа или оплата счета в интернет-магазине, допустимо только на сервере. Некоторые разработчики даже рекомендуют это делать не на Web-сервере, а использовать еще дополнительный уровень — сервер приложений. Но управлять поведением элементов пользовательского интерфейса при помощи полной перерисовки HTML-документа, полученного новым запросом с Web-сервера, как это делалось в классических Web-приложениях, — слишком тяжеловесное решение и с точки зрения Web-разработчика, и с точки зрения пользователя.

Как бы то ни было, инертность мышления Web-разработчиков некоторое время сдерживала развитие технологии DHTML и Ajax, но это не могло продолжаться вечно. Слово Ajax было произнесено, и приложения, построенные с использованием этой технологии, позволили сделать то, что еще совсем недавно никто не мог ожидать от Web-приложений.

Я уже говорил, что технология Ajax использует для реализации обмена данными с Web-сервером объект `XMLHttpRequest`. Что же представляет собой этот объект? Его можно назвать браузером в браузере или невидимым браузером. Этот объект может в фоновом режиме (без перезагрузки основного HTML-документа) отправить запрос на Web-сервер и получить ответ в виде текста или XML-документа. Полученный с Web-сервера текст или XML-документ не отображается в Web-браузере непосредственно, как при обычном запросе Web-браузера к Web-серверу. Ответ Web-сервера доступен только из языка программирования JavaScript (или другого скриптового языка, поддерживаемого Web-браузером), и это открывает новые возможности для Web-программирования. Если вам сейчас это не совсем понятно — не огорчайтесь. Ведь именно изучением того, как это сделать на практике, мы и будем с вами заниматься на протяжении всей книги.

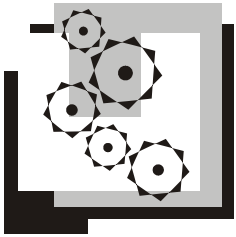
Возможности объекта `XMLHttpRequest` гораздо шире, чем это можно было бы предположить из названия объекта. Как и название Ajax, название объекта `XMLHttpRequest` может поначалу увести от понимания его прямого назначения. Разберем этот момент более подробно:

- XML — на самом деле, объект `XMLHttpRequest` может работать как с XML-документами, так и с текстом, например `text/html` или `text/plain`;
- Http — объект `XMLHttpRequest` может работать как с протоколом HTTP, так и с другими протоколами, которые поддерживает Web-браузер, например с протоколами HTTPS, file;
- Request — объект `XMLHttpRequest` может отправлять (методом `send()`) запрос (request) к Web-серверу. Кроме того, объект может получать ответ (response) Web-сервера (свойства `responseText` и `responseXML`).

Я подчеркиваю этот момент, потому что, исходя из названия объекта, у вас может сложиться впечатление, что технология Ajax предназначена исключительно для рафинированных теоретиков, имеющих дело с генерацией и разбором XML-документов. Нет, Ajax-приложение имеет дело с динамической загрузкой контента с Web-сервера в самых разнообразных, удобных и нужных вам вариантах. Это может быть и загрузка фрагментов HTML-документа, что с успехом используется в тех случаях, когда традиционно применялись фреймы. Это может быть загрузка программного кода JavaScript и, наконец, работа с XML-документами.

Теперь рассмотрим вопрос, чем Ajax-приложение отличается от классического Web-приложения. Ajax — это аббревиатура от *асинхронный JavaScript и XML*. Как это иногда бывает, Ajax — это еще и яркое запоминающееся название, которое служит продвижению этой замечательной технологии. Как бы то ни было, не стоит искать ответы на все вопросы только в названии. Если постараться сформулировать то, что выделяет Ajax-приложения в мире Интернета — так это будет новый удобный или, как принято говорить, *богатый* пользовательский интерфейс. В свою очередь, появление нового *богатого интерфейса* позволило расширить круг решаемых задач и расширить сферу применения Web-приложений. Ajax-приложение — это приложение, которое удобно для пользователя и обеспечивает функциональность, ставшую уже привычной для настольного (desktop) приложения. Несмотря на то, что технология Ajax связана с применением конкретных средств, которые мы называем инфраструктурой Ajax, основное в этой технологии — не используемые средства, а расширение границ применения Web-приложений, новое ощущение комфорта, который получает пользователь при работе с Web-приложением.

Я рад, что издательство "БХВ-Петербург" решило внести вклад в развитие технологии Ajax и поручило мне написать книгу, которую имею честь предложить вашему вниманию. В процессе работы над книгой я получал всяческую поддержку от коллектива издательства, за что хочу выразить свою благодарность. Той скорости решения вопросов, которые неизбежно возникали у меня в процессе подготовки книги, могли бы позавидовать даже железнодорожники. В связи с этим я особо хотел бы поблагодарить сотрудников издательства "БХВ-Петербург" Е. Е. Рыбакова, Г. Л. Добина и Александру, которые, несмотря на свою сверхзагруженность, всегда находили время, чтобы оперативно ответить на мои вопросы. Когда вы, уважаемые читатели, будете читать эту книгу, вспомните со словами благодарности о замечательном редакторе издательства "БХВ-Петербург" Анне Сергеевне Кузьминой. Просто текст отличается от хорошего текста тем что слова появляются вовремя и кстати, и еще чем-то неуловимым, о чем знают только мастера, такие как Анна Сергеевна. Спасибо Вам, уважаемая Анна Сергеевна! А также хочу поблагодарить своего друга и наставника Б. С. Свитского за предоставления широкого пространства для приложения Ajax-технологии и, особенно, за удачный перевод на русский язык моей любимой английской пословицы о том, что *не бывает обстоятельств столь благоприятных, которыми глупый человек не воспользовался бы себе во вред, и не бывает обстоятельств столь неблагоприятных, из которых умный человек не мог бы извлечь пользу. (Нет худа без добра)*



Глава 1

Создание компонента "Аккордеон"

В 80-х годах XX века было принято начинать обучение программированию с написания программы "Hello, world", которая печатала на черном экране монохромного дисплея два слова: "Hello World!" В 90-е годы программирование перешло на объектно-ориентированные рельсы, и программисты начали мыслить объектами. Поэтому первая программа начинающего программиста создавала объект `Dog` и вызывала единственный метод этого объекта: `Dog.bark()`. Осторожные программисты не забывали реализовать более полезный метод `Dog.foo()`.

Есть свои традиции и у Ajax-программистов. Первый компонент, который создают Ajax-программисты, называется *"Аккордеон"*. Такое музыкальное имя компонент "Аккордеон" получил благодаря своему внешнему сходству с этим замечательным инструментом. Когда компонент "Аккордеон" находится в свернутом состоянии, на экране отображаются только заголовки разделов. Если пользователь щелкнет по заголовку кнопкой мыши, разворачивается полный текст раздела — и мехи "Аккордеона" как бы растягиваются. Если пользователь щелкнет кнопкой мыши по другому заголовку — текущий раздел сворачивается и разворачивается новый раздел, соответствующий выбранному заголовку. "Аккордеон" играет!

Сложно сказать, почему именно этот компонент так полюбился Ajax-программистам. Нам же разработка компонента "Аккордеон" позволит сконцентрироваться на очень важных моментах в технике Ajax-программирования. В частности, на работе с *HTML-элементами* `DIV` и `SPAN`, *каскадными таблицами стилей CSS* и *объектной моделью документа DOM*. Кроме того, совсем неплохо на первом же уроке создать полнофункциональный компонент, который можно применить на практике.

Следующие два раздела посвящены использованию CSS в HTML-документах, после чего мы сможем приступить к разработке компонента "Аккордеон" во всеоружии.

Код готового компонента "Аккордеон" и некоторых примеров, рассматриваемых в данной главе, вы можете найти в папке \bhvajax\accordion на прилагаемом к книге компакт-диске.

1.1. HTML-элементы *DIV* и *SPAN* — основа построения современного HTML-документа

Основу построения современного *HTML-документа* составляют *элементы* `DIV` (от англ. *division* — раздел) и `SPAN` (от англ. *span* — фрагмент). Иногда вместо термина *"элемент"* не совсем точно употребляют термин *"тег"*, хотя тег и элемент — это совершенно разные понятия. *Открывающий тег* `<div>`, *закрывающий тег* `</div>` и часть документа, которая ограничена этими тегами, или *тело элемента* — называют элементом `DIV`. Тело элемента может содержать другие HTML-элементы и текст.

Элемент `DIV` отображается на экране в виде прямоугольного блока. Его свойство стиля `style.display` по умолчанию имеет значение `block`. Элемент `SPAN` отображается построчно, т. е. заполняет родительский блочный элемент построчно, как это делает печатная машинка на листе бумаги. Его свойство стиля `style.display` по умолчанию имеет значение `inline`. Для того чтобы ярче представить, о чем идет речь, создадим документ с использованием элементов `DIV` и `SPAN` (листинг 1.1).

Листинг 1.1. Использование HTML-элементов `DIV` и `SPAN`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <title>Test CSS</title>
  </head>
  <body>
    <h1>Заголовок первого уровня.</h1>
    <p>
      Абзацы P выделяются при форматировании Web-браузером.
    </p>
    <p>
      Часто применяют выделение текста <b>полужирным шрифтом</b>
      и <i>курсивом</i>.
    </p>
```

```
    <div>
      Элемент DIV может выступать в качестве заголовка.</div>
    <div>
      Подобно абзацу P, элемент DIV отображается как блочный элемент.
    </div>
    <div>
      А этот текст не будет выделяться <span>полужирным шрифтом</span>
      и <span>курсивом</span>.
    </div>
    <div>
      А для этого элемента стиль может быть задан
      <span id="mySpan">по идентификатору id=mySpan</span>.
    </div>
  </body>
</html>
```

Документ можно создать в любом текстовом редакторе, например Блокноте, и сохранить в файле с расширением html, например divandspan.html. Теперь сохраненный в файле HTML-документ можно открыть любым Web-браузером, дважды щелкнув по нему кнопкой мыши или используя меню Web-браузера **Файл | Открыть файл**.

Разберем приведенный пример. Открывает и закрывает HTML-документ открывающий тег `<html>` и закрывающий тег `</html>`. Любой HTML-документ всегда содержит ровно один корневой элемент `HTML`. Если в документе отсутствуют теги `<html>` и `</html>`, Web-браузер сформирует элемент `HTML` автоматически, как это предусмотрено спецификацией HTML. Элемент `HTML` содержит ровно два дочерних элемента: `HEAD` и `BODY`, которые так же будут обязательно добавлены Web-браузером, если они отсутствуют в тексте HTML-документа. Элемент `HEAD` содержит специальную информацию об HTML-документе в целом. Собственно текст документа содержится в теле элемента `BODY`, между открывающим тегом `<body>` и закрывающим тегом `</body>`.

> ***Замечание***
>
> Важным отличием элемента `HEAD` от элемента `BODY` является то, что все ресурсы, на которые ссылается документ в элементе `HEAD`, будут загружены с Web-сервера в Web-браузер до начала отображения элемента `BODY`. В качестве таких ресурсов чаще всего выступают скрипты JavaScript и файлы с определением стилей документа CSS. Крайне важно загружать файлы с определением стилей CSS именно в элементе `HEAD` HTML-документа. Скрипты JavaScript не обязательно помещать в элемент `HEAD`. Более того, некоторые скрипты можно размещать только в `HEAD` или только в `BODY`. В `BODY` помещают скрипты, которые манипулируют HTML-элементами `BODY` HTML-документа.

Как вы заметили, Web-браузер допускает некоторые вольности в написании HTML-документа. Это закреплено в спецификации HTML. Кроме того, Web-браузер может исправлять некоторые некритичные ошибки разработчика HTML-документа. В частности, в любом документе обязательно будет присутствовать корневой элемент `HTML`. Ссылку на элемент `HTML` можно получить из свойства `document.documentElement`, где `document` — это предопределенный объект Web-браузера, представляющий документ как объект Web-браузера, а `document.documentElement` — это свойство объекта `document`, которое представляет документ как корневой элемент `HTML` объектной модели документа (DOM).

Как мы уже выяснили, элемент `HTML` содержит ровно два дочерних элемента — `HEAD` и `BODY`. Некоторые HTML-элементы могут присутствовать только в теле элемента `HEAD`. Это элементы `STYLE`, `META` и `LINK`. Некоторые HTML-элементы могут присутствовать как в теле элемента `HEAD`, так и в теле элемента `BODY` — это элементы `SCRIPT` и `OBJECT`. Все прочие типы HTML-элементов и текст могут располагаться только в теле элемента `BODY`.

Разметка текста в листинге 1.1 дана в двух вариантах. Первый вариант — с использованием так называемых типографских тегов, отвечающих за форматирование текста: `H1` — заголовок первого уровня, `P` — абзац (paragraf), `I` — курсив (italic), `B` — полужирный шрифт (bold). Второй вариант разметки — с использованием всего двух тегов `DIV` и `SPAN`.

При помощи типографских тегов можно красиво оформить HTML-документ с учетом самых тонких требований дизайна. Для чего же, в таком случае, понадобилось вводить в спецификацию HTML еще два новых элемента `DIV` и `SPAN`? Потребность в двух новых элементах возникла с появлением технологии *каскадные таблицы стилей CSS*. Каскадные таблицы стилей применяются для форматирования документа и для придания ему динамичности. Вместе с *объектной моделью документа DOM* и *языком программирования JavaScript*, каскадные таблицы стилей CSS стали основой технологии *динамический HTML (DHTML)*, а впоследствии одной из составляющих *инфраструктуры Ajax*. Для того чтобы старая модель форматирования документа при помощи типографских тегов была совместима с новой моделью CSS, были введены два совершенно новых элемента: `DIV` и `SPAN`. Их внешний вид полностью определяется каскадными таблицами стилей. Без использования CSS элементы `DIV` и `SPAN` отображаются как обычный неформатированный текст.

Большинство типографских тегов (такие как `H1`, `P`, `B`, `I`) совместимы и широко употребляются с новой моделью форматирования CSS. Исключение составляют теги `BASEFONT`, `CENTER`, `FONT`, `S`, `STRIKE` и `U`, которые объявлены *нежелательными (deprecated)*. Кроме того, нежелательными объявлены и без того редко используемые теги `DIR`, `ISINDEX` и `MENU`.

1.2. CSS — каскадные таблицы стилей

Каскадные таблицы стилей CSS — ключевая технология при разработке пользовательских интерфейсов Ajax-приложений. Именно на технологии CSS построены динамические эффекты DHTML и Ajax. Меню появляются и скрываются, окна (не окна Web-браузера, а рисованные окна, реализованные при помощи HTML-элемента `DIV`) открываются, перетаскиваются и бросаются (Drag-and-Drop), распахиваются и сворачиваются, списки раскрываются и закрываются — все это основано на использовании каскадных таблиц стилей CSS.

В данном разделе будут рассмотрены основы CSS, достаточные для начала работы с технологией Ajax. Кроме того, предлагаемый раздел будет служить в качестве краткого справочника для работы с CSS на протяжении всей книги. Поэтому некоторый материал будет дан подробнее, чем это следовало бы сделать при первом знакомстве. К счастью, технология CSS компактна, поэтому даже если вы никогда с ней не работали — очень скоро будете хорошо в ней ориентироваться.

Суть технологии CSS заключается в том, что стиль отображения HTML-элемента в окне Web-браузера определяется значением свойств атрибута `style`, который может быть задан для каждого HTML-элемента или для множества элементов, принадлежащих к одному стилевому классу. Атрибуту `style` HTML-элемента можно задавать следующие свойства: цвет текста `style.color`, цвет или рисунок фона `style.background`, шрифт `style.font`, внешний вид рамки `style.border`, отступы текста от рамки внутри элемента `style.padding` и отступы от соседних элементов `style.margin`, размеры элемента (`style.width`, `style.height`), координаты (`style.top`, `style.left`), способ отсчета координат (`style.position`: `static`, `relative` или `absolute`), способ отображения текста, выходящего за рамки элемента (`style.ovreflow`: `visible`, `scroll`, `hidden` или `auto`), способ врезки элемента в текст документа (`style.display`: `inline`, `block`, `none` и др.), видимость элемента (`style.visibility`: `inherit`, `visible` или `hidden`) и некоторые другие.

Определять стиль для каждого элемента HTML-документа заданием всех свойств атрибута `style` было бы слишком утомительно. Поэтому в технологии CSS введены *стилевые правила*. Стилевое правило может определять свойства стиля сразу для всех HTML-элементов одного типа, имеющих одинаковое имя тега. Например, для всех элементов `H1` (заголовков первого уровня) или для всех элементов `P` (абзац). Но чаще стилевые правила связывают не с именем тега элемента, а с именем *стилевого класса*. Имя стилевого

класса задается в атрибуте HTML-элемента `class`. Например, для всех элементов, у которых атрибут `class` равен `mystyleclass`.

Можно задать и более узкое правило: одновременно по имени тега элемента и по имени стилевого класса. Например, только для элементов `DIV` с атрибутом `class` равным `mystyleclass`. Наконец, можно задать самое узкое правило, которое будет распространяться на один-единственный HTML-элемент с конкретным идентификатором, который задается в атрибуте `id` HTML-элемента. Например, для элемента с атрибутом `id` равным `myelement`.

Наиболее рационально определять стилевые правила для HTML-документа в отдельном файле с расширением css (cascade stylesheet). Формат определения стилевого класса в css-файле следующий:

```
имя_HTML_тега {
  имя_свойства1: значение_свойства1;
  имя_свойства2: значение_свойства2;
  ...
}
имя_HTML_тега.имя_стилевого_класса {
  имя_свойства1: значение_свойства1;
  имя_свойства2: значение_свойства2;
  ...
}
.имя_стилевого_класса {
  имя_свойства1: значение_свойства1;
  имя_свойства2: значение_свойства2;
  ...
}
#идентификатор_элемента {
  имя_свойства1: значение_свойства1;
  имя_свойства2: значение_свойства2;
  ...
}
```

В css-файле записывается *селектор* правила, в качестве которого выступает имя тега, имя стилевого класса или идентификатор элемента, заданный в атрибуте `id`. За селектором следует стилевое правило, которое выделяется фигурными скобками. Список свойств в определении стилевого правила разделяется точкой с запятой, а имя стилевого свойства и значение стилевого свойства — двоеточием.

Например, если вы решили, что все заголовки первого уровня `H1` должны отображаться буквами размером `14pt` цвета индиго, стилевой класс необходимо определить так:

```
H1 {
  color: indigo;
  font: 14pt
}
```

Чтобы Web-браузер прочитал определение стилей из css-файла, необходимо в элемент `HEAD` документа добавить элемент `LINK`, как это показано в примере:

```
<html>
<head>
<title>Test CSS</title>
<link rel="stylesheet" type="text/css" href="divandspan.css">
</head>
<body>
...
</body>
</html>
```

Для того чтобы связать стилевые классы, определенные в css-файле, с HTML-элементами, служат атрибуты HTML-элементов `class` и `id`. Все атрибуты HTML-элемента записываются в открывающем теге HTML-элемента в форме *имя_арибута="значение_атрибута"*. Значение атрибута рекомендуется всегда брать в двойные или одинарные кавычки. Если вы всегда будете заключать атрибуты HTML-элементов в двойные кавычки — это сделает ваш текст более наглядным.

Вернемся к примеру и применим полученные знания CSS (листинг 1.2).

Листинг 1.2. Применение каскадных таблиц стилей CSS

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <title>Test CSS</title>
    <link rel="stylesheet" type="text/css" href="divandspan.css">
  </head>
  <body>
    <h1>Заголовок первого уровня</h1>
```

```
    <p>
      Абзацы P выделяются при форматировании Web-браузером
    </p>
    <p>
      Часто применяют выделение текста <b>полужирным шрифтом</b>
      и <i>курсивом</i>.
    </p>
    <div class="header1">
      Элемент DIV в стиле заголовка первого уровня
    </div>
    <div class="paragraf1">
      Подобно абзацу P, элемент DIV отображается как блочный элемент.
    </div>
    <div>
      Другой способ выделения текста
      <span class="bold1">полужирным шрифтом</span>
      и <span class="italic1">курсивом</span>.
    </div>
    <div>
      А для этого элемента стиль задан
      <span id="mySpan">по идентификатору id=mySpan</span>
    </div>
  </body>
</html>
```

Приведенный в листинге 1.2 пример ссылается в HTML-элементе `LINK` на файл с определением стилей divandspan.css. Путь к файлу задан относительно текущей папки `src="divandspan.css"`, поэтому файл divandspan.css следует расположить в той же папке, что и файл с текстом примера divandspan.html. Заметим, что в промышленных приложениях разрабатываются css-файлы единые для всего приложения, и располагаются они в отдельной папке Web-сервера, которую часто называют \css.

Разработаем правила для отображения документа divandspan.html и поместим эти правила в файл divandspan.css. Пусть все элементы `H1` будут отображаться наклонным полужирным шрифтом размера `14pt` без засечек цвета индиго. Это правило для всех элементов `H1` будет определено в файле divandspan.css так:

```
H1 {
  color: indigo;
  font: italic bold 28pt/150% sans-serif
}
```

Для элементов стилевого класса `header1` определим точно такое же правило. Точка перед `.header1` в определении правила обязательна и обозначает, что `.header1` — это не имя HTML-элемента, а имя стилевого класса:

```
.header1 {
  color: indigo;
  font: italic bold 28pt/150% sans-serif
}
```

Два идентичных правила (как в нашем случае) можно записать в сокращенной форме через запятую:

```
H1, .header1 {
  color: indigo;
  font: italic bold 28pt/150% sans-serif
}
```

Запятая между `H1` и `.header1` обязательна. Без запятой такая запись имела бы совершенно другой смысл, и применялась бы к HTML-элементам с атрибутом `class` равным `header1` (т. е. стилевого класса `header1`), расположенным в теле элемента `H1`.

Для отображения элементов стилевого класса `paragraf1` определим стилевое правило: белые буквы на зеленом фоне в синей рамке:

```
.paragraf1 {
  color: white;
  background-color: green;
  border: solid 2px blue;
  font: 14pt/150% sans-serif
}
```

Стилевой класс `bold1` будет форматировать текст полужирным шрифтом без использования тегов `<b> </b>`:

```
.bold1 {
  font: bold 14pt/150% sans-serif
}
```

Стилевой класс `italic1` будет форматировать текст наклонным шрифтом без использования тегов `<i> </i>`:

```
.bold1 {
  font: bold 14pt/150% sans-serif
}
```

Наконец, чтобы закрепить работу с правилами для HTML-элементов, заданных по идентификатору элемента (атрибут `id`), определим правило для HTML-элемента с идентификатором `id` равным `mySpan`:

```
#mySpan {
  border-color: orange
}
```

Обратите внимание, в определении стиля для элемента с идентификатором `mySpan` используется не свойство `border`, а свойство `border-color`. Свойство `border` применяется как сокращенная форма записи сразу трех свойств: `border-style`, `border-color` и `border-width`. Если бы в файле divandspan.css вместо правила `border-color: orange` было определено правило `border: orange`, это бы означало, что неявно определяется правило `border-color: orange; border-style: none; border-width: 0px`, и элемент был бы отображен без рамки.

Свойство стиля `border` будет нами использоваться достаточно широко. При помощи этого свойства можно легко создавать динамические эффекты, например "подсвечивание" HTML-элемента, "нажатие кнопки". Разберем особенности применения свойства стиля `border` до конца. Это позволит нам в дальнейшем ссылаться на данный материал как на краткий справочник.

До сих пор мы применяли одинаковые стили ко всем четырем рамкам элемента: верхней, правой, нижней и левой. Стиль каждой из рамок можно определять раздельно при помощи свойств `border-top`, `border-right`, `border-bottom` и `border-left`:

```
#mySpan {
  border-top: solid orange 2px
}
```

Или, наконец, в самой полной форме:

```
#mySpan {
  border-top-color: orange
}
```

Допускается вариант сокращения, если вы хотите задать разные атрибуты для всех четырех рамок в одном правиле:

```
#mySpan {
  border-color: orange blue red yellow
}
```

Такая запись означает, что верхняя граница будет цвета `orange`, правая `blue`, нижняя `red` и левая `yellow`. Порядок перечисления границ ведется с верхней

границы по движению часовой стрелки. Это общее правило для всех аналогичных атрибутов CSS.

Значения свойства `border-style` определяет внешний вид линии, которой будет прорисована рамка. По умолчанию `border-style` линии равен `none`, т. е. никакой линии. Это означает, что для отображаемых линий свойство `border-style` всегда должно быть задано явно. Возможные значения свойства `border-style` следующие:

- `None` — значение по умолчанию, без рамки, принудительно устанавливает значение `border-width`, равное нулю;
- `Hidden` — скрытая, аналогично `none`, но имеет больший приоритет при наследовании стилей, что важно в некоторых сложных случаях наследования;
- `Solid` — сплошная линия (просто линия);
- `Double` — сплошная двойная линия;
- `Dotted` — точечная линия;
- `Dashed` — штриховая линия;
- `Groove` — вдавленная линия;
- `Ridge` — выпуклая линия;
- `Inset` — ступенька вверх;
- `Outset` — ступенька вниз.

Стили рамки `inset` и `outset` часто используют для рисования кнопок, которые можно нажимать и отпускать. Стиль `inset` (ступенька вверх) — обычное состояние кнопки. Стиль `outset` (ступенька вниз) — кнопка в нажатом состоянии.

Свойства стиля можно задавать непосредственно в атрибуте `style`. Я долгое время путал атрибуты `style` и `class`, потому что, на мой вкус, эти имена не совсем точно отражают их функциональность. Особенно это касается имени `class`, которое является зарезервированным словом в JavaScript и всегда рождает массу ненужных ассоциаций в головах объектно-ориентированных программистов. Подчеркнем разницу атрибутов `class` и `style`. Атрибут `class` содержит имя стилевого класса, который определен в css-файле и загружен в Web-браузер посредством элемента `LINK`. Атрибут `style` содержит строку с непосредственным определением параметров своего стилевого оформления. Например:

```
<span style="font: italic bold 14pt/150% sans-serif">
  Определение стиля в атрибуте style
</span>
```

Синтаксис определения стиля в css-файле и в атрибуте `style` одинаковый. Но в атрибуте `style` не следует брать определение стилевых правил в фи-

гурные скобки. Напомним, что подобно всем атрибутам, атрибут `style` следует взять в двойные или одинарные кавычки (апострофы).

Есть еще один способ определения стилевых классов, который мы не будем использовать в своих примерах. Определять стилевые классы можно в элементе `STYLE` непосредственно в `HEAD` HTML-документа. При этом теряется одно из главных преимуществ технологии CSS — отделение стилей оформления документа от текста документа (поэтому я даже не даю примера использования этой возможности).

Мы завершили обзор технологии CSS каскадных таблиц стилей. Теперь ответим на вопрос: почему таблицы стилей называют каскадными таблицами стилей? Дело в том, что на каждый элемент действует одновременно несколько стилевых правил, которые могут быть:

- унаследованы от родительского элемента, заданы в файле CSS или заданы непосредственно в атрибуте `style` элемента;
- взяты из профиля пользователя, настроек Web-браузера и операционной системы.

Из всех этих правил будет выбрано единственное правило с наибольшим приоритетом, которое и будет использовано для отображения элемента в окне Web-браузера. Такая логика определения стиля элемента называется наследованием и каскадированием. Правила наследования и каскадирования отвечают на вопрос, какое значение будет присвоено свойству стиля элемента при отображении HTML-документа, если на него действует сразу несколько стилевых правил.

Подведем итоги. Применение каскадных таблиц стилей позволяет вынести описание стилей оформления HTML-документа в отдельный файл с расширением css, ссылка на который должна содержаться в теге `LINK`:

```
<link rel="stylesheet" type="text/css" href="имя_файла.css">
```

Для привязки HTML-элемента к стилевому классу, определенному в css-файле, используется атрибут HTML-элемента `class`. Кроме того, стилевой класс можно определить для HTML-элементов по имени тега элемента или для конкретного элемента по его идентификатору, заданному в атрибуте `id`.

1.3. Разработка каскадных таблиц стилей для компонента "Аккордеон"

Сейчас мы приступаем к разработке компонента "Аккордеон". Преимущества использования технологии CSS заключаются в том, что разработку стилевого оформления компонента можно проводить независимо от разработки кода

компонента. Не имеет смысла задумываться, с чего начинать: с разработки кода или с разработки стилевого оформления. Разработку кода компонента и стилевого оформления можно вести параллельно, предварительно согласовав имена стилевых классов. Но формат книги требует последовательного изложения.

Компонент "Аккордеон" состоит из двух типов подкомпонентов: *заголовков разделов* и *панелей разделов*. Заголовки разделов всегда отображены в окне Web-браузера, если только отображен компонент "Аккордеон". Из панелей разделов всегда отображена только одна панель. Назовем такую видимую панель *активной* или *текущей панелью*. Все *неактивные панели* скрыты и представлены в окне Web-браузера только своими заголовками (рис. 1.1).

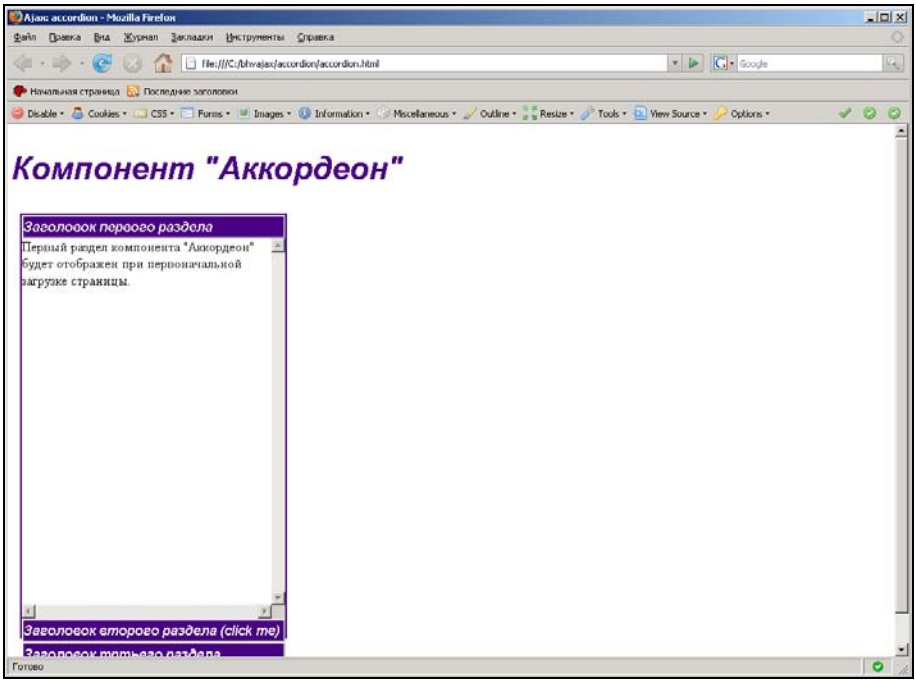


Рис. 1.1. Внешний вид компонента "Аккордеон"

Предварительно можно предположить, что для стилевого оформления компонента должны быть определены следующие стилевые классы:

- стиль компонента `class="accordionComponent"`;
- стиль активного заголовка `class="accordionCurrentTitle"`;
- стиль неактивного заголовка `class="accordionOtherTitle"`;

- стиль активной панели `class="accordionCurrentPane"`;
- стиль неактивной панели `class="accordionOtherPane"`.

Определения стилей поместим в файл accordion.css.

Я предложил начать разработку компонента "Аккордеон" с разработки стилей документа, чтобы вы на практике могли убедиться в удобстве технологии CSS. Если стилевое оформление документа отделено от самого документа, вы можете поручить разработку стилей специалистам в области художественного дизайна, можете вести работу параллельно, можете разработать несколько стилей для одного компонента или можете использовать один стиль для разных компонентов, чтобы все компоненты выглядели как единое целое. Согласитесь, заманчивая перспектива!

Для стилевого класса компонента "Аккордеон" зададим высоту `500px` и ширину `30%` в процентах от размеров родительского HTML-элемента (в нашем случае родительским элементом компонента "Аккордеон" будет HTML-элемент `BODY`). Обязательно присвоим значение свойству стиля `overflow: visible`, т. е. при переполнении размер элемента будет увеличен. Это существенно упрощает поставленную задачу, т. к. избавляет нас от необходимости вычислять высоту и ширину компонента в зависимости от многих заранее непредвиденных обстоятельств: типа Web-браузера, разрешения экрана, размеров родительского элемента, объема текста разделов компонента "Аккордеон", количества разделов. Более точные методы нам еще предстоит освоить, но сейчас мы создаем наш первый компонент. В файле accordion.css определение стилевого класса компонента "Аккордеон" будет следующим:

```
.accordionComponent{
  width: 30%;
  margin: 10px;
  border: solid indigo;
  border-width: 2px 2px 2px 2px;
  overflow: visible;
  height: 500px
}
```

Стиль заголовков разделов может быть произвольным. Но обязательно следует предусмотреть рамку (о рамках мы уже знаем все), иначе в свернутом состоянии заголовки "Аккордеона" будут сливаться. Заголовки активных и неактивных разделов в предлагаемом варианте будут иметь одинаковое стилевое оформление. Но, возможно, у вас появится желание сделать их различными. Можете прямо сейчас поэкспериментировать с цветами фона и стилевым оформлением рамок. Например, может быть, вы захотите задать для

активного заголовка рамку `inset`, а для неактивного заголовка рамку `outset`. Мне же кажутся более подходящими для данного случая такие стилевые правила:

```
.accordionCurrentTitle{
  background: indigo;
  color: white;
  font: italic bold 12pt/150% sans-serif;
  border: outset 2px
}
.accordionOtherTitle{
  background: indigo;
  color: white;
  font: italic bold 12pt/150% sans-serif;
  border: outset 2px
}
```

Для стилевого класса активной панели следует учесть несколько важных моментов. Значение свойства `display: block` означает, что панель будет отображена в окне Web-браузера. Значение свойства `overflow: scroll` означает, что при отображении текста, который не помещается на панели — будут сформированы полосы прокрутки. И, наконец, свойство `height: 90%`, заданное в процентном отношении от высоты родительского компонента "Аккордеон", позволяет логично отобразить разделы в случае, если текст раздела не заполняет все свободное пространство панели. Для таких разделов высота панели будет принудительно увеличена до `90%` от высоты компонента "Аккордеон". Это позволяет более логично отобразить панель, которая содержит короткий текст. Без указания `height: 90%`, панели компонента "Аккордеон" сжимались бы по размеру текста и имели бы различную высоту в зависимости от размера текста раздела. Сейчас мы используем примитивные, но надежные методы подгонки размеров компонентов, ведь это наш первый компонент. Не огорчайтесь, уже в этой главе мы применим более точные методы подгонки при помощи анализа свойств элементов `clientHeight` и `offsetHeight` средствами JavaScript.

Стилевой класс для активной панели в файле accordion.css определен так:

```
.accordionCurrentPane{
  border: none;
  overflow: scroll;
  display: block;
  height: 90%
}
```

Единственным свойством стилевого класса неактивной панели является `display: none`, что означает: панель невидима в окне Web-браузера и не занимает места в этом окне.

```
.accordionOtherPane{
    display: none
}
```

> ***Замечание***
>
> Свойство стиля `display` немного похоже на свойство стиля `visibility`. Свойство стиля `visibility` может принимать значение `inherit` (унаследовано от родительского элемента), `visible` и `hidden`. Если свойству стиля `visibility` присвоить значение `hidden`, элемент становится невидимым, но продолжает занимать место на экране. Нам необходимо сделать элемент неактивной панели невидимым и освободить занимаемое им место для отображения активной панели. Поэтому мы используем свойство стиля `display` со значением `none`.

Все определения стилей сохраним в файле accordion.css, который можно найти на компакт-диске. Продолжим разработку компонента.

1.4. Разработка HTML-документа для компонента "Аккордеон"

Компонент "Аккордеон" будет представлен в документе в виде HTML-элемента `DIV`. Заголовки разделов и панели разделов будут также представлены элементами `DIV`, которые содержатся в главном элементе `DIV` компонента "Аккордеон" и расположены в следующем порядке:

```
<div title="Компонент 'Аккордеон'">
<div>Заголовок раздела №1</div>
<div>Панель раздела №1</div>
<div>Заголовок раздела №2</div>
<div>Панель раздела №2</div>
...
</div>
```

Для работы с компонентом создадим файл accordion.html (см. на компакт-диске). Прошу вас обратить особое внимание на заголовок документа:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                "http://www.w3.org/TR/html4/loose.dtd">
```

Существует несколько схем `DTD` для HTML-документов. Мы будем использовать нестрогий тип `Transitional`. В практической работе лучше использо-

вать более строгую схему `Strict`. Иногда разработчики совсем не используют объявление типа документа в HTML-документах. Это неправильно. Дело в том, что поведение документов с явно объявленным типом является более кросс-браузерным и может существенно отличаться от поведения документа без явно объявленного типа. Если вы пишите документы сложнее, чем простой текст, явное объявление типа документа вам просто необходимо. Такой документ требует большей аккуратности разработчика. Если вы разработали документ без явного объявления типа, а затем решили добавить описание типа в готовый документ — будьте готовы к тому, что всю работу придется проделать заново. Поэтому явное объявление типа документа должно присутствовать с первых этапов разработки документа.

Теперь вы можете раскрыть Web-браузером файл accordion.html и... конечно заняться отладкой компонента, который включает на данном этапе два файла: accordion.css и accordion.html. Совершенно невероятно, чтобы все получилось сразу. Не стоит отчаиваться. Наконец начинается интересная работа. Проверьте наличие открывающих и закрывающих фигурных скобок в определениях стилей в файле accordion.css. Далее убедитесь в наличии двоеточий между именем и значением атрибута, а также точки с запятой после каждой пары *`имя_атрибута`*`:` *`значение_атрибута`*`;` в файле accordion.css. Теперь раскройте файл accordion.html. Здесь следует проверить парность открывающих и закрывающих тегов элементов, наличие наклонной черты в закрывающем теге перед именем тега элемента. Проверьте, все ли строковые атрибуты элементов правильно закрыты. Открывающая и закрывающая кавычки строкового атрибута должны быть одинаковыми — обе двойные или обе одинарные. Очень часто строковую константу просто забывают закрыть, и весь текст документа оказывается внутри одной большой строковой константы. Если вы добились правильного отображения документа — можно продолжить разработку компонента.

Наш компонент "Аккордеон" отображен Web-браузером: как и предполагалось, видны заголовки всех разделов и содержимое панели активного раздела. Все неактивные панели скрыты. Но нет возможности отобразить содержимое скрытых панелей.

Как же заставить компонент "Аккордеон" работать? Очевидное решение — по щелчку кнопкой мыши на заголовке раздела присвоить атрибуту `class` текущего раздела имя стилевого класса `accordionOtherPane`, а атрибуту `class` следующего выбранного раздела — имя стилевого класса `accordionCurrentPane`. Для этого мы должны привлечь возможности программирования на JavaScript.

1.5. "Аккордеон" начинает играть. Первое приближение к созданию компонента

Сейчас мы займемся программированием на JavaScript и сделаем первое приближение к созданию компонента "Аккордеон". Почему первое приближение? Дело в том, что первый вариант нашего компонента будет предполагать подробное ручное кодирование компонента. Мы должны будем явно в HTML-документе прописать стилевые классы и назначить обработчики события `onclick` для выбора очередной панели "Аккордеона". В окончательном варианте все это будет делать программа на JavaScript. Все, что нам необходимо будет сделать — вызвать разработанную нами функцию на языке JavaScript, которой в качестве параметра передать идентификатор `id` HTML-элемента `DIV` компонента "Аккордеон". Но не будем слишком забегать вперед.

Мы уже определили основное направление дальнейшей разработки компонента. Если вы помните, это звучит так: по щелчку кнопкой мыши на заголовке раздела присвоить атрибуту `class` текущего раздела имя стилевого класса неотображаемого неактивного раздела, т. е. `accordionOtherPane`, а атрибуту `class` следующего выбранного раздела — имя стилевого класса активного отображаемого раздела, т. е. `accordionCurrentPane`.

Большинству HTML-элементов можно назначать функции-обработчики событий, которые будут вызываться при наступлении определенных событий от клавиатуры или мыши, и некоторых событий высокого уровня — таких как получение или потеря фокуса ввода, изменение значения элемента, окончание загрузки документа. Все Web-браузеры обязаны поддерживать события `onclick` (щелчок кнопкой мыши или нажатие HTML-элемента `INPUT` типа `button`), `ondblclick`, `onmouseover`, `onmouseout`, `onmousedown`, `onmouseup`, `onmousemove`, `onselect` (выделение текста в HTML-элементе `INPUT`, `TEXTAREA`, в некоторых Web-браузерах — выделение любого текста), `onfocus` (элемент получил фокус), `onblur` (элемент потерял фокус), `onchange` (элемент изменил значение), `onsubmit` (нажата кнопка `SUBMIT`), `onreset` (нажата кнопка **Reset**), `onkeydown`, `onkeyup`, `onkeypress`, `onload` (документ полностью загружен), `onunload` (документ выгружается, например, при переходе к другой странице или при закрытии окна Web-браузера). Можно рекомендовать использовать только перечисленные события, если вы хотите добиться кросс-браузерности разрабатываемых приложений, потому что эти события закреплены в спецификации HTML.

Некоторые браузеры поддерживают и более специальные события — например `ondrag`, `onselectstart`, `onpaste` и др. Вы можете использовать эти нестандартные события, если уверены, что все пользователи будут работать

с конкретной версией Web-браузера. Это, разумеется, может быть только в том случае, если вы пишете приложение для Интранета или разрабатываете сайт, на котором ценители конкретной версии Web-браузера будут обмениваться мнениями о достоинствах своего фаворита.

Скопируем файл accordion.html в файл accordion1.html и продолжим работу с новым файлом. Поскольку на прилагаемом к книге компакт-диске содержится история нашей работы над компонентом "Аккордеон" — новый этап работы над компонентом представлен в файле accordion1.html. И завершающий этап работы будет представлен в файлах accordion2.html и accordion.js. Вы можете следовать или не следовать этим соглашениям.

Назначить обработчик события `onclick` (щелчок кнопкой мыши) можно в открывающем теге HTML-элемента следующим образом:

```
<div class="accordionOtherTitle"
onclick="accordionTitleOnClick(this);">
Заголовок второго раздела (click me)
</div>
```

Замечу, что `onclick` — это такой же атрибут HTML-элемента, как `id` или `class`. Правила его записи аналогичные: *`имя_атрибута="значения_атрибута"`*. Значение атрибута `onclick` содержит в виде строки код на языке программирования JavaScript. При срабатывании события `onclick`, т. е. когда пользователь щелкнет кнопкой мыши по HTML-элементу, будет выполнен заданный в строке программный код JavaScript.

> ***ЗАМЕЧАНИЕ***
>
> Назначение обработчика события в атрибуте открывающего тега HTML-элемента приводит к созданию новой функции JavaScript. У Web-браузеров, передающих функции-обработчику события объект `Event` в первом параметре функции-обработчика, создается функция с формальным параметром `event`, который может использоваться в коде, задаваемом в атрибуте открывающего тега HTML-элемента. Для тех браузеров, в которых объект `Event` доступен как свойство `window.event`, имя `event` будет также доступно, потому что все свойства объекта `window` доступны как имена глобального контекста.

В нашем конкретном случае по щелчку кнопкой мыши на заголовке раздела будет вызвана некоторая JavaScript-функция, которую нам предстоит еще разработать, `accordionTitleOnClick()` с фактическим параметром `this`. Имя `this` является ключевым словом языка JavaScript и означает ссылку на текущий объект. Поскольку событие `onclick` вызывается для HTML-элемента `DIV`, в переменной `this` содержится ссылка именно на тот элемент `DIV`, по которому щелкнули кнопкой мыши.

Строчку `onclick="accordionTitleOnClick(this);"` необходимо добавить в каждый открывающий тег всех *заголовков разделов*. Совсем скоро мы усовершенствуем компонент "Аккордеон" и вся рутинная работа будет выполняться программой на JavaScript. Но сейчас эту работу придется проделать вручную, чтобы постепенно осваивать приемы разработки Ajax-приложений. Если вы уже добавили в открывающий тег `DIV` заголовка каждого раздела определение обработчика события `onclick`, не закрывайте файл accordion1.html и добавьте новый элемент `SCRIPT`, в который мы поместим определение функции на языке JavaScript `accordionTitleOnClick()`.

Любой элемент `SCRIPT` должен содержать открывающий тег `<script>` и закрывающий тег `</script>`:

```
<script type="text/javascript"> </script>
```

Это правило нужно строго соблюдать, даже если тело элемента `SCRIPT` пустое.

Текст скрипта может содержаться либо между открывающим и закрывающим тегами элемента `SCRIPT` прямо в тексте HTML-документа, либо в отдельном файле с расширением js. Имя файла и путь к файлу, содержащему код JavaScript, задают в атрибуте `src` HTML-элемента `SCRIPT`. Обратите внимание, что любой элемент `SCRIPT` обязательно должен содержать атрибут `type` в открывающем теге `SCRIPT`. Этого строго требует спецификация HTML, в которой атрибут `type` определен как обязательный (required). Если скрипт написан на языке JavaScript, атрибут задается следующим образом: `type="text/javascript"`.

Элементы `SCRIPT`, в которых задана ссылка на js-файл в атрибуте `src`, в большинстве случаев следует определять в теле элемента `HEAD` между тегами `<head>` и `</head>`. Это связано с особенностью работы Web-браузера. Все внешние файлы, ссылки на которые определены в теле элемента `HEAD`, будут загружены до начала отображения элемента `BODY` документа в окне Web-браузера. В некоторых случаях это принципиально важно.

Первый вариант функции `accordionTitleOnClick()` будет играть роль так называемой *заглушки* для тестирования проделанной до этого момента работы. Это означает, что функция не будет выполнять действия по изменению стилевого оформления компонента "Аккордеон", а будет тестировать щелчки кнопкой мыши по заголовкам разделов компонента "Аккордеон" и выдавать отладочные сообщения. То есть она не будет выполнять всю ту работу, которую мы на нее возложим в ближайшем будущем для обеспечения правильной работы компонента "Аккордеон". Это сделано для того, чтобы упростить разработку и отладку компонента.

В листинге 1.3 приведен текст функции-"заглушки" `accordionTitleOnClick()`. Целиком текст документа accordion1.html вы можете найти на компакт-диске.

Листинг 1.3. Функция `accordionTitleOnClick()`

```
<script type="text/javascript">
// Функция-"заглушка"
function accordionTitleOnClick(element) {
    alert(element.innerHTML);
    alert(element.parentNode.innerHTML);
}
</script>
```

Конечно, наша функция-заглушка еще не умеет играть на аккордеоне, но помогает разобраться с объектной моделью документа. Функция вызывается с фактическим параметром `this`, а именно `accordionTitleOnClick(this)`. Фактический параметр функции ссылается на текущий объект, в качестве которого выступает выбранный HTML-элемент `DIV`, по которому пользователь щелкнул кнопкой мыши. В теле функции этот элемент доступен как формальный параметр функции — `element`.

Каждый HTML-элемент имеет целый набор свойств и методов. В частности, свойство `element.innerHTML` содержит строку HTML-текста, расположенного между открывающим и закрывающим тегом элемента. Это свойство доступно как для чтения, так и для изменения.

Каждый элемент, кроме корневого элемента `HTML` или `document.documentElement`, содержит ссылку на HTML-элемент, который является его непосредственным родителем. В компоненте "Аккордеон" непосредственным родителем заголовков разделов и панелей разделов является HTML-элемент `DIV` компонента "Аккордеон":

```
<div class="accordionComponent" id="accordion1">
```

Ссылку на непосредственного родителя можно получить из свойства элемента `element.parentNode`. В некоторых версиях Web-браузеров ссылка на родительский элемент содержится в свойстве `element.parentElement`. Из соображений кросс-браузерности, иногда в коде JavaScript анализируют наличие свойства `element.parentNode` и в случае его отсутствия пытаются получить ссылку из свойства `element.parentElement`. Мы этого не делаем по двум причинам. Во-первых, все популярные современные Web-браузеры поддерживают, по крайней мере, *спецификацию DOM Level 1*, согласно которой элемент должен иметь свойство `parentNode`, и, во-вторых, более перспективный путь — использовать готовые библиотеки поддержки кроссбраузерности. Обзор некоторых популярных библиотек приведен в *приложении 1*.

В теле функции-заглушки вызывается функция JavaScript `alert()`. Напоминаю, что функцию-заглушку мы используем для отладочных целей и, заодно, для изучения объектной модели документа (DOM). Функция `alert()` выводит окно с текстом, заданным как фактический параметр функции и ожидает нажатия кнопки **OK** или стандартной кнопки оконного интерфейса, закрывающей окно. Функция `alert()` является методом предопределенного объекта Web-браузера `window`. Все свойства и методы объекта `window` доступны в глобальном контексте. Поэтому функция может быть вызвана либо как `window.alert()`, либо как `alert()`. Функция `alert()` иногда используется при отладке скриптов JavaScript. Использовать ее в действующих приложениях следует очень осторожно, т. к. она может нарушить событийную модель документа, принимая на себя фокус ввода и приостанавливая (suspend) выполнение текущего скрипта.

Подведем предварительные итоги. Мы создали функцию-заглушку для тестирования компонента "Аккордеон" и связали эту функцию с событием `onclick` каждого из заголовков разделов. Теперь наша цель — написать полноценную функцию `accordionTitleOnClick()`, которая приведет в действие разрабатываемый компонент "Аккордеон".

Определение функции в JavaScript может содержаться в произвольном месте скрипта, если только используется определение в стиле:

```
function accordionTitleOnClick(element) {...}
```

Часто мы будем использовать определение функции при помощи анонимной функции в стиле:

```
var accordionTitleOnClick = function(element) {...}
```

Эти определения имеют одно существенное отличие. В первом случае определение функции может располагаться в любом месте кода. Во втором случае определение анонимной функции должно обязательно предшествовать вызову этой функции.

Вспомним, что единственный параметр, с которым вызывается функция `accordionTitleOnClick()`, ссылается на объект связанного с событием `onclick` заголовка раздела компонента "Аккордеон". Поэтому из свойства объекта `DIV` заголовка раздела можно извлечь ссылку на родительский HTML-элемент `DIV`, который является корневым элементом компонента "Аккордеон":

```
var accordion = element.parentNode;
```

Имея в своем распоряжении ссылку на объект корневой `DIV` компонента "Аккордеон", мы можем получить массив (коллекцию) ссылок на элементы `DIV`, которые содержат заголовки разделов и панели содержимого разделов.

```
var childs = accordion.childNodes;
```

Массив (коллекция) `childNodes` кроме элементов `DIV` может содержать *текстовые узлы* (text nodes), состоящие из пробелов и переводов строк. (Я употребляю слово "может" потому, что некоторые Web-браузеры не создают узлов, содержащих только пробельные символы.) Поэтому необходимо отфильтровать коллекцию `childNodes`, ссылка на которую сохранена в переменной `childs`, и выделить элементы с именем тега `DIV`, которые, как мы условились, содержат заголовки и панели содержимого разделов. Это можно сделать с использованием свойства элемента `childs[i].tagName`, которое содержит имя тега элемента в верхнем регистре (большими латинскими буквами):

```
var div = [];
for  (var i = 0; i < childs.length; i++)
  if (childs[i].tagName == "DIV")
    div[div.length] = childs[i];
```

Рассмотрим подробнее приведенный фрагмент кода.

```
var div = [];
```

Переменной `div` присваивается ссылка на пустой массив, как это было бы при вызове

```
var div = new Array();
```

Далее, в цикле `for` перебираются все *дочерние узлы* (child nodes) элемента `DIV` компонента "Аккордеон". Из этих узлов выбираются только элементы типа `DIV` и сохраняются в массиве `div`. Свойство `div.length` содержит текущий размер массива `div`. Поскольку индексация элементов массива начинается с нуля, присваивая значение элементу с индексом, равным `length`, мы добавляем новый элемент в конец массива. При этом `length` массива увеличивается на единицу и становится равным текущему размеру массива. За этим следит интерпретатор JavaScript Web-браузера, т. к. массив `div` является объектом встроенного JavaScript типа `Array`.

Далее, мы находим в массиве два следующие друг за другом элемента `DIV`, которые станут активным заголовком и активной панелью после вызова функции. Это элемент `DIV` заголовка, который вызвал функцию-обработчик события `onclick` и следующий за ним элемент `DIV` панели содержимого раздела:

```
var nextTitle = element;
  for (var i = 0; i < div.length; i ++)
    if (div[i] == nextTitle)
      nextPane = div[i+1]
```

Затем мы находим по имени класса элементы `DIV` заголовка и панели, которые были активными до вызова функции `onclick`:

```
for (var i = 0; i < div.length; i ++)
  if (div[i].className == "accordionCurrentTitle")
    currentTitle = div[i];
  else if (div[i].className == "accordionCurrentPane")
    currentPane = div[i];
```

Все, что нам осталось сделать, — поменять стилевые классы у активных и неактивных заголовков, так чтобы на экране отобразилась новая активная панель:

```
currentTitle.className = "accordionOtherTitle";
currentPane.className = "accordionOtherPane";
nextTitle.className = "accordionCurrentTitle";
nextPane.className = "accordionCurrentPane";
```

Полный код функции приведен в файле accordion.js на компакт-диске.

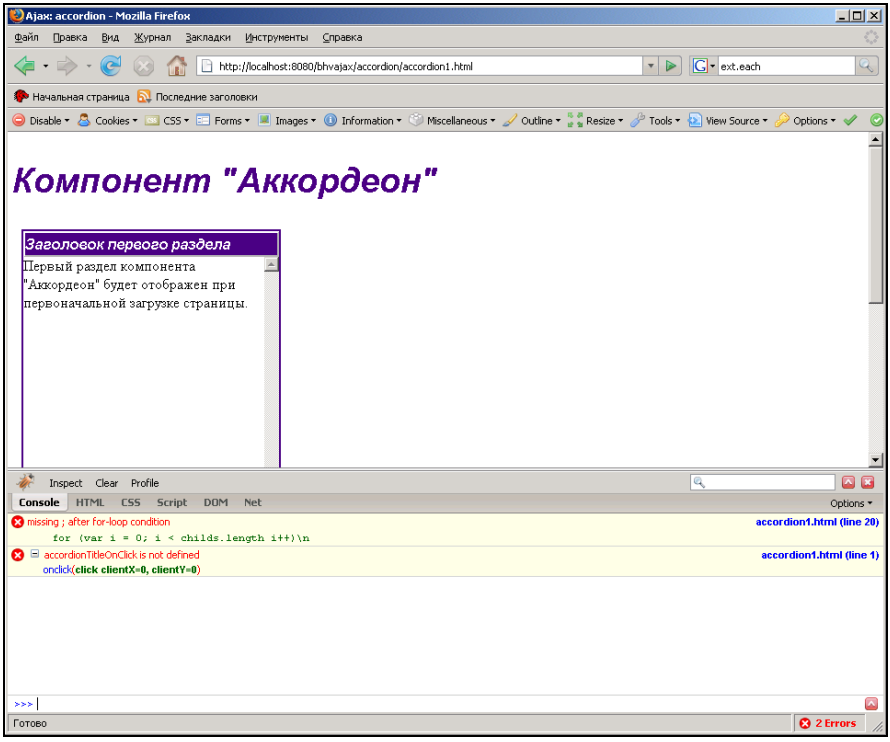


Рис. 1.2. Отладка программного кода при помощи плагина Firebug Web-браузера Firefox

У меня отладка данной функции заняла определенное время. Готовьтесь к тому, что и вам придется затратить некоторое время на отладку. Для отладки скриптов можно использовать самые разнообразные средства. Возможно, вы не догадывались, что некоторые средства уже давно у вас под рукой. Откройте Web-браузер Firefox и выберите пункт меню **Инструменты | Консоль ошибок**. Аналогично в Web-браузере Opera вы можете открыть пункт меню **Инструменты | Дополнительно | Консоль ошибок**. Перед вами раскроется окно, содержащее перечень сообщений об ошибках. Щелкнув кнопкой мыши по сообщению, вы раскроете текст кода JavaScript, HTML или CSS, который содержит ошибку. Очень многие разработчики используют плагин Firebug для Web-браузера Firefox. На рис. 1.2 вы можете увидеть, как моя очередная попытка отладить код в файле accordion1.html закончилась неудачей, и панель **Firebug**, открытая в нижней части Web-браузера, выдала мне очередную порцию грозных предупреждений.

Как видите, ошибку можно обнаружить, а значит, можно исправить. Добивайтесь успеха, и продвижение по океану Ajax с каждым новым компонентом будет для вас все более и более увлекательным.

1.6. Окончательное оформление компонента "Аккордеон"

Итак, наш компонент "Аккордеон" работает. Но для его формирования в HTML-документе необходимо проделать много ручной работы: прописать явно для каждого элемента `DIV` атрибут `class` и назначить обработчики события `onclick`. Все это можно сделать программно с помощью JavaScript. Этим мы сейчас и займемся.

Для начала скопируем файл accordion1.html, который мы только что отладили, в файл accordion2.html и немного поработаем в обратном направлении: удалим атрибуты `class` и `onclick` из открывающих тегов элементов `DIV`, уберем текст с определением функции из элемента `SCRIPT`. Весь JavaScript-код компонента будет вынесен в отдельный файл с расширением js. В этом случае JavaScript-код компонента может быть многократно использован в других приложениях. JavaScript-код компонента "Аккордеон" я поместил в файл accordion.js. Поэтому в тело HTML-элемента `HEAD` документа accordion2.html добавим элемент `SCRIPT` с атрибутом `src="accordion.js"`:

```
<script type="text/javascript" src="accordion.js"></script>
```

Привязка JavaScript-кода к HTML-элементам будет происходить с использованием идентификатора компонента — атрибута `id`. Добавим в открываю-

щий тег DIV компонента "Аккордеон" атрибут id="accordion1". Если вас не смущает, что в файле accordion2.html мы определяем элемент DIV с атрибутом id="accordion1", значит, вы все еще с нами. Окончательный вариант компонента "Аккордеон" можно найти на компакт-диске в файле accordion2.html.

Откройте файл accordion2.html любым Web-браузером, и вы увидите, что компонент "Аккордеон" исчез вместе с атрибутами class и onclick элементов DIV. Давайте подумаем, как можно восстановить компонент программным путем? Ответ витает в воздухе: все, что мы делали до этого момента вручную, нужно поручить программе на JavaScript. А именно: необходимо присвоить стилевые классы и обработчики событий onclick элементам DIV, содержащимся в компоненте "Аккордеон".

Напишем функцию accordionAddBehivior(), т. е. присоединим к HTML-элементу поведение компонента "Аккордеон". Это один из часто используемых приемов Ajax-программистов — взять обыкновенный HTML-элемент и присоединить к нему некоторую функциональность и стилевое оформление. Важно заметить, что функциональность компонента "Аккордеон" можно присоединить не к произвольному элементу, а к элементу, имеющему определенную структуру. В качестве контейнера для всего компонента "Аккордеон" должен выступать элемент DIV, который содержит элементы DIV с заголовками разделов и панелями содержимого разделов в следующем порядке:

```
<div>
<div>Заголовок раздела 1</div>
<div>Панель содержимого раздела 1</div>
<div>Заголовок раздела 2</div>
<div>Панель содержимого раздела 2</div>
...
</div>
```

Зная эту структуру, мы можем присоединить поведение (присоединить стилевые классы и обработчики событий onclick) к HTML-элементу DIV и получить в результате работающий компонент "Аккордеон".

> ***Замечание***
>
> Присоединение поведения к существующей в HTML-документе разметке — не единственный способ создания компонентов и не всегда самый удачный. В большинстве случаев, такая разметка может оказаться слишком сложной для ручного кодирования. В следующих главах мы будем идти немного другим путем. Компонент будет создаваться на основании пустого HTML-элемента с заданным идентификатором, а все необходимые данные будут передаваться функции создания компонента в виде JavaScript-объекта, содержащего модель данных создаваемого компонента. Но для компонента "Аккордеон" модель данных не так уж и сложна. Поэтому определение модели данных прямо в HTML-разметке документа вполне оправдано.

В первых строчках функции `accordionAddBehivior()` идет поиск по идентификатору, т. е. атрибуту `id`, элемента `DIV`, в котором будет располагаться компонент "Аккордеон". Делаем мы это при помощи метода объекта `document.getElementById()`. Объект `document`, как и объект `window`, всегда доступен для программы на JavaScript, которая выполняется под управлением Web-браузера. Методы и свойства объекта `document`: `createElement()`, `createTextNode()`, `getElementsByTagName()`, `getElementById()`, `documentElement` — очень активно используются Ajax–программистами. В частности, метод `document.getElementById()` ищет HTML-элемент с заданным атрибутом `id` (идентификатором), который передается этому методу в виде строкового параметра:

```
function accordionAddBehivior(accordion) {
  var accordion = document.getElementById(accordion);
  accordion.className = "accordionComponent";
...
```

Далее формируется массив `div`, содержащий HTML-элементы `DIV` с заголовками и содержимым панелей. Вспомним, что массив `accordion.childNodes` может содержать не только элементы `DIV`, но и текстовые узлы, состоящие из пробельных символов (включая перевод строки):

```
var childs = accordion.childNodes;
var div = [];
for  (var i = 0; i < childs.length; i++)
  if (childs[i].tagName == "DIV")
    div[div.length] = childs[i];
```

Далее мы работаем с массивом `div[]`, у которого в элементах с четным индексом (начиная с нуля) хранятся заголовки разделов, а в элементах с нечетным индексом — панели разделов:

```
for (var i = 0; i < div.length; i += 2){
  div[i].onclick = accordionTitleOnClick;
  if (i == 0){
    div[i].className = "accordionCurrentTitle";
    div[i+1].className = "accordionCurrentPane";
  }
  else{
    div[i].className = "accordionOtherTitle";
    div[i+1].className = "accordionOtherPane";
 }
}
```

Имя стилевого класса присваивается свойству `className`, которое соответствует атрибуту HTML-элемента `class`. Обычно JavaScript-свойства имеют то же самое имя, что и соответствующие атрибуты HTML-элемента. Замена имени HTML-атрибута `class` на имя JavaScript-свойства `className` — исключение, связанное с тем, что `class` является зарезервированным словом языка программирования JavaScript, хотя и не используется в текущей версии JavaScript. Свойству `onclick` заголовков раздела присваивается ссылка на функцию `accordionTitleOnClick`. Обратите внимание, что функция в операторе присваивания `div[i].onclick = accordionTitleOnClick` указывается без скобок после имени функции. Такая запись означает, что при вызове метода `div[i].onclick()` будет вызываться функция `accordionTitleOnClick()`, поскольку свойству `div[i].onclick` присвоена ссылка на функцию `accordionTitleOnClick`. И это еще не все! Функция `accordionTitleOnClick` автоматически становится методом объекта `div[i]`, который в теле функции будет доступен в качестве текущего объекта `this`. Таким образом, обработчик `onclick` и функция `accordionTitleOnClick` — это одна и та же функция, и объект `this` внутри функции `accordionTitleOnClick` ссылается на соответствующий элемент `DIV` заголовка раздела. В соответствии с этим различием, код новой функции по сравнению с уже рассмотренным нами вариантом функции `accordionTitleOnClick` будет изменен. Вместо использования формального параметра `element` для обращения к HTML-элементу `DIV` в первом варианте функции будет использован неявный параметр функции `this`.

```
function accordionTitleOnClick() {
  var accordion = this.parentNode;
  var childs = accordion.childNodes;
  var div = [];
  for  (var i = 0; i < childs.length; i++)
    if (childs[i].tagName == "DIV")
      div[div.length] = childs[i];
  var currentTitle = null;
  var currentPane = null;
  var nextTitle = this;
  var nextPane = null;
  for (var i = 0; i < div.length; i ++)
    if (div[i] == nextTitle)
      nextPane = div[i+1]
  for (var i = 0; i < div.length; i ++)
    if (div[i].className == "accordionCurrentTitle")
      currentTitle = div[i];
```

```
    else if (div[i].className == "accordionCurrentPane")
      currentPane = div[i];
currentTitle.className = "accordionOtherTitle";
nextTitle.className = "accordionCurrentTitle";
currentPane.className = "accordionOtherPane";
nextPane.className = "accordionCurrentPane";
nextPane.style.height = calculatePaneHeight(accordion)+ "px";
}
```

Если вы внимательно просмотрели код функции, наверное, обнаружили еще одно нововведение. Перед выводом на экран свойству стиля элемента активной панели раздела `style.height` присваивается некоторое вычисленное значение `calculatePaneHeight(accordion)`. Если вы помните, до сих пор подгонка размера компонента осуществлялась довольно примитивно. Настало время сделать это более точно. Функция `calculatePaneHeight(accordion)` работает со свойствами элемента `clientHeight` и `offsetHeight`. Подчеркиваю, что это свойства элемента, а не свойства атрибута `style` элемента. Свойства элемента `clientHeight` и `offsetHeight` являются очень удобными для программирования, потому что они:

- всегда имеют корректное значение;
- их значение числовое (например, `12`, но не `"12px"`);
- их значение измеряется в пикселах (`px`);
- они поддерживаются всеми популярными Web-браузерами;
- они доступны только для чтения.

Кроме `clientHeight` и `offsetHeight` есть такие же удобные свойства для ширины (`clientWidth` и `offsetWidth`) и координат (`clientLeft`, `clientTop`, `offsetLeft` и `offsetTop`) HTML-элемента. Web-браузеры поддерживают и другие свойства для вычисления высоты, ширины и координат HTML-элемента. О них можно сказать следующее:

- на момент вызова они могут быть либо не определены, либо равны 0;
- их значения строковые и могут содержать единицы измерения: `%`, `px`, `pt`, `mm`, `in`, `ex`, `em`;
- в каждом типе Web-браузера есть особенные свойства, не поддерживаемые в других Web-браузерах.

Функция `calculatePaneHeight(accordion)` работает следующим образом: вычисляет высоту компонента "Аккордеон", доступную для размещения дочерних элементов:

```
var paneHeight = accordion.clientHeight;
```

А затем из полученного значения вычитает в цикле высоту, необходимую для размещения элементов, содержащих заголовки разделов компонента "Аккордеон":

```
for (var i = 0; i < div.length; i += 2)
  paneHeight -= div[i].offsetHeight;
return paneHeight;
```

Обратите внимание, что для компонента, в котором размещаются элементы, мы вычисляем доступную высоту для рисования элементов `clientHeight`, а для элементов, которые размещаются внутри `clientHeight`, вычисляем высоту, необходимую для размещения — `offsetHeight`. Если вся работа проведена правильно, активизируемая панель четко впишется в рамки компонента "Аккордеон".

Вся проделанная нами работа представлена в файле accordion.js.

В заключение обсудим один очень важный момент. Зададимся вопросом, в каком месте HTML-документа следует вызывать инициализирующую функцию `accordionAddBehivior()`? И вообще, где можно и где нельзя вызывать JavaScript-код, отвечающий за инициализацию HTML-документа. Ответ опирается на следующие соображения. Функции, подобные `accordionAddBehivior()`, обращаются к объектной модели документа (DOM), чтобы находить, модифицировать и создавать HTML-элементы. Для успешной работы с DOM, HTML-документ должен быть полностью загружен. Если вы попытаетесь, например, добавить элемент в DOM HTML-документа до окончания полной загрузки документа — результат не всегда предсказуем. Все зависит от типа Web-браузера и мелких деталей, например, был ли достигнут закрывающий тег элемента, к которому вы пытаетесь добавить дочерний элемент. Правы те, кто рекомендует код, модифицирующий DOM, размещать в обработчиках события `onload`, которое вызывается после полной загрузки документа. Есть, по крайней мере, два кандидата для этой цели: `window.onload` и `body.onload`. Как показали мои эксперименты с различными моделями Web-браузеров, в некоторых из них `window.onload` и `body.onload` — это фактически одно и то же событие, а в других типах — разные. В пользу использования события `body.onload` (а не `window.onload`) говорит то, что это событие закреплено в *спецификации HTML*. Поэтому в продолжение всей книги для вызова JavaScript-кода инициализации последовательно будет использоваться событие `body.onload`. Хотя в литературе и в профессиональных библиотеках вы гораздо чаще можете встретить использование `window.onload`. Выбор за вами. Я изложил свою точку зрения.

1.7. Размещение компонента "Аккордеон" на Web-сервере Apache

Я специально начал свою книгу с разработки компонента, который с одинаковым успехом может работать и с использованием Web-сервера, и без Web-сервера. Это сделано для того, чтобы начинающие Web-программисты могли попробовать свои силы в технологии Ajax сразу, даже если на их компьютере не установлен Web-сервер. Теперь для начинающих Web-программистов пришел черед познакомиться с работой Web-сервера. Если вы не новичок в Web-программировании, предлагаемый материал не добавит ничего нового к вашим знаниям, но желательно все же ознакомиться с тем, какие конфигурационные установки рекомендуется применить, поскольку в изложении последующего материала книги я буду опираться на определенную конфигурацию Web-сервера Apache без дополнительных пояснений.

Вы можете использовать для работы с примерами книги любой Web-сервер, который поддерживает язык программирования PHP 5. Более того, все примеры из книги легко адаптируются для использования с произвольным серверным языком. Например, один из наиболее развернутых компонентов книги я адаптировал для использования с серверным кодом на языке программирования Python за два часа. Наиболее сложная часть Ajax-технологии связана как раз с клиентским программным кодом JavaScript, HTML и CSS. Поэтому, если вы хорошо ориентируетесь в серверной технологии, отличной от PHP 5, могу посоветовать сразу адаптировать примеры из книги для вашей любимой серверной технологии.

Для тех же, кто хочет получить дополнительные преимущества от использования примеров на прилагаемом к книге компакт-диске, можно рекомендовать ориентироваться на использования серверной среды Apache 2.2, PHP 5 и MySQL 5. Пока из этой великолепной тройки нам понадобится только Web-сервер Apache. Его мы и установим на свой компьютер, чему будет посвящена оставшаяся часть главы.

ВНИМАНИЕ!

Рекомендации по установке и конфигурированию Web-сервера Apache, приведенные в книге, нельзя использовать для конфигурирования реально работающих в сети Web-серверов. Рекомендации в этой книге даны по установке и конфигурированию Web-сервера Apache для использования его в качестве среды обучения программированию. Устанавливая Web-сервер на компьютере и не имея специальных знаний по администрированию сети, компьютера, Web-сервера и защите информации, вы открываете информацию как вашего компьютера, так и других компьютеров, с которым связан ваш компьютер, для потенциальных атак из сети. Установка Web-сервера в качестве среды обучения программированию предполагает максимальный уровень открытости этого

Web-сервера для отладочных операций, которые должны быть исключены при работе реального Web-сервера в сети. На компьютере, на который будет устанавливаться Web-сервер для обучения программированию, и на компьютерах, с которыми он соединен, не должна находиться конфиденциальная информация, и эти компьютеры не должны использоваться ни для каких-либо других целей, кроме обучения программированию.

ЗАМЕЧАНИЕ

Я рекомендую для целей изучения технологии Ajax разместить и Web-сервер Apache, и сервер баз данных MySQL на вашем локальном компьютере. Это предоставит вам неограниченную свободу действий и позволит экспериментировать с конфигурационными параметрами. Если по мере приобретения опыта вы захотите использовать более сложную конфигурацию с использованием нескольких компьютеров в сети, те из вас, кто только начинает изучать Web-программирование, будут способны сделать это самостоятельно.

Процесс установки Web-сервера Apache максимально прост для пользователя и никогда не вызывает затруднений. Для установки вам необходимо получить дистрибутив Web-сервера Apache. Легче всего это сделать через Интернет. Откройте любой поисковый сайт (поисковую машину) в Web-браузере и введите ключевые слова для поиска: *Apache download*. Откроется перечень ссылок, среди которых на первом месте будут ссылки на интернет-ресурсы разработчиков:

- **http://apache.org**;
- **http://httpd.apache.org**;
- **http://httpd.apache.org/download.cgi**.

Последняя ссылка на момент написания книги указывала на интернет-ресурс, с которого вы можете скачать дистрибутив Web-сервера Apache. Можете начать сразу с этой ссылки, но учесть, что URL-адреса интернет-ресурсов могут меняться.

На момент написания книги последней доступной версией Web-сервера был Apache 2.2.8. Его дистрибутив находится в файле с именем apache_2.2.8-win32-x86-no_ssl.msi, который и необходимо скачать с сервера. Реально, скачивание происходит с одного из зеркальных серверов. Это делается для разгрузки основного сервера. Иногда зеркальный сервер, ссылка на который по умолчанию будет вам предложена, может быть недоступен. Тогда вам необходимо из списка зеркальных серверов **mirror** выбрать другой сервер и подтвердить свой выбор, нажав кнопку **Change** на Web-странице, с которой загружается дистрибутив.

После загрузки файла дистрибутива сервера Apache с расширением msi, вы должны инсталлировать Web-сервер на свой компьютер. Для этого необхо-

димо запустить файл с расширением msi на выполнение (на установку), например, щелкнув по нему кнопкой мыши, из контекстного меню или другим известным вам способом. В процессе установки вам придется внимательно ознакомиться и принять лицензионное соглашение, а также ответить на ряд вопросов. В любой момент инсталляцию можно прервать, нажав кнопку **Cancel**. Для продвижения по шагам инсталляции вперед вам необходимо будет нажимать кнопку **Next**. Когда программа инсталляции предложит ввести информацию о сервере (рис. 1.3), для работы сервера на локальном компьютере вы должны ввести имя, которое используется для локального доступа к компьютеру `localhost`, и любой несуществующий e-mail-адрес, например `email@localhost`. Задание таких имен позволит вам работать с Web-сервером, даже если компьютер не подключен к сети. После окончания инсталляции эти имена можно изменить в конфигурационном файле Web-сервера Apache. На этом же шаге инсталляции, вы должны определить способ запуска Web-сервера — автоматически в виде службы для всех пользователей с использованием порта 80 (рекомендовано Apache) или вручную только для вас с использованием порта 8080. Как вы можете убедиться на рис. 1.3, мной был выбран второй вариант. И вам для изучения я рекомендую выбрать второй вариант. Это не идет в разрез с рекомендациями Apache, т. к. их рекомендации касаются запуска сервера в рабочем режиме, а мы конфигурируем среду для изучения Ajax-технологии.

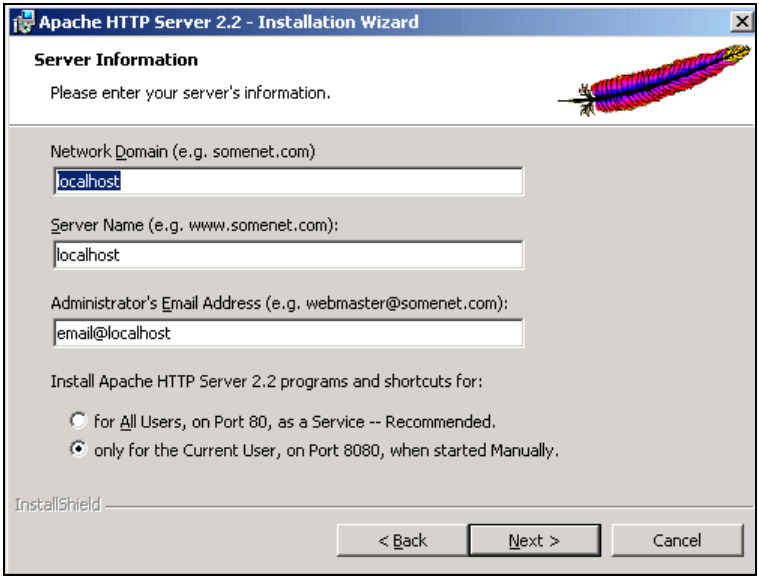


Рис. 1.3. Инсталляция Web-сервера Apache. Информация о сервере

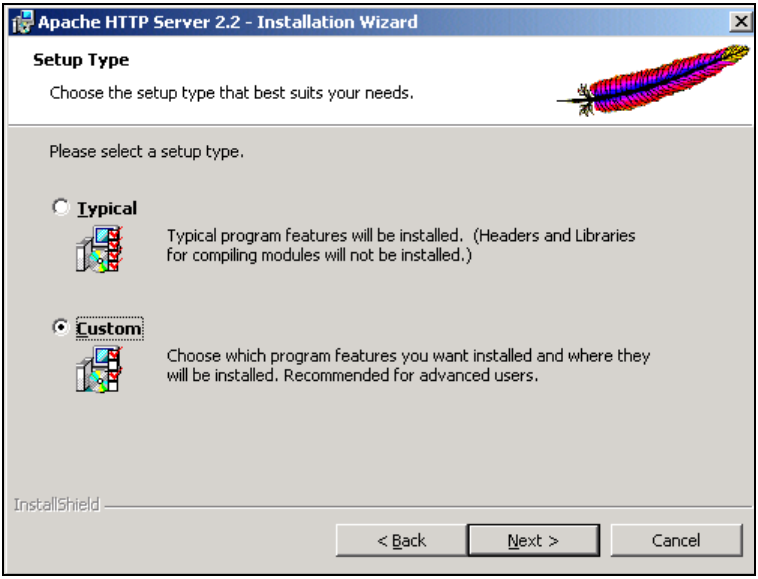


Рис. 1.4. Инсталляция Web-сервера Apache. Выбор типа инсталляции

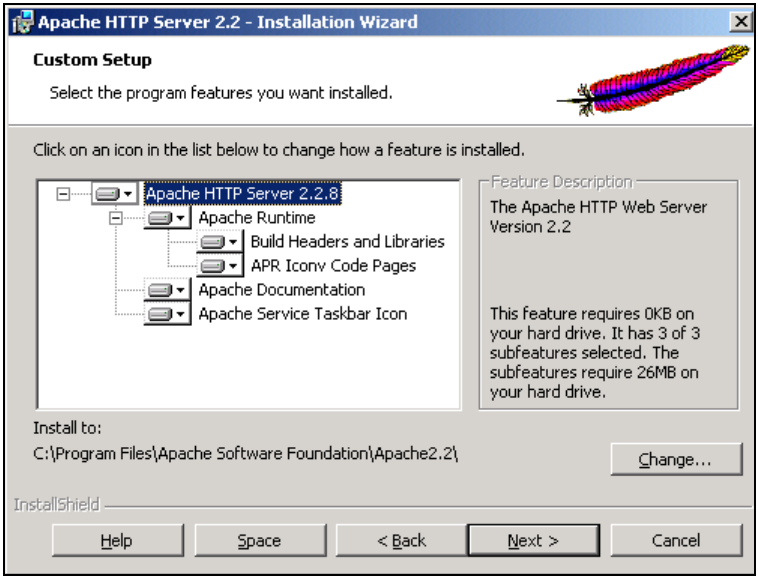


Рис. 1.5. Инсталляция Web-сервера Apache. Выбор параметров инсталляции

На следующем шаге выбирается тип инсталляции (рис. 1.4). Вам необходимо выбрать инсталляцию **Custom** (Выборочная). Это позволит изменить предлагаемую по умолчанию папку, в которую будет инсталлироваться сервер (рис. 1.5) на более простой путь C:\Apache. Для изменения пути инсталляции необходимо нажать кнопку **Change** (см. рис. 1.5) и в появившемся окне выбора папки ввести новый путь C:\Apache вместо пути, предлагаемого по умолчанию. Все мои рекомендации носят необязательный характер и могут быть проигнорированы. Но дальнейшее изложение будет опираться именно на такие конфигурационные параметры, и вам будет удобнее работать с книгой, если вы последуете моим рекомендациям.

После завершения инсталляции, на диске C: вашего компьютера должна появиться папка C:\Apache (рис. 1.6), а в меню **Пуск | Программы** — новый пункт **Apache HTTP Server 2.2**. Этот пункт меню должен содержать выпадающее подменю, в котором для запуска Web-сервера необходимо выбрать пункт **Control Apache Server | Start Apache in console**. После запуска Web-сервера на экране появится консоль, в которую будут выводиться сообщения Web-сервера. В случае если по каким-либо причинам Web-сервер не может стартовать, в консоль будет выведено предупреждающее сообщение о причинах сбоя, и через некоторое время, достаточное для того, чтобы прочитать предупреждающее сообщение, консоль закроется.

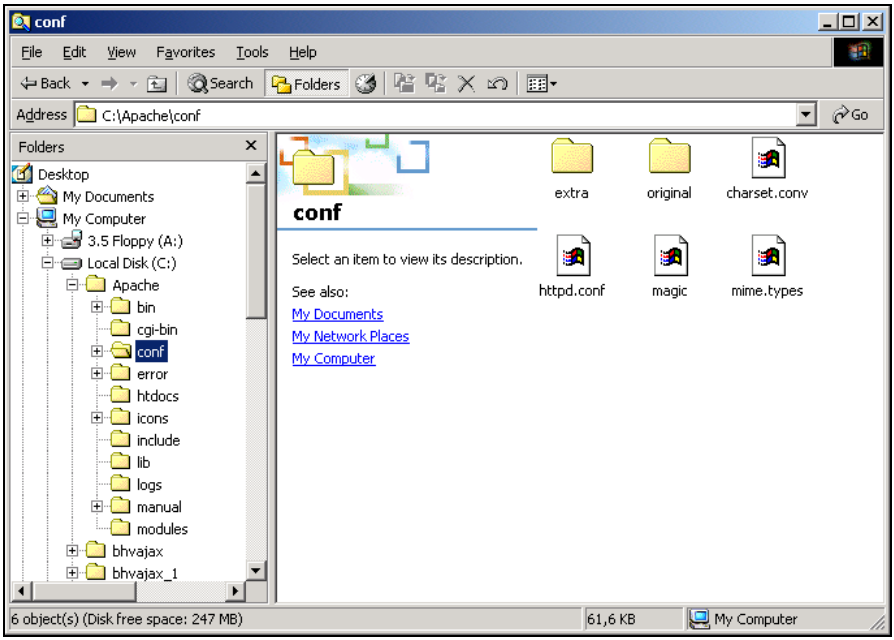


Рис. 1.6. Папка C:\Apache, в которую установлен Web-сервер Apache

Если Web-сервер правильно инсталлирован и правильно запущен, вы можете открыть Web-браузер и в строке адреса ввести **http://localhost**. В окне Web-браузера должно отобразиться содержимое файла index.html из корневого каталога DocumentRoot Web-сервера. В последних версиях Web-сервера Apache этот файл крайне лаконичен и выглядит так:

```
<html><body><h1>It works!</h1></body></html>
```

Web-браузер должен отобразить фразу "It works!". Более ранние версии Web-сервера Apache после инсталляции отображали в качестве основной страницы ссылку на документацию по Web-серверу. Почему от этого отказались — понять очень просто. Ссылка на документацию, которая случайно могла бы остаться в рабочей конфигурации Web-сервера, становилась потенциальной лазейкой для компьютерных хулиганов.

Теперь давайте вспомним, что примеры с прилагаемого к книге компакт-диска переписаны вами на диск C: в папку C:\bhvajax, а вы свои примеры располагаете в папке C:\testajax. Самый простой путь сделать эти папки доступными для Web-сервера и Web-браузера — переписать их в корневой каталог документов DocumentRoot Web-сервера. В нашем случае это каталог C:\Apache\htdocs. Но это не лучшее решение. Гораздо удобнее оставить наши папки на своем месте, но изменить конфигурацию Web-сервера так, чтобы к этим папкам можно было получить доступ. Этот вариант более выгоден тем, что дает возможность устанавливать необходимые конфигурационные установки для каждого каталога раздельно. Ведь для разных приложений могут использоваться различные, порой противоречащие друг другу конфигурационные установки. Перейдем к делу.

Нам нужно изменить конфигурационные файлы Web-сервера Apache так, чтобы папки C:\bhvajax и C:\testajax стали доступны. Иногда это называют публикацией папок или документов на Web-сервере. Конфигурационные файлы Web-сервера Apache обычно располагаются в папке conf, которая расположена в корневом каталоге сервера ServerRoot. В нашем случае конфигурационные файлы будут расположены в папке C:\Apache\conf (см. рис. 1.6). Основной конфигурационный файл Web-сервера Apache httpd.conf. С ним мы и будем работать.

> ***Замечание***
>
> В терминологии Web-сервера вместо термина "папка" (folder), как правило, употребляют слово "каталог" (directory). Исторически раньше для названия папок файловой системы повсеместно употреблялся термин "каталог". После создания дружественных пользовательских интерфейсов операционных систем, каталоги стали называть папками и отображать на экране пиктограммой с изображением конторской папки. А файлы стали называть документами. В большинстве случаев можно считать, что применительно к файловой системе

> папка и каталог — это одно и то же понятие. В контексте моего изложения я использую термин "папка", если речь идет о файловой системе, и "каталог", если речь идет о файловой системе применительно к Web-серверу.

Скопируйте файл httpd.conf в надежное место, чтобы можно было легко восстановить предыдущий вариант конфигурации. Затем откройте файл httpd.conf, который расположен в папке C:\Apache\conf в любом текстовом редакторе, например Блокноте. Перейдите в конец файла и вставьте следующий текст:

```
Alias /bhvajax/ "C:/bhvajax/"
<Directory "C:/bhvajax">
    Options Indexes
    AllowOverride None
    Allow from 127.0.0.1
    Dany from all
    AddDefaultCharset Windows-1251
</Directory>
```

Разберем эти конфигурационные параметры подробнее. В первой строчке назначается алиас (псевдоним) папке C:\bhvajax, которая с Web-сервера будет доступна по имени \bhvajax:

```
Alias /bhvajax/ "C:/bhvajax/"
```

В данном случае алиас (псевдоним) выбран похожим на физическое имя папки. Но имя может быть и произвольным, например:

```
Alias /serverajax/ "C:/folderajax/"
```

Теперь вам необходимо для папки C:\bhvajax задать конфигурационные параметры. Конфигурационные параметры задаются между открывающим и закрывающим тегом `Directory`:

```
<Directory "C:/bhvajax">
...
</Directory>
```

Немного напоминает HTML, не так ли?

Первая директива позволяет получать список файлов из папки:

```
Options Indexes
```

Эта возможность обычно закрывается в реальных Web-серверах из соображений безопасности, но вам позволит обращаться к файлам, введя имя каталога и выбрав нужный файл из списка по ссылке. Это очень удобно при разработке приложения.

Следующая директива запрещает переопределять конфигурационные параметры в специальных файлах .htaccess:

```
AllowOverride None
```

Разрешить такое переопределение имеет смысл, например, когда вы не имеете доступа к конфигурационному файлу сервера httpd.conf. Это не наш случай.

Следующие директивы разрешают доступ к вашей папке с локального компьютера, который имеет специальный IP-адрес 127.0.0.1 (localhost), и запрещают со всех остальных компьютеров:

```
Allow from 127.0.0.1
Dany from all
```

Если вы знакомы с конфигурированием Web-сервера, то легко измените эти директивы для того, чтобы ваше приложение было доступно не только с локального компьютера. Для начинающих я рекомендую именно такие директивы.

Теперь перейдем к последней, по порядку, но не по значимости директиве:

```
AddDefaultCharset Windows-1251
```

Этой директивой мы устанавливаем отправку Web-сервером в Web-браузер по умолчанию специального заголовка, который говорит, что содержимое файла содержит текст в кодировке Windows-1251, если такой заголовок не задан явно другим способом. Наличие различных кодировок символов кириллицы и связанные с этим неудобства очень часто вызывают вопросы у начинающих Web-программистов. Я в данной книге буду придерживаться системы правил, которые позволят правильно отображать в Web-браузере кириллические тексты. Одним из таких правил, которое определяется при конфигурировании Web-сервера, является директива `AddDefaultCharset`.

На протяжении всей книги будут последовательно использоваться файлы в кодировке Windows-1251. Хорошей альтернативой, снимающей массу проблем с кодировками, могло бы стать использование кодировки UTF-8. Я совершенно сознательно выбрал более сложный вариант с использованием кодировки Windows-1251, поскольку это позволит нам поговорить о проблемах с кодировками и найти приемлемые пути решения этих проблем. Я еще это делаю в интересах той части читателей, которые по каким-то причинам не могут использовать кодировку UTF-8. Например, хостинг не поддерживает такую кодировку, или необходимо обеспечить совместимость с существующим программным обеспечением, или используется Framework, который не поддерживает UTF-8. Если вы уверены, что будете всегда работать с UTF-8,

можете все ваши файлы сохранять как файлы UTF-8 и модифицировать соответствующие фрагменты кода, приведенного в книге, для использования с кодировкой UTF-8, например:

```
AddDefaultCharset UTF-8
```

После того как вы внесли изменения в фал httpd.conf и сохранили их, необходимо перезапустить Web-сервер Apache. (Если вы затрудняетесь перезапустить только Web-сервер, можете перезагрузить компьютер.) Если Web-сервер при запуске выдает в консоль сообщение об ошибке в файле конфигурации — вспомните, что вы сохранили в надежном месте предыдущую версию файла httpd.conf. Перепишите его в каталог conf Web-сервера и проделайте всю работу заново. Если вы запускаете сервер не в консоли, сообщения об ошибках можно прочитать в файле logs\error.log, который находится в папке logs Web-сервера.

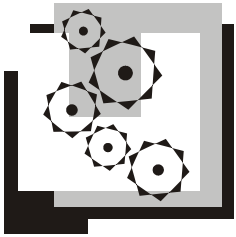
Наконец, вы опубликовали на Web-сервере каталог C:\bhvajax, в котором содержатся файлы с прилагаемого к книге компакт-диска. Теперь проделайте аналогичную работу и для своего тестового каталога C:\testajax:

```
Alias /testajax/ "C:/testajax/"
<Directory "C:/testajax">
    Options Indexes
    AllowOverride None
    Allow from 127.0.0.1
    Dany from all
    AddDefaultCharset Windows-1251
</Directory>
```

Теперь можно открыть компонент "Аккордеон" в Web-браузере с использованием Web-сервера. Для этого введите в строке адреса Web-браузера **http://localhost:8080/bhvajax/accordion/accordion2.html**. Аналогично, к своему тестовому компоненту вы теперь можете обратиться по адресу **http://localhost:8080/testajax/accordion/accordion2.html**.

* * *

Все вопросы, связанные с созданием компонента "Аккордеон", рассмотрены. Кроме того, вы получили работоспособный компонент "Аккордеон" для создания приложений и установили Web-сервер Apache на своем компьютере. А теперь мы готовы перейти к следующему этапу освоения технологии Ajax — изучению работы с объектом `XMLHttpRequest`.



Глава 2

Использование объекта *XMLHttpRequest* в Ajax-приложениях

В этой главе будут подробно рассмотрены все вопросы, связанные с использованием объекта `XMLHttpRequest`, который является ключевым элементом в инфраструктуре Ajax-приложения. При помощи объекта `XMLHttpRequest` можно:

- ❒ отправить http-запрос к Web-серверу;
- ❒ получить ответ с Web-сервера в виде текста или XML-документа.

Отправка запроса на Web-сервер и получение ответа с Web-сервера посредством объекта `XMLHttpRequest` выполняется в фоновом режиме, незаметно для пользователя или, как говорят, *асинхронно*. Под асинхронностью в данном случае понимается то, что пользователь продолжает работать с Web-браузером, пока запрос путешествует по сети и возвращается обратно в виде ответа Web-сервера. Это позволяет создавать Web-приложения, которые по комфортности работы для пользователя ничем не уступают настольным (desktop) приложениям с графическим пользовательским интерфейсом. Такой уровень комфортности для пользователей обеспечивает совместное использование объекта `XMLHttpRequest` и Динамического HTML (DHTML).

Объект `XMLHttpRequest` в данной главе будет разобран достаточно полно, так что у вас не останется больше вопросов к особенностям его работы. Кроме того, мы разработаем очень полезную функцию `sendRequest()`, которая существенно упростит работу с объектом `XMLHttpRequest`. Эта функция послужит основой для создания Ajax-компонентов во всех последующих главах книги.

Работа с объектом `XMLHttpRequest` напрямую, без вспомогательных функций или библиотек, не совсем удобна для разработчиков Web-приложений. Это связано с тем, что создатели объекта `XMLHttpRequest` и не предполагали, что их детище станет основным звеном инфраструктуры Ajax. В некоторых новых версиях Web-браузеров объект `XMLHttpRequest` частично усовершенствован. Но использовать его новые возможности, например доступ к объекту `XMLHttpRequest` в теле функции-обработчика через неявный параметр `this`,

еще не пришло время. При разработке Web-приложений мы должны стараться охватить как можно больший круг пользователей, у которых могут быть установлены самые различные версии Web-браузеров. Кроме того, даже усовершенствованный объект `XMLHttpRequest` все равно не обеспечивает удобный интерфейс для работы Web-программистов.

В чем же заключаются неудобства для разработчика прикладного программного обеспечения при работе с объектом `XMLHttpRequest`? Трудности начинаются с того, что в некоторых Web-браузерах используется ActiveX-объект `Msxml2.XMLHTTP` или `Microsoft.XMLHTTP`, в то время как другая часть Web-браузеров использует встроенный объект `window.XMLHttpRequest`. Эти два объекта различаются в деталях работы. Например, ActiveX-объект `Msxml2.XMLHTTP` или `Microsoft.XMLHTTP` вызывает функцию-обработчик ответа Web-сервера и для асинхронных, и для синхронных запросов, в то время как объект `window.XMLHttpRequest` вызывает функцию-обработчик ответа Web-сервера только для асинхронных запросов. Проблемой при работе с объектом `XMLHttpRequest` является невозможность без использования *замыкания* (closure) передать параметры функции-обработчику ответа Web-сервера. Подробно о замыканиях и функциях-обработчиках ответа Web-сервера мы поговорим далее в этой главе. Абсолютно все перечисленные проблемы найдут исчерпывающее решение, и мы общими усилиями создадим функцию `sendRequest()`, которая скроет от Web-разработчика все неудобства работы с объектом `XMLHttpRequest`.

Разрабатываемая в этой главе функция `sendRequest()` использует достаточно сложные для начинающих конструкции языка JavaScript. Если это ваше первое знакомство с JavaScript — можно не добиваться полного понимания всех тонкостей работы программного кода. Вам важно получить общее представление о принципах работы с объектом `XMLHttpRequest`, а код функции `sendRequest()` можно скопировать с прилагаемого к книге компакт-диска. Когда же вы освоите JavaScript в полном объеме, то сможете вернуться к более подробному изучению программного кода функции `sendRequest()`.

Текст примеров данной главы вы можете найти в папке \bhvajax\xmlhttprequest, а функцию `sendRequest()` — в файле \bhvajax\bhv\util.js на прилагаемом к книге компакт-диске.

2.1. Варианты использования объекта *XMLHttpRequest* при взаимодействии Web-браузера с Web-сервером

Процесс отправки запроса и получения ответа при помощи объекта `XMLHttpRequest` происходит без перезагрузки текущего HTML-документа,

загруженного в Web-браузер. Как же можно использовать возможности объекта `XMLHttpRequest`? Все разнообразие взаимодействия Web-браузера с Web-сервером в Ajax-приложениях при помощи объекта `XMLHttpRequest` сводится к трем основным вариантам:

- ответ Web-сервера содержит фрагмент HTML-документа;
- ответ Web-сервера содержит XML-документ;
- ответ Web-сервера содержит фрагмент программы JavaScript.

Гораздо реже используют запрос с Web-сервера файлов, содержащих определение каскадных таблиц стилей CSS.

Перечисленные варианты использования больше касаются технической стороны взаимодействия Web-браузера с Web-сервером. С точки зрения логики использования перечисленных технических возможностей, вариантов использования будет гораздо больше. Например, XML-документ может быть использован как источник данных, может быть подвергнут XSLT-преобразованию в HTML-документ или использоваться для работы с Web-сервисами. Фрагмент программы JavaScript может представлять собой код статического модуля JavaScript (js-файла), загружаемого по требованию в Web-браузер, или динамически генерируемый Web-сервером фрагмент программы JavaScript, подробности которого будут зависеть от параметров запроса. Но чаще всего в качестве фрагмента программы JavaScript загружается объект, записанный в JavaScript Object Notation (JSON). Давайте познакомимся со всеми перечисленными возможностями подробнее.

2.1.1. Использование объекта *XMLHttpRequest* для загрузки фрагмента HTML-документа

Объект `XMLHttpRequest` может запросить с Web-сервера фрагмент HTML-документа. Ответ Web-сервера будет содержаться в свойстве `responseText` объекта `XMLHttpRequest`, которое имеет тип `String`. Для обновления HTML-документа данными, полученными с Web-сервера, значение свойства `responseText` необходимо присвоить свойству `innerHTML` существующего или вновь создаваемого HTML-элемента. Web-браузер в соответствии с новым значением свойства `innerHTML` HTML-элемента обновляет объектную модель документа (DOM) и внешний вид документа. Это обычный путь использования объекта `XMLHttpRequest`, если ответ Web-сервера содержит фрагмент HTML-документа.

Рассмотрим простой пример. Пусть в условном приложении интернет-магазина Web-сервер в качестве ответа на запрос `XMLHttpRequest` формирует

фрагмент HTML-документа с разметкой переключателей для выбора цвета заказываемого товара с учетом текущего остатка на складе:

```
<input type="radio" name="color" value="13" checked>оливковый<br>
<input type="radio" name="color" value="19">хаки<br>
<input type="radio" name="color" value="21">индиго<br>
```

Полученный с Web-сервера фрагмент HTML-документа присваивается свойству `innerHTML` HTML-элемента `FIELDSET` с атрибутом `id="selectColor"`:

```
var xhr = new (window.XMLHttpRequest || ActiveXObject)("Msxml2.XMLHTTP");
xhr.open("GET", "ajax_response.php", false);
xhr.send(null);
var selectColorVar = document.getElementById("selectColor");
selectColorVar.innerHTML = xhr.responseText;
```

Свойство `innerHTML` представляет собой часть HTML-документа между открывающим и закрывающим тегом элемента. После выполнения приведенного кода, элемент `FIELDSET` будет содержать три элемента `<input type="radio">`, как если бы в тексте HTML-документа содержалась следующая разметка:

```
<fieldset id="selectColor">
  <input type="radio" name="color" value="13">оливковый<br>
  <input type="radio" name="color" value="19" checked>хаки<br>
  <input type="radio" name="color" value="21">индиго<br>
</fieldset>
```

Такой вариант взаимодействия Web-браузера с Web-сервером является переходным от классического Web-приложения к Ajax-приложению. Его использование позволяет с минимальными изменениями в существующем коде адаптировать классическое Web-приложение к технологии Ajax. Шагом вперед является то, что текущая страница не перезагружается и работа пользователя становится более комфортной, без вынужденных перерывов, связанных с ожиданием ответа Web-сервера и полной перерисовкой страницы в окне Web-браузера. Но роль Web-браузера по-прежнему остается статичной. Web-браузер только отображает полученный с Web-сервера готовый фрагмент HTML-документа.

Использовать загрузку фрагмента HTML-документа в качестве ответа Web-сервера можно рекомендовать в тех случаях, когда традиционно использовались HTML-элементы `IFRAME`. Однако для более серьезных приложений, интенсивно работающих с данными, такой подход приведет к усложнению разработки за счет появления жесткой зависимости серверного и клиентского кода. Загружаемый с Web-сервера фрагмент HTML-документа должен быть сформирован с таким расчетом, чтобы точно вписаться в текущий HTML-

документ. Следовательно, программный код серверного и клиентского приложений должен быть тщательно согласован, и разработчик серверного кода должен хорошо знать HTML. Этот недостаток первого из трех основных вариантов использования асинхронных запросов к Web-серверу при помощи объекта `XMLHttpRequest` в объектно-ориентированном анализе называют *жесткой связанностью*. Недостатки жесткой связанности особенно ярко проявляются при разработке проектов, в которых участвует несколько разработчиков разного направления, как это часто бывает при разработке Web-приложений.

ЗАМЕЧАНИЕ

Вас может заинтриговать фрагмент кода, создающего объект `XMLHttpRequest`:

```
var xhr = new (window.XMLHttpRequest || ActiveXObject)
("Msxml2.XMLHTTP");
```

Этот оператор следует читать так. Выражение в скобках вернет первое значение, отличное от `null`:

```
(window.XMLHttpRequest || ActiveXObject)
```

Это значение будет использовано для создания объекта оператором `new`. Фактически будет выполнен один из вариантов программного кода, в зависимости от модели Web-браузера:

```
var xhr = new window.XMLHttpRequest("Msxml2.XMLHTTP");
```

или

```
var xhr = new ActiveXObject("Msxml2.XMLHTTP");
```

В первом варианте параметр функции-конструктора будет проигнорирован. Второй вариант будет выполнен только при отсутствии у Web-браузера объекта `window.XMLHttpRequest`.

2.1.2. Использование объекта *XMLHttpRequest* для загрузки XML-документа

Объект `XMLHttpRequest` может запросить с Web-сервера XML-документ. Для того чтобы упростить работу с XML-документом на стороне Web-браузера, объект `XMLHttpRequest` имеет, кроме свойства `responseText`, еще и свойство `responseXML`. В отличие от свойства `responseText`, которое имеет тип `String`, свойство `responseXML` имеет тип `Document` (иногда его называют `DOMDocument`). Это свойство реализует спецификацию *DOM Level 1* (и выше). Подробно об этом мы поговорим в *главе 4*. А сейчас приведем фрагмент кода, для того чтобы вы постепенно готовились к восприятию материала *главы 4*:

```
var xhr = new (window.XMLHttpRequest || ActiveXObject)("Msxml2.XMLHTTP");
xhr.open("GET", "ajax_response.xml", false);
```

```
xhr.send(null);
var doc = xhr.responseXML;
var products = doc.getElementsByTagName("product")
var id = products[0].getElementsByTagName("id")[0].firstChild.data;
```

Как вы заметили, работа с объектом типа `Document` (`DOMDocument`) немного напоминает работу с предопределенным объектом `document` Web-браузера. И это не удивительно, потому что объектная модель HTML-документа (DOM), которую реализует предопределенный объект Web-браузера `document`, также определена в спецификации DOM Level 1. Именно в этой спецификации определены часто используемые свойства, коллекции и методы предопределенного объекта `document` Web-браузера: `documentElement`, `forms`, `images`, `cookie`, `getElementById()`, `getElementsByTagName()`, `write()`, `writeln()` и др.

На основании разбора полученного с Web-сервера XML-документа обновляются модель данных на стороне Web-браузера, объектная модель документа (DOM) и внешний вид документа. Этот вариант использования объекта `XMLHttpRequest` наиболее перспективный, т. к. XML является своеобразным общим знаменателем, язык которого понимают разработчики разных направлений. Формат XML-документов может быть согласован на самых ранних этапах разработки приложения, после чего каждый разработчик (группа разработчиков) продолжает работу над своей частью приложения, опираясь на формат согласованного XML-документа. Это существенно ускоряет разработку приложения и устраняет *жесткую связанность* между серверным и клиентским кодом.

На работе с XML построена технология преобразования XML-документов в HTML-документы, которая называется XSLT. В большинстве случаев такое преобразование выполняется на Web-сервере. Но вполне возможно, что у вас будет необходимость реализовать XSLT-преобразование средствами Web-браузера. Для использования технологии XSLT на стороне Web-браузера, вам пришлось бы проделать слишком много работы, если бы такая работа не была уже проделана разработчиками специализированных библиотек JavaScript. Такие библиотеки обеспечивают поддержку кроссбраузерной работы с XML-документами и собственно XSLT-преобразование XML-документа в HTML-документ.

Другой технологией, построенной на использовании XML-документов, является использование Web-сервисов. Возможно, вы уже сейчас планируете использовать Ajax-запросы для обращения к Web-сервисам, но тут вас может ожидать одна проблема. В Web-браузере действует модель защиты, называемая "песочницей". Такое образное название достаточно точно характеризует

суть происходящего. Если пользователь загрузил основной HTML-документ с вашего домена, то все его попытки обратиться при помощи объекта `XMLHttpRequest` к интернет-ресурсам за пределами вашего домена (за пределами "песочницы") вызовут ошибку запрета доступа. Это делает практически невозможным непосредственный доступ при помощи объекта `XMLHttpRequest` к Web-сервисам, расположенным за пределами вашего домена. Чтобы преодолеть такое ограничение, доступ к Web-сервисам на чужих доменах необходимо реализовывать, используя свой Web-сервер как ретранслятор запросов. Разумеется, для этого следует использовать готовые специализированные серверные и клиентские библиотеки, так что для прикладного Web-программиста вся работа по ретрансляции запросов и ответов к Web-сервисам останется за кадром.

Замечание

Следует учитывать, что модель защиты "песочница" не распространяется на загружаемые изображения (HTML-элементы `IMG`) и скрипты (HTML-элементы `SCRIPT`), если только не задать такое ограничение явно в опциях Web-браузера. Загрузка скриптов с чужих доменов является потенциально опасной операцией, о чем должен всегда помнить Web-разработчик.

2.1.3. Использование объекта *XMLHttpRequest* для загрузки фрагментов программы JavaScript

Объект `XMLHttpRequest` может запросить с Web-сервера фрагмент программы JavaScript. Полученный с сервера фрагмент программы JavaScript выполняется встроенной функцией JavaScript `eval()`. Эта функция выполняет фрагмент программы JavaScript, заданный в качестве параметра типа `String`, и возвращает значение последнего выполненного оператора, которое можно присвоить некоторой переменной:

```
var xhr = new (window.XMLHttpRequest || ActiveXObject)("Msxml2.XMLHTTP");
xhr.open("GET", "ajax_script.js", false);
xhr.send(null);
var returnValue = eval(xhr.responseText);
```

Использовать такую возможность можно по-разному. Например, фрагмент программы JavaScript может динамически генерироваться Web-сервером, исходя их параметров запроса Web-браузера. Основным недостатком такого способа является необходимость подробнейшего согласования серверного и клиентского кода. При этом связанность серверного и клиентского кодов получается даже более жесткой, чем при загрузке фрагмента HTML-документа (т. е. первого варианта использования объекта `XMLHttpRequest`).

Поэтому чаще применяется подход, при котором функцию `eval()` используют для создания полученного с Web-сервера *JSON-объекта*, т. е. объекта, записанного в *JavaScript Object Notation*. JSON-объект является одновременно и облегченной альтернативой XML, и фрагментом исполняемого кода на языке JavaScript. Например, XML-документ:

```
<colors>
  <color value='13'>оливковый</color>
  <color value='19'>хаки</color>
  <color value='21'>индиго</color>
</colors>
```

можно записать как JSON-объект:

```
[
  {value: "13", color: "оливковый"},
  {value: "19", color: "хаки"},
  {value: '21', color: "индиго"}
]
```

или даже так:

```
{
  "13": "оливковый",
  "19": "хаки",
  "21": "индиго"
}
```

Такая запись может оказаться существенно проще, чем XML-документ. Кроме того, JSON-объект не требует XML-разбора. Вызов функции `eval()` помещает JSON-объект в память интерпретатора JavaScript и возвращает ссылку на этот объект. Для корректной работы функции `eval()`, строка, содержащая JSON-объект, обязательно должна быть взята в круглые скобки:

```
var jsonObject1 = {
  "13": "оливковый",
  "19": "хаки",
  "21": "индиго"
};
var jsonString1 = '{'+
'  "13": "оливковый",'+
'  "19": "хаки",'+
'  "21": "индиго"'+
```

```
'}';
jsonObject2 = eval('(' + jsonString1 + ')');
```

После выполнения функции `eval()` вы можете работать с JSON-объектом при помощи всего спектра возможностей языка программирования JavaScript. Использование JSON-объектов при взаимодействии Web-браузера и Web-сервера считается хорошим приемом. Этот способ по своей логике ближе к использованию XML-документа в качестве ответа Web-сервера, а по формальному признаку является загрузкой фрагмента программы JavaScript. В PHP 5, начиная с версии 5.2.0, реализовано JSON-расширение. Функция `json_encode()` транслирует объект (ассоциативный массив) PHP 5 в строку JSON, а функция `json_decode()` транслирует строку JSON в объект (ассоциативный массив) PHP 5. Очевидно, что JSON-нотация вышла за рамки применения в контексте исключительно JavaScript и превратилась в удобное средство обмена данных в текстовом формате между разнородными системами. Обратите внимание на технологию JSON. Существует мнение, что JSON может реально заменить технологию XML.

Подведем итоги. Взаимодействие Web-браузера с Web-сервером в Ajax-технологии реализуется при помощи объекта `XMLHttpRequest`. Мы рассмотрели три основных варианта взаимодействия Web-браузера и Web-сервера в Ajax-приложениях, когда в качестве ответа Web-браузер получает с Web-сервера:

- фрагмент HTML-документа;
- XML-документ;
- фрагмент программы на языке JavaScript.

Наиболее универсальным и перспективным способом взаимодействия клиента и Web-сервера является использование XML-документов в качестве ответа Web-сервера. Хорошей альтернативой использования XML-документа служит использование JSON-объектов в качестве ответа Web-сервера.

2.2. Основы работы с объектом *XMLHttpRequest*

Объект `XMLHttpRequest` создается при помощи оператора JavaScript `new`. Для каждого запроса к Web-серверу можно создавать новый объект `XMLHttpRequest`. Или можно создать пул объектов `XMLHttpRequest`, чтобы использовать для очередного запроса освободившийся объект из пула. Можно, наконец, обойтись всего одним объектом `XMLHttpRequest`, направляя запросы в очередь и обрабатывая их в порядке поступления одним-единственным экземпляром

объекта XMLHttpRequest. Существуют доводы за и против использования каждой из этих возможностей.

Создание нового объекта для каждого запроса к Web-серверу может привести в некоторых типах Web-браузеров к утечке памяти, потому что объект запроса обычно содержит циклическую ссылку на функцию-обработчик запроса, что затрудняет работу "сборщика мусора". Вариант с использованием пула объектов позволяет ограничить количество создаваемых объектов XMLHttpRequest и облегчает работу "сборщика мусора". Но реализация пула замедляет работу приложения, а самое главное, независимо от количества объектов в пуле, реально, с Web-сервером открывается не более двух соединений (если не менять настройки системы по умолчанию). Поэтому реализация пула объектов XMLHttpRequest на JavaScript применяется сравнительно редко. Реализация пула объектов XMLHttpRequest разобрана в статье автора "Как организовать пул объектов XMLHttpRequest для использования в Ajax-приложениях", которая доступна по URL-адресу **http://www.gotdotnet.ru/LearnDotNet/ASPNET/453076.aspx**.

В связи с ограничением на два активных соединения с Web-сервером идея использовать всего один объект XMLHttpRequest для всех запросов может показаться достаточно обоснованной. Если бы все запросы к Web-серверу были успешными, одного объекта XMLHttpRequest вполне хватило бы для работы Ajax-приложения. Но если хотя бы один запрос зависнет на время, пока не истечет тайм-аут, такой запрос блокирует единственный объект XMLHttpRequest и работу всего приложения.

В наших примерах мы всегда будем использовать новый объект XMLHttpRequest для каждого запроса к Web-серверу, принимая при этом меры, предупреждающие утечку памяти.

После создания объекта XMLHttpRequest оператором new запрос открывается методом open(). Открытие запроса методом open() — это еще не отправка запроса к Web-серверу, которая выполняется методом send(). После открытия запроса можно устанавливать заголовки запроса методом setRequestHeader(). У объекта запроса нет отдельного метода, устанавливающего заголовок Content-Type, который тоже устанавливается методом setRequestHeader(). Самым естественным было бы использовать Content-Type: application/xml (тип text/xml в настоящее время объявлен устаревшим) и разбирать на сервере не параметры запроса, а тело запроса, которое в этом случае должно содержать XML-документ.

Несмотря на то, что объект XMLHttpRequest был разработан как раз для обмена XML-документами, его часто используют для запросов к серверу типа application/x-www-form-urlencoded. Объясняется это тем, что такие запросы более привычны Web-разработчикам, и серверные языки программирования

без дополнительных усилий со стороны Web-программиста разбирают параметры этих запросов, в то время как работа с телом запроса, которое содержит XML-документ, требует, пусть даже самых небольших, дополнительных усилий.

Для корректной работы запросов типа `application/x-www-form-urlencoded` все параметры запроса должны быть правильно закодированы в соответствии со спецификацией MIME-типа `application/x-www-form-urlencoded`. Для этого в JavaScript предусмотрена функция `encodeURIComponent()`. Функция `encodeURIComponent()` отличается от похожей функции `encodeURI()` тем, что дополнительно перекодирует знак + (плюс). Функцию `encodeURIComponent()` следует всегда использовать для перекодирования строковых параметров запроса перед их отправкой на сервер с помощью объекта `XMLHttpRequest`, если запрос имеет тип `application/x-www-form-urlencoded`:

```
var xmlHttpRequest = new XMLHttpRequest();
xmlHttpRequest.open("post","test.php",true);
xmlHttpRequest.setRequestHeader("Content-Type",
                          "application/x-www-form-urlencoded");
xmlHttpRequest.send("count=0&what=" + encodeURIComponent(
  "JavaScript & DOM & CSS + XMLHttpRequest == Ajax-технология?"));
```

Если вы отправите на сервер подобный запрос без использования функции `encodeURIComponent()`, смысл запроса будет сильно искажен, потому что строка, передаваемая в качестве параметра `what`, содержит пробелы, специальные знаки и кириллические символы. Использовать функцию `encodeURIComponent()` для отправки на сервер запросов другого типа, в том числе `application/xml`, не следует.

После отправки запроса на сервер методом `send()` объект `XMLHttpRequest` соединяется с Web-сервером и получает от него ответ, доступ к которому можно получить при помощи двух свойств этого объекта — `responseText` и `responseXML`.

Свойство `responseText` объекта `XMLHttpRequest` содержит ответ Web-сервера в виде строки типа `String`. Свойство `responseXML` объекта `XMLHttpRequest` содержит ответ Web-сервера в виде объекта типа `Document` (иногда его называют `DOMDocument`). Некоторые Web-браузеры не загружают ответ Web-сервера в свойство `responseXML`, если в http-заголовках ответа Web-сервера явно не указано, что тело ответа содержит XML-документ. Этот http-заголовок Web-сервер может отправить автоматически, в том случае, если запрашивается файл с расширением xml. Чтобы понять, как это происходит, рассмотрим фрагмент конфигурационного файла mime.types Web-сервера Apache:

```
application/xhtml+xml                    xhtml xht
application/xml                          xml xsl
application/xml-dtd                      dtd
```

Отправляя файл с расширением xml, Web-сервер Apache, в полном соответствии с конфигурационными параметрами, отправляет и соответствующий http-заголовок `application/xml`. Для этого вам не придется конфигурировать Web-сервер самостоятельно, поскольку такая конфигурация предусмотрена по умолчанию. Получив http-заголовок ответа Web-сервера `Content-Type: application/xml`, Web-браузер распознает, что получен XML-документ, и пытается сделать разбор этого XML-документа. Если полученный XML-документ не содержит ошибок, формируется объект типа `Document` (`DOMDocument`), ссылка на который присваивается свойству `responseXML` объекта `XMLHttpRequest`.

Если XML-документ формируется серверным скриптом, например, на языке программирования PHP 5, заголовок ответа необходимо отправить явно, поскольку запрашиваемый ресурс будет иметь расширение php, либо другое (из соображений безопасности), но не xml:

```
<?php
  #head("Content-Type: text/xml");
  head("Content-Type: application/xml");
?>
```

Если такой заголовок не отправить явно, для некоторых типов Web-браузеров свойство `responseXML` объекта `XMLHttpRequest` будет пустым. Еще одной возможностью сообщить Web-браузеру, что текст ответа содержит XML-документ, является явное объявление XML-документа в первой строке текста этого документа:

```
<?xml version="1.0" encoding="windows-1251"?>
```

Но полагаться только на объявление XML-документа не стоит. Явное задание http-заголовка `Content-Type: application/xml` является единственно надежным способом заставить Web-браузер сформировать в свойстве `responseXML` непустой объект `Document`, содержащий XML-документ.

В приведенном фрагменте кода, объявление XML-документа содержит атрибут `encoding`, который задает кодировку XML-документа. Кроме этого, кодировку можно задать в конфигурации Web-сервера *(см. главу 1)*, конфигурационном файле PHP 5 php.ini и в серверном скрипте PHP 5:

```
<?php
  #head("Content-Type: text/xml");
  head("Content-Type: application/xml; charset='windows1251'");
?>
```

В отличие от явного задания типа `application/xml`, явно задавать кодировку `charset` в скрипте PHP 5 не рекомендуется. Все, что связано с кодировками,

лучше выносить в конфигурационные параметры Web-сервера и конфигурационный файл PHP 5 php.ini. Это позволит легко адаптировать приложения для использования с различными кодировками. Во всяком случае, явное задание кодировки `charset` в PHP-скрипте нужно использовать только в том случае, если все остальные способы испробованы и не принесли положительного результата.

Вопрос с кириллическими кодировками может стать настоящей проблемой для начинающего Web-программиста. Когда Web-браузер отображает текст в неверной кодировке, заочно никто не поможет вам преодолеть эту проблему, особенно если речь идет о чужом хостинге, на котором вы размещаете свои приложения, не имея доступа к конфигурации Web-сервера. Если вы следуете моим рекомендациям по конфигурированию Web-сервера и имеете полный контроль над конфигурацией, то должны избежать проблем с кодировкой. Обеспечение правильной работы с кириллическими кодировками в Ajax-технологии относится, в большей степени, не к сфере программирования, а к сфере конфигурирования серверной среды. Если у вас что-то не ладится с кодировками и вы, как говорится, "застряли", я настоятельно рекомендую отложить на некоторое время вопрос с кодировками и идти вперед. Пока вы еще не освоились с конфигурированием (а нужно ли это Web-программисту вообще?), можно изучать Ajax-технологию, используя только символы латиницы из диапазона кодов символов от 0 до 127, которые без проблем отображаются при любой конфигурации серверной среды.

Я боюсь больше испытывать ваше терпение и перехожу к реальному примеру. При помощи объекта `XMLHttpRequest` можно отправлять запросы к Web-серверу в *синхронном* и *асинхронном* режимах. В большинстве случаев используют асинхронные запросы. Тем не менее, бывают случаи (их не так много), когда без синхронных запросов не обойтись, или асинхронные решения получаются слишком сложными. Это касается таких задач, как загрузка программного кода JavaScript, и некоторые другие случаи, когда для продолжения работы текущего скрипта необходимо, чтобы запрос наверняка отработал.

В большинстве случаев используются асинхронные запросы. Положительным моментом при использовании асинхронных запросов является то, что выполнение текущего скрипта не приостанавливается и интерпретатор JavaScript Web-браузера не блокируется. Платой за использование асинхронных запросов является некоторое усложнение программирования. Мы начнем знакомство с объектом `XMLHttpRequest` на самом простом примере. В данном конкретном примере следовало бы использовать асинхронный запрос, но для этого у нас еще нет достаточных знаний. Первый наш запрос будет синхронным только потому, что такой запрос легче реализовать. Кроме

того, познакомившись с синхронными запросами, вы лучше оцените преимущества асинхронных запросов.

Давайте загрузим при помощи синхронного запроса `XMLHttpRequest` фрагмент HTML-документа и на основании полученного фрагмента обновим текущий документ в Web-браузере. Это соответствует первому варианту взаимодействия Web-браузера с Web-сервером в технологии Ajax *(см. разд. 2.1)*. Вернемся к нашему примеру из *разд. 2.1* и реализуем его в полном объеме. Для начала, создадим файл response.txt (см. компакт-диск) и поместим в него фрагмент HTML-документа, который будет загружаться в Web-браузер при помощи объекта `XMLHttpRequest`.

Мы поместили фрагмент HTML-документа в файл с расширением txt, а не html, подчеркивая тем самым, что этот фрагмент не является самостоятельным HTML-документом. В реальных приложениях формирование фрагментов HTML-документов будет выполняться с использованием серверного программирования, например при помощи PHP 5.

Основной документ testxmlhttp.html (см. компакт-диск) содержит HTML-элемент `FIELDSET`, в который будет помещаться подготовленный на сервере фрагмент. Напоминаю, что все файлы мы договорились сохранять в кодировке Windows-1251, и для этого специально сконфигурировали Web-сервер Apache *(см. главу 1)*. Для сохранения файла в кодировке Windows-1251, необходимо задать специальные опции текстового редактора или использовать пункт меню редактора **Сохранить как** (Save as), где выбрать кодировку Windows-1251. В некоторых редакторах, например Блокноте, для сохранения в кодировке Windows-1251 необходимо в появившемся диалоговом окне **Сохранить как** (Save as) выбрать кодировку ANSI.

В разметке HTML-документа testxmlhttp.html присутствуют HTML-элементы `FIELDSET` и `INPUT type="button"`:

```
<fieldset id="selectColor" style="width:250"></fieldset>
<input type="button"
       onclick="sendColorRequest();" value="Отправить запрос">
```

Для HTML-элемента `FIELDSET` задан атрибут `id="selectColor"`. Атрибут `id`, который называют идентификатором, используется для поиска элемента в объектной модели документа (DOM):

```
var selectColor = document.getElementById("selectColor");
```

Элемент `INPUT type="button"` (кнопка) служит для вызова функции, которая загружает файл response.txt с Web-сервера. Для этого в элементе задан атрибут `onclick="sendColorRequest();"`.

Теперь рассмотрим, как отправляется запрос на Web-сервер. Сначала необходимо создать с учетом кроссбраузерности новый объект запроса:

```
var xmlHttpRequest;
if (window.XMLHttpRequest)
  xmlHttpRequest = new window.XMLHttpRequest();
else
  xmlHttpRequest = new ActiveXObject("Msxml2.XMLHTTP");
```

После того как объект запроса создан, запрос необходимо открыть:

```
xmlHttpRequest.open("get","response.txt",false);
```

Первый параметр запроса задает http-метод запроса, который может принимать различные значения, но чаще всего это будет запрос типа `GET` или `POST`. Второй параметр запроса задает URL-адрес загружаемого ресурса. В данном примере задан относительный путь к ресурсу, поэтому файл response.txt будет загружен из той же папки, из которой загружен основной документ testxmlhttp.html. Третий параметр, который имеет логический тип `false`, определяет, будет ли запрос асинхронным. Значение `false` означает, что запрос будет синхронным (не асинхронным). При выполнении синхронного запроса интерпретатор JavaScript приостанавливает выполнение текущего скрипта до полной загрузки запрашиваемого с Web-сервера документа. Как правило, стандартные HTML-элементы ввода `INPUT`, `TEXTAREA`, `SELECT`, `BUTTON` и `A` (anchor) на это время не блокируется. Но элементы интерфейса, которые разработаны с использованием технологий DHTML и Ajax, будут блокированы вместе с интерпретатором JavaScript. Если соединение с сервером медленное или запросы к серверу интенсивные, синхронные запросы могут очень серьезно повлиять на производительность приложения.

```
xmlHttpRequest.send(null);
var selectColor = document.getElementById("selectColor");
selectColor.innerHTML = xmlHttpRequest.responseText;
```

В приведенном фрагменте при синхронном (не асинхронном) запросе после вызова метода `send()` и до выполнения следующего оператора может пройти достаточно большой промежуток времени. На рис. 2.1 приведена диаграмма синхронного запроса к Web-серверу при помощи объекта `XMLHttpRequest`. На диаграмме видно, что все то время, пока Web-браузер отправляет запрос на сервер, сервер обрабатывает запрос и возвращает ответ Web-браузеру, интерпретатор JavaScript находится в состоянии ожидания. Давайте подумаем, не слишком ли большая роскошь блокировать работу приложения на время работы запроса? Для того чтобы сделать работу приложения более производительной, следует использовать преимущественно асинхронные запросы. Но об этом мы поговорим в следующем разделе.

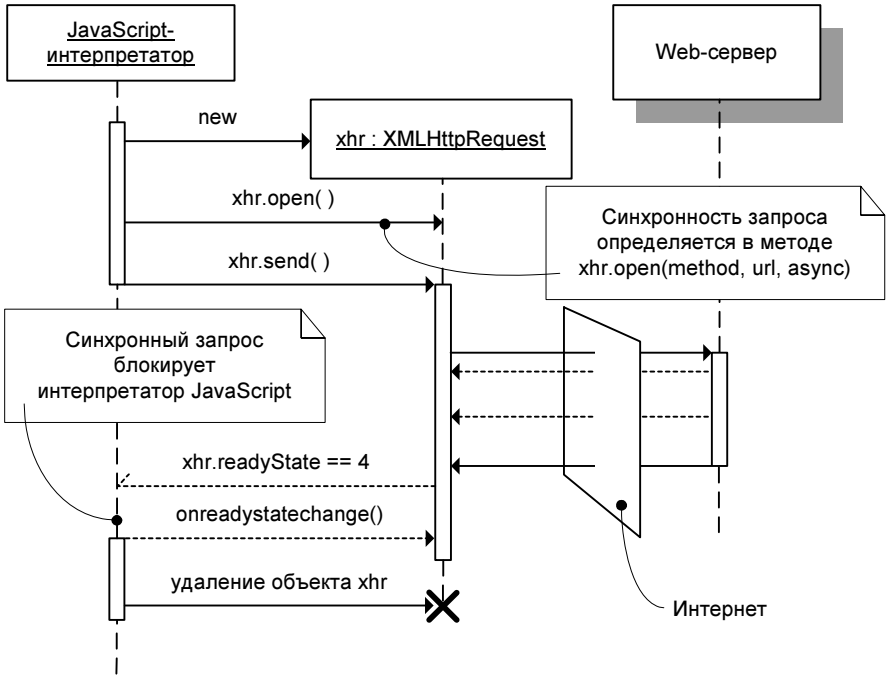


Рис. 2.1. Синхронный запрос при помощи объекта `XMLHttpRequest`

Файлы response.txt и testxmlhttp.html вы можете найти на прилагаемом к книге компакт-диске в папке \bhvajax\xmlhttprequest. Если вы установили Web-сервер так, как это было предложено в *главе 1*, то можете протестировать свою и мою работу, введя в строке адреса Web-браузера:

http://localhost:8080/bhvajax/xmlhttprequest/testxmlhttp.html

После загрузки документа в Web-браузер нажмите кнопку **Отправить запрос** — и можно немного похвалить себя, если все работает правильно. Или заняться отладкой — последнее даже интересней.

Теперь попробуйте раскрыть Web-браузером файл testxmlhttp.html из локальной файловой системы без использования Web-сервера, например, щелкнув по значку файла кнопкой мыши в Проводнике. Похоже, что все работает, но не торопитесь делать выводы. Нажмите кнопку **Отправить запрос**, и вы увидите, что текст, загруженный из файла response.txt в кириллической кодировке Windows-1251, отображается неправильно. А почему? Ведь оба файла мы сохранили в одинаковой кодировке Windows-1251. Тут вам пора познакомиться с одной особенностью работы объекта `XMLHttpRequest`. Этот объект независимо от используемой текущим HTML-документом кодировки

по умолчанию ожидает, что ответ пришел в кодировке UTF-8. Когда файл запрашивался с Web-сервера, который мы сконфигурировали для использования по умолчанию кодировки Windows-1251 *(см. главу 1)*, ответ Web-сервера содержал специальные http-заголовки:

```
Date: Thu, 03 Apr 2008 05:11:26 GMT
Server: Apache/2.2.8 PHP/5.2.5
Last-Modified: Sat, 29 Mar 2008 04:37:27 GMT
Accept-Ranges: bytes
Content-Length: 185
Content-Type: text/plain; charset=windows-1251

200 OK
```

Среди заголовков присутствует и заголовок `Content-Type`, который сообщает объекту `XMLHttpRequest`, что пришел ответ в кодировке Windows-1251, и именно поэтому текст отображается Web-браузером правильно. Когда файл открывается из локальной файловой системы — список заголовков пуст, их просто некому отправить. Поэтому объект `XMLHttpRequest` предполагает, что ответ пришел в кодировке по умолчанию UTF-8. В результате, над текстом из файла response.txt проводится лишняя операция преобразования в кодировку Windows-1251, хотя текст изначально уже имел кодировку Windows-1251.

Без использования Web-сервера можно отобразить загружаемый документ правильно, сохранив файл response.txt в кодировке UTF-8. Для этого необходимо задать специальные опции текстового редактора или использовать пункт меню редактора **Сохранить как** (Save as), где выбрать кодировку UTF-8. Вообще, использовать кодировку UTF-8 в Ajax-приложениях проще, чем любую другую. Если вы решите использовать кодировку UTF-8 для своих Ajax-приложений, первый вопрос, который вам необходимо проработать — как будет обеспечиваться взаимодействие с базой данных. В настоящее время все ведущие системы управления базами данных поддерживают работу с кодировками Unicode и UTF-8. Вы должны убедиться, что работа с базой данных при использовании кодировки UTF-8 в ваших приложениях не вызовет проблем с конкретной системой управления базами данных.

Итак, мы создали простейшую программу, которая направила синхронный запрос `XMLHttpRequest` к Web-серверу и получила ответ в виде фрагмента HTML-документа. Текущий HTML-документ был обновлен в соответствии данными, полученными с Web-сервера. Для этого свойству `innerHTML` HTML-элемента `FIELDSET` было присвоено значение, полученное с Web-сервера.

2.3. Функция-обработчик события *onreadystatechange* объекта *XMLHttpRequest*

Асинхронные запросы к серверу — это те запросы, которые вы будете в основном применять в своих Ajax-приложениях. При работе с асинхронными запросами используется событийно-ориентированная модель программирования. Объект `XMLHttpRequest` отправляет асинхронный запрос на сервер методом `send()` и сразу же, не дожидаясь получения ответа Web-сервера, возвращает управление интерпретатору JavaScript. В процессе получения ответа с Web-сервера, генерируется несколько событий `onreadystatechange`, которые сообщают, что состояние запроса изменилось. В случае успешного прохождения запроса, последнее событие `onreadystatechange` генерируется после полной загрузки ответа Web-сервера в Web-браузер.

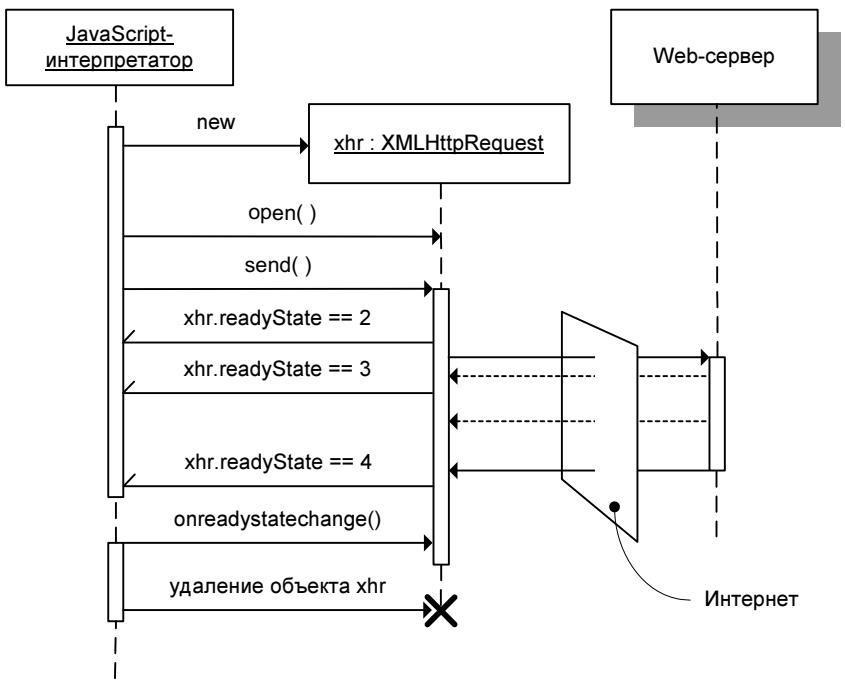


Рис. 2.2. Асинхронный запрос при помощи объекта `XMLHttpRequest`

На рис. 2.2 процесс работы с асинхронным запросом показан в виде диаграммы. Вы видите, что объект запроса создается оператором `new`, открывается методом `open()`, отсылается методом `send()`. После отправки методом

`send()` запрос начинает взаимодействовать с Web-сервером, а интерпретатор JavaScript продолжает работу с текущим скриптом. Так они работают параллельно, асинхронно, и объект `XMLHttpRequest` сообщает о прохождении запроса генерацией событий. События на схеме показаны стрелками с одной засечкой в виде вязального крючка (см. рис. 2.2). Состояние запроса можно определить из свойства `readyState` объекта `XMLHttpRequest`.

Обратите внимание на один очень важный момент. На схеме видно, что в моменты времени, когда поступают события от объекта `XMLHttpRequest`, интерпретатор JavaScript может быть занят выполнением текущего скрипта. Поэтому обработка событий может начинаться не сразу при их поступлении, а только после завершения работы текущего скрипта, когда интерпретатор JavaScript освободиться для обработки поступивших событий. До этого момента события, а это могут быть события сразу от нескольких объектов `XMLHttpRequest`, несколько событий от одного объекта `XMLHttpRequest`, события от пользовательского интерфейса — находятся в очереди и ожидают освобождения интерпретатора JavaScript. Освободиться интерпретатор JavaScript может не только после полного завершения работы текущего скрипта, а еще в случае его приостановки SUSPEND. Например, выполнение функции `window.alert()` вызывает приостановку SUSPEND текущего скрипта и дает возможность отработать обработчикам событий, которые находятся в очереди. Поэтому функцию `window.alert()` не рекомендуется использовать в Ajax-приложениях — это может непредсказуемо повлиять на порядок, в котором будут вызываться функции-обработчики событий. Замечу сразу, что явно вызывать приостановку SUSPEND текущего скрипта нельзя — такого оператора в JavaScript не существует.

После получения ответа Web-сервера вызывается функция-обработчик события `onreadystatechange` (см. рис. 2.2), которое генерируется при смене состояния запроса `XMLHttpRequest`. Вызов функции-обработчика события может происходить не сразу при поступлении соответствующего события, а только после высвобождения интерпретатора JavaScript, который может быть занят в момент поступления ответа Web-сервера выполнением текущего скрипта. Я особо обращаю ваше внимание на этот момент, потому что это не укладывается в стандартную модель асинхронной обработки событий и вызывает массу вопросов у начинающих Ajax-программистов. Дело в том, что Web-браузер является многопоточной средой, поэтому вы можете продолжать взаимодействовать с интерфейсом, в то время как выполняется загрузка изображений в HTML-элементы `IMG` или в то время, когда объект `XMLHttpRequest` взаимодействует с Web-сервером в асинхронном режиме. В отличие от Web-браузера, интерпретатор JavaScript допускает только последовательную работу программного кода JavaScript. Если в текущий

момент времени интерпретатор JavaScript занят выполнением скрипта, все обработчики событий становятся в очередь и ожидают освобождения интерпретатора. Таким образом, с точки зрения многопоточного программирования интерпретатор JavaScript является синхронизирующим объектом или монитором. Хотя в большинстве случаев этот момент можно не учитывать.

Функцию-обработчик события `onreadystatechange` объекта `XMLHttpRequest` обычно называют *`onreadystatechange`-функцией*. Если вы уже привыкли к названию объекта `XMLHttpRequest`, запомнить название события `onreadystatechange` и соответствующей этому событию `onreadystatechange`-функции не составит труда, тем более что, в отличие от объекта `XMLHttpRequest`, в этом названии используются только буквы в нижнем регистре. В качестве `onreadystatechange`-функции может выступать любая функция с произвольным именем, часто даже без имени — анонимная функция, разработанная на языке JavaScript. Эта функция вызывается несколько раз в течение одного запроса при *смене состояния* объекта запроса `XMLHttpRequest`. Смена состояния объекта запроса `XMLHttpRequest` наступает при вызове методов `open()`, `send()` объекта `XMLHttpRequest`, а также при поступлении ответа с Web-сервера, когда загружены все заголовки ответа и когда загружен весь запрошенный документ. Наиболее важным является последний вызов `onreadystatechange`-функции, когда весь запрошенный с Web-сервера документ загружен в Web-браузер и доступен как свойства `responseText` или `responseXML` объекта `XMLHttpRequest`.

После создания объекта `XMLHttpRequest` оператором `new` и до отправки запроса методом `send()` всем асинхронным (и только асинхронным) запросам необходимо назначить функцию-обработчик ответа Web-сервера. Для этого свойству `onreadystatechange` объекта `XMLHttpRequest` присваивается ссылка на функцию-обработчик ответа Web-сервера, которую мы условились называть `onreadystatechange`-функцией:

```
function callbackFunction(){...}
var xmlHttpRequest = new XMLHttpRequest();
...
xmlHTTPRequest.onreadyststechange = callbackFunction;
...
xmlHTTPRequest.send(...);
```

> ***Замечание***
>
> Назначать функцию-обработчик события `onreadystatechange` объекту `XMLHttpRequest` можно в любой части кода. Конечно, объект запроса до этого должен быть уже создан оператором `new`. Ранее — до вызова метода `open()` — назначение функции-обработчика позволяет обработать события, связанные с открытием и отправкой запроса. Хотя я рекомендовал назначать

функцию-обработчик до отправки запроса методом `send()`, на самом деле это можно сделать и позже. Некоторые библиотеки вовсе не назначают функцию-обработчик и периодически запрашивают состояние `readyState` всех отправленных запросов. Разумеется, вы должны знать, зачем это вам необходимо. В подавляющем большинстве случаев можно рекомендовать назначать функцию-обработчик непосредственно перед отправкой запроса методом `send()`. Однако я не хотел бы, чтобы такая рекомендация стала для вас догмой и ограничила в выборе средств.

Обратите внимание, что в операторе присваивания ссылки на функцию `callbackFunction()` свойству `onreadystatechange` имя функции записывается без списка фактических параметров. Если бы список фактических параметров присутствовал, этот оператор присваивания имел бы совершенно другой смысл: функция была бы вызвана, и результат, возвращенный оператором `return` функции, был бы присвоен свойству `onreadystatechange` объекта запроса. Такой вариант мы будем очень скоро использовать, но совершенно особым образом. В этом случае функция должна возвращать ссылку на другую функцию, которая и станет `onreadyststechange`-функцией для данного объекта запроса `XMLHttpRequest`. Такой усложненный код позволит нам не только назначить функцию-обработчик, но и передать в функцию-обработчик параметры при помощи замыкания (closure). К этому вопросу мы очень скоро вернемся.

Функция-обработчик события `onreadystatechange` в процессе отправки запроса к Web-серверу и получения ответа с Web-сервера будет вызвана неоднократно. Точное количество вызовов зависит от типа Web-браузера и успешности прохождения запроса. Но гарантировано, что в случае успешного прохождения запроса функция-обработчик будет вызвана после полной загрузки документа ровно один раз в состоянии готовности COMPLETED.

Название функции означает *On Readystate Change — при смене состояния готовности (запроса)*. Состояний готовности запроса всего пять:

- 0 UNINITIALIZED — объект запроса создан оператором `new`, но запрос не открыт методом `open()`;
- 1 LOADING — запрос открыт методом `open()`, но не отправлен методом `send()`;
- 2 LOADED — запрос отправлен методом `send()` и получены заголовки ответа Web-сервера;
- 3 INTERACTIVE — ответ Web-сервера доступен, но загружен не полностью;
- 4 COMPLETED — ответ Web-сервера загружен полностью и доступен как свойства `responseText` и `responseXML` объекта `XMLHttpRequest`.

Для разработчиков Ajax-приложений наиболее интересным является состояние COMPLETED, когда ответ Web-сервера доступен и полностью загружен в Web-браузер. Гарантировано, что в случае удачного прохождения запроса функция-обработчик события `onreadystatechange` в состоянии COMPLETED будет вызвана ровно один раз. В случае неудачного запроса функция-обработчик в состоянии COMPLETED может быть вызвана, а может быть и не вызвана. Что же касается вызовов функции-обработчика в состоянии, отличном от COMPLETED, то такие вызовы при успешном прохождении запроса могут или не наступить, или наступить один раз, или наступить более одного раза. Все зависит от типа Web-браузера и других причин. Поэтому строить логику приложения на обработке состояний, отличных от COMPLETED, следует очень осторожно. В книге мы будем обрабатывать только событие `onreadystatechange` в состоянии COMPLETED.

Использование функции обработчика события `onreadystatechange` очень похоже на использование обработчиков событий для HTML-элементов. Вспомните хотя бы обработку события `onclick` для HTML-элементов `DIV` компонента "Аккордеон" из *главы 1*. Когда загрузка ответа Web-сервера полностью завершена — генерируется событие `onreadystatechange` в состоянии COMPLETED и вызывается функция-обработчик события `onreadystatechange`.

Если вы хорошо знакомы с тонкостями работы обработчиков событий для HTML-элементов, то можете ошибочно предположить, что после выполнения оператора

```
xmlHttpRequest.onreadystatechange = callbackFunction;
```

в теле функции `callbackFunction()` объект `xmlHttpRequest` будет доступен как неявный параметр `this`. Но это не так! Вернее, на сегодняшний день это не так для части Web-браузеров. Рост популярности технологии Ajax побудил разработчиков Web-браузеров сделать работу с объектом `XMLHttpRequest` более привычной для Web-программистов и обеспечить доступ к объекту запроса через неявный параметр `this` внутри функции-обработчика события `onreadystatechange`. Последние версии некоторых Web-браузеров обеспечивают такую возможность, но мы не будем ее использовать, чтобы наши Web-приложения работали во всех наиболее распространенных версиях Web-браузеров.

Как же из тела `onreadyststechange`-функции можно получить доступ к объекту `XMLHttpRequest`, если Web-браузер не обеспечивает доступ к объекту запроса через неявный параметр `this`? Конечно, этот объект можно сохранить в глобальной переменной, которая будет доступна в любом месте кода JavaScript и в теле `onreadyststechange`-функции. Но что же делать, если вы работаете с двумя, тремя объектами `XMLHttpRequest` одновременно? В этом

случае вариант с глобальной переменной не будет работать. Да и вообще глобальными переменными не стоит злоупотреблять.

Я уверен, что пройдет еще некоторое время и объект `XMLHttpRequest` будет доступен для большинства Web-браузеров в теле `onreadystatechange`-функции как неявный параметр `this`. Но сейчас мы должны обеспечить все наиболее популярные Web-браузеры надежным способом доступа к объекту `XMLHttpRequest`. Сделать это можно при помощи так называемого *замыкания* (closure). Для того чтобы понять, что такое замыкание, необходимо ближе познакомиться с функциями JavaScript. Это мы и сделаем в следующем разделе, а сейчас скажу вам, на что следует обратить особое внимание. Функции JavaScript имеют особенности, отличающие их от функций любого другого популярного языка программирования. Если язык LISP называют процессором списков, JavaScript можно смело назвать процессором функций. При помощи функций JavaScript можно проделывать трюки, сравнимые только с использованием регулярных выражений Perl. Это может показаться немного фантастическим, что через 10 лет после своего создания JavaScript начинает завоевывать популярность как язык программирования, на котором можно создавать красивый программный код.

2.4. Функции, объекты, конструкторы и прототипы в JavaScript

В настоящем разделе будут рассмотрены особенности использования функций в JavaScript, которые могут показаться немного необычными не JavaScript-программистам. Например, их может удивить тот факт, что функции в JavaScript являются объектами и имеют свойства и методы, что на функциях-конструкторах построена система пользовательских объектов, реализующих объектно-ориентированные возможности JavaScript. Без рассмотрения особенностей работы с функциями в JavaScript, код работы с объектом `XMLHttpRequest` и функции `sendRequest()`, которую мы будем разрабатывать далее в этой главе, может показаться загадочным. Магистральным направлением в изучении функций JavaScript в настоящем разделе является подготовка к использованию замыканий (closure).

Мы уже использовали со словом "функция" уточняющие определения — функция-обработчик события, функция-метод объекта, функция-конструктор объекта, функция как объект. На самом деле, все это просто функции JavaScript. Любая функция в языке программирования JavaScript является объектом, может быть использована в качестве конструктора при создании пользовательского объекта оператором `new`, может быть привязана к объекту

в качестве метода, может быть назначена обработчиком события. Синтаксис определения для функций всегда будет одним и тем же. Давайте рассмотрим его.

Для создания функции в JavaScript применяют ключевое слово `function`. Использовать его можно в двух контекстах — для *определения функции* (function definition) и в *выражении функции* (function expression).

Определение функции выглядит так:

```
function someFunction1(param1, ...) {
 ...
}
```

Заданное в *определении функции* имя функции становится доступным в текущем контексте и может быть использовано в любом месте текущего скрипта (в том числе до определения функции в тексте программы).

В противоположность этому, *выражение функции* создает функцию только в момент выполнения кода:

```
var someFunction2 = function someFunction3(param1, ...) {
 ...
}
```

Поэтому использовать имя *someFunction2* в этом примере можно только после выполнения оператора, содержащего *выражение функции*.

В текущем контексте к функции, которая определена с помощью *выражения функции*, можно обращаться только по имени переменной, в которой сохранена ссылка на функцию — в нашем примере это *someFunction2*. Как же может быть использовано имя функции из выражения функции, т. е. *someFunction3*? Ответ будет немного неожиданным. Имя функции *someFunction3* доступно только в теле самой функции *someFunction3*, а в текущем контексте оно будет недоступно. Поскольку такой вариант использования будет полезен преимущественно для организации рекурсии, в *выражении функции* часто не задают имя функции и используют выражение *анонимной функции*:

```
var someFunction4 = function(param1, ...) {
 ...
}
```

Несмотря на то, что спецификация JavaScript различает *определение функции* и *выражение функции*, в литературе все еще редко используют термин *выражение функции*, которое не совсем точно называют *определением анонимной функции*. Я так подробно остановился на этом вопросе, чтобы термин *"выражение функции"* стал для вас более привычным.

С функциями в JavaScript тесно связаны понятия локального контекста и видимости переменных. Правила видимости переменных бывают достаточно сложны в различных языках программирования, но в JavaScript они предельно просты и лаконичны. Приведу их.

В JavaScript различают *глобальный контекст* и *локальный контекст*. Локальный контекст составляют имена, определенные в теле некоторой функции. К локальному контексту функции принадлежат формальные параметры функции, имена переменных, объявленных в теле функции с ключевым словом `var`, и имена внутренних функций, которые определены в теле функции. К глобальному контексту принадлежат все имена переменных и функций, определенные вне тела какой-либо функции.

Как правило, переменные должны объявляться с ключевым словом `var`:

```
var someVar1;
```

Переменные и имена функций, объявленные вне тела какой-либо функции, являются глобальными переменными и видны из любого контекста, в частности, в теле любой функции:

```
function testGlobalVar1() {
  alert(globalVar1);
}
var globalVar1 = "global";
testGlobalVar1();
```

В теле функции `testGlobadVar1()` переменная, определенная в глобальном контексте `globalVar1`, доступна.

Каждая функция создает новый локальный контекст. Это означает, что переменные, определенные в теле функции с ключевым словом `var`, будут доступны только в теле функции и недоступны вне тела функции. Можно определять локальные переменные и внутренние функции с тем же именами, что и глобальные переменные и функции:

```
function testLocalVar1() {
  var globalVar1 = "local";
  var localVar2 = "local";
}
var globalVar1 = "global";
testLocalVar1();
alert(glovalVar1);       // "global"
alert(typeof localVar1); // "undefined"
```

Если вы определили внутри функции локальную переменную, формальный параметр или внутреннюю функцию, одноименные глобальные переменные

и функции становятся недоступными в теле самой функции и всех внутренних функций. При этом безразлично, переопределяется имя функции или имя переменной.

К локальному контексту функций принадлежат не только переменные, определенные в теле функции, но и формальные параметры функции. А также к локальному контексту принадлежат внутренние функции.

Внутренняя функция — это просто функция, определенная в теле другой функции. При этом может быть использовано как определение внутренней функции, так и выражение внутренней функции (которое нестрого называют определением анонимной внутренней функции):

```
function outerFunction1(){
  function innerFunction1(){
    var localVar1;
    function innerFunction2(){
    }
  }
  var innerFunction2 = function{
    var localVar1;
  };
}
```

В примере показано, что в теле функции `outerFunction1()` определены две внутренние функции: `innerFunction1()` и `innerFunction2()`. В теле внутренних функций допустимо определять внутренние функции любой степени вложенности. Правила видимости имен переменных, формальных параметров и функций сохраняются и для внутренних функций произвольной степени вложенности, и они очень лаконичны:

- в теле внутренней функции доступны по имени формальные параметры, переменные и внутренние функции всех охватывающих функций и имена из глобального контекста, если они не переопределены в теле внутренней функции;
- в теле охватывающих функций недоступны по имени формальные параметры, локальные переменные и имена функций, определенных в теле внутренней функции;
- аналогично, в глобальном контексте недоступны по имени формальные параметры, локальные переменные и имена функций, определенных в теле произвольной функции;
- если в теле функции присваивается значение переменной, которая не объявлена в теле функции с ключевым словом `var` — будет инициирован

поиск переменной с таким именем во всех охватывающих функциях, пока не будет достигнут глобальный контекст. Если переменная с указанным именем не была определена в охватывающих функциях и в глобальном контексте — такая переменная будет создана в глобальном контексте и останется доступной после окончания работы функции.

Последнее правило показывает, что явное объявление переменных с ключевым словом `var` позволяет уменьшить риск написать неправильную программу. Все же приписывание переменных к глобальному контексту для переменных без ключевого слова `var` является потенциально опасной особенностью JavaScript, потому что не всегда использование этой возможности является осознанным. Чаще так происходит при ошибке, когда ключевое слово `var` просто пропущено, или при элементарной опечатке, когда с ключевым словом `var` объявлена одна переменная, а в тексте программы используется другая переменная с похожим именем (ошибка в написании имени переменной). Чтобы избежать подобных казусов, можно посоветовать использовать редактор кода, который позволял бы отслеживать правильное употребление имен переменных.

Рассмотрев глобальный и локальный контексты, мы приготовились к рассмотрению вопроса о *замыканиях*, которые в JavaScript используются очень активно, особенно в функциях-обработчиках событий, таких как `onreadystatechange`-функция.

При каждом вызове функции создается новый локальный контекст функции, который доступен в теле функции, и становится *недоступным по имени* после возврата из функции. Я недаром выделил фразу: *недоступным по имени*, потому что к этому локальному контексту все же можно получить доступ и после возврата из функции, но совершенно особым образом — не по имени, а с использованием замыкания. Суть использования замыкания заключается в том, что ссылка на внутреннюю функцию из локального контекста некоторой функции может быть возвращена оператором `return` и сохранена в глобальном контексте. "Сборщик мусора" не уничтожает такой локальный контекст, пока ссылка на внутреннюю функцию остается доступной. Существуют языки программирования, в которых замыкания работают аналогично, при этом отличаясь в деталях. Существуют языки программирования, в которых замыкания не допускаются. В JavaScript локальный контекст внутренней функции, ссылка на которую сохранена в глобальном контексте, запоминается с теми значениями локальных переменных, которые были на момент выхода из функции.

Разберем работу замыкания на простом примере. Для сравнения, сначала приведем код, в котором не используется замыкание:

```
function outerFunction(a, b){
  var innerFunction = function(){
```

```
    a++;
    return a + b;
  };
  alert(innerFunction());
}
outerFunction(1, 1); // 3
```

Как работает этот пример? То, что в теле анонимной функции `innerFunction()` доступны формальные параметры `a` и `b` охватывающей функции, следует из правил видимости имен JavaScript. Имена из области видимости охватывающей функции всегда доступны для внутренней функции. После выхода из функции `outerFunction()` локальный контекст, в котором определены имена `a`, `b`, `innerFunction()`, станет недоступным по имени. Более того, память, выделенная под эти переменные, будет освобождена для дальнейшего использования. Все идет по плану. Но давайте изменим всего лишь одну деталь. Пусть функция `outerFunction()` возвратит ссылку на функцию `innerFunction()`. В этом случае имя переменной `innerFunction` попрежнему останется недоступным после выхода из функции `outerFunction()`, но ссылку на объект анонимной функции можно будет сохранить в переменной из вызывающего функцию `outerFunction()` контекста:

```
function outerFunction(a, b){
  var innerFunction = function(){
    a++;
    return a + b;
  };
  return innerFunction;
}
var closureFunction1 = outerFunction(1, 1);
var closureFunction2 = outerFunction(10, 10);
alert(closureFunction1()); // 3
alert(closureFunction1()); // 4
alert(closureFunction2()); // 21
```

Ссылка на анонимную функцию в примере присваивается двум переменным `closureFunction1` и `closureFunction2`. Эти переменные принадлежат к контексту, в котором была определена и вызвана функция `outerFunction()`. Теперь переменные `closureFunction1` и `closureFunction2` доступны в текущем контексте и содержат ссылку на функцию `innerFunction()`. В свою очередь, функция `innerFunction()` обращается к значениям формальных параметров функции `outerFunction()` — `a` и `b`. Следовательно, "сборщик мусора"

не уничтожает `innerFunction()`, `a` и `b` — пока на `innerFunction()` существует, по крайней мере, одна ссылка. В нашем примере для каждой из переменных `closureFunction1` и `closureFunction2` сохраняется свой набор локального контекста функции `innerFunction`. Это и называется замыканием.

Замыкания мы будем постоянно использовать в примерах книги. Поэтому внимательно разберите этот пример. Без замыканий при написании Ajax-приложений не обойтись. Но использовать замыкания следует очень осторожно. Непродуманное использование замыканий может привести к утечке памяти. Посудите сами, замыкание сохраняет локальный контекст, который при других обстоятельствах был бы удален "сборщиком мусора" сразу после выхода из функции. Нередки случаи непродуманного использования замыкания в цикле или с рекурсивным вызовом, когда очень скоро исчерпывается вся свободная оперативная память — это, конечно, происходит в результате грубых ошибок разработчика. Очень часто замыкания хранят циклические ссылки, усложняющие работу "сборщика мусора". Поэтому использовать замыкания нужно только там, где это действительно необходимо. Создавать приложения в стиле сплошных замыканий просто недопустимо.

Имя любой функции может использоваться в четырех контекстах:

- при определении функции (а также в выражении функции, о чем было сказано ранее);
- при вызове функции (со списком фактических параметров, который может быть пустым);
- как ссылка на объект функции (без списка фактических параметров);
- с оператором `new` (в качестве функции-конструктора объекта).

Поскольку функции являются полноценными объектами JavaScript, со ссылкой на объект функции можно производить все действия, которые доступны для ссылки на любой объект JavaScript, в том числе:

- присваивать значение объекта-функции переменным;
- присваивать значение объекта-функции свойствам объекта и использовать функции как функции-методы объекта;
- передавать значение объекта-функции в качестве фактического параметра другим функциям;
- вызывать методы объекта типа `Function`;
- запрашивать и определять значения свойств объекта типа `Function`.

С переменными, свойствами объектов и формальными параметрами, которые имеют тип ссылки на функцию, можно производить все действия, которые

допустимы для функций, в частности, их можно вызывать со списком фактических параметров.

Рассмотрим возможные варианты действий с функциями на простом примере:

```
function someFunction1(){
  alert("someFunction1");
}
function someFunction2(someFunction3){
  if (typeof someFunction3 == "function")
    someFunction3();
}
var someFunction4 = someFunction1;
someFunction1();
someFunction2(someFunction1);
somefunction4();
```

Мы определили функцию `someFunction1()`, которая выводит окно с сообщением. Ссылку на эту функцию мы сохранили в переменной `someFunction4`, которую теперь можно вызвать как функцию с пустым списком параметров: `someFunction4()`. Далее мы определили функцию `someFunction2()`, которой передается в качестве фактического параметра функция `someFunction1`, и в теле функции `someFunction2()` эта функция вызывается с пустым списком параметров.

Теперь разберемся с использованием функций в качестве функций-методов объектов JavaScript. Наиболее общим типом объектов JavaScript является тип `Object`, который можно создать либо при помощи оператора `new Object()`, либо при помощи специального синтаксиса, который называется *JSON-нотацией* (JavaScript Object Notation):

```
var obj1 =  new Object();
var obj2 = {}; // JSON-нотация
```

Теперь специальный JavaScript-оператор `typeof` для этих переменных вернет строковое значение `"object"`, а оператор `instanceof` при сравнении со встроенным JavaScript типом `Object` вернет значение `true`:

```
alert(typeof obj1);             // "object"
alert(obj1 instanceof Object); // true
alert(typeof obj2);             // "object"
alert(obj2 instanceof Object); // true
```

В качестве конструктора объекта, который используется с оператором `new`, кроме встроенных объектов, может выступать любая функция JavaScript.

Такие объекты называются пользовательскими объектами, чтобы отличить их от встроенных объектов JavaScript:

```
function SomeObject(){
}
var obj3 = new SomeObject();
alert(typeof SomeObject);          // "function"
alert(typeof obj3);                // "object"
alert(obj3 instanceof Object);     // true
alert(obj3 instanceof SomeObject); // true
```

Обратите внимание, что оператор `typeof` вернет для пользовательского объекта строковое значение `"object"`, а оператор `instanceof` — логическое значение `true` и при сравнении с типом `Object`, и при сравнении с типом `SomeObject`.

К любому объекту JavaScript можно привязывать свойства и функции-методы. Свойства и функции-методы можно привязывать к объекту, ссылка на которой сохранена в переменной. В теле функции-конструктора или другой функции-метода новую функцию-метод или новое свойство можно привязать к неявному параметру `this`, который содержит ссылку на текущий объект. Но лучше всего привязывать методы и общие для всех объектов свойства к *объекту-прототипу*, о котором мы поговорим далее в этом разделе. Все эти возможности рассмотрим на простом примере:

```
function functionAlert(alertString){
  alert(alertString);
}
function SomeObject(){
}
var obj3 = new SomeObject();
obj3.methodAlert = functionAlert;
obj3.methodAlert("Функция-метод привязана к переменной");
```

В приведенном коде мы привязали функцию к объекту, ссылка на который сохранена в переменной `obj3`. Такое решение нельзя признать удачным, поскольку после создания каждого объекта `SomeObject` необходимо будет вызывать код инициализации. Очевидное решение — поместить код инициализации в функцию-конструктор. В теле функции-конструктора создаваемый объект доступен как неявный параметр `this`. После привязки функции-метода к объекту текущий объект будет доступен как неявный параметр `this` в теле функции-метода:

```
function functionAlert(){
  alert(this.propertyString);
```

```
}
function SomeObject(alertString){
    this.propertyString = alertString;
    this.methodAlert = functionAlert;
}
var obj3 = new SomeObject("Функция-метод привязана к this");
obj3.methodAlert();
var obj4 = new SomeObject("И строка сообщения привязана к this");
obj4.methodAlert();
```

Очевидно, что инициализация объекта внутри функции-конструктора позволила упростить код инициализации объекта. Теперь можно создавать любое количество объектов типа `SomeObject`, и все они будут обладать необходимыми свойствами и функциями-методами сразу после выполнения оператора `new`. При этом код инициализации задается в одном месте — в теле функции-конструктора.

Все это было бы просто прекрасно, но существует еще более красивое решение с использованием *объекта-прототипа*. Объектно-ориентированную модель JavaScript называют моделью, *основанной на прототипах*, в противовес распространенной в подавляющем большинстве объектно-ориентированных языков программирования модели, *основанной на классах*. В JavaScript слово `class` является зарезервированным, но в текущих версиях языка не используется.

Модель, основанная на прототипах, предполагает использование объекта-прототипа для создания пользовательских объектов. В качестве объекта-прототипа может выступать любой объект JavaScript. После того как функции-конструктору назначен объект-прототип, все пользовательские объекты, созданные при помощи этой функции-конструктора, будут обладать свойствами и функциями-методами своего объекта-прототипа. Слово "прототип" достаточно точно выражает происходящее.

Привязка функций-методов в функции-конструкторе к неявному параметру `this` приводит к нерациональному использованию памяти, поскольку каждый экземпляр объекта будет содержать ссылку на все свои функции-методы. Использование объекта-прототипа решает эту проблему. Каждая функция-конструктор (фактически, каждая функция JavaScript) имеет свойство `prototype`, в котором содержится ссылка на объект-прототип. При создании функции в свойстве `prototype` уже хранится ссылка на экземпляр объекта-прототипа, имеющий встроенный тип `Object`. Можно этот объект заменить новым объектом, но при этом часть функциональности будет утрачена (впрочем, это можно поправить). Перейдем к рассмотрению примера:

```
function functionAlert(){
  alert(this.propertyString);
```

```
}
function SomeObject(alertString){
    this.propertyString = alertString;
}
SomeObject.prototype.methodAlert = functionAlert;
var obj3 = new SomeObject("Функция-метод привязана к прототипу");
obj3.methodAlert();
var obj4 = new SomeObject("А строка сообщения привязана к this");
obj4.methodAlert();
```

В приведенном примере функция-метод привязывается уже не в функции-конструкторе к неявному параметру `this`, а после определения функции-конструктора к свойству функции-конструктора `prototype`, т. е. к экземпляру объекта-прототипа, которым обладает каждая функция JavaScript.

Слышу ваши удивленные возгласы: функции имеют свойства? Да, и свойства, и методы! Дело в том, что, как я и говорил, язык программирования JavaScript можно смело называть процессором функций. Не удивительно, что функции в JavaScript являются полноценными объектами и подобно всем объектам имеют свойства и методы.

Свойство `prototype` — это особенное свойство, которое играет важную роль в объектно-ориентированной модели JavaScript, основанной на прототипах. Если вы обращаетесь к методу или свойству экземпляра пользовательского объекта, то имя этого метода или свойства интерпретатор JavaScript пытается найти в экземпляре текущего объекта (можно сказать, в `this`). Если имя в экземпляре текущего объекта не найдено, интерпретатор JavaScript продолжает поиск имени метода или свойства в объекте-прототипе. Но и это не все. Экземпляр объекта-прототипа тоже может иметь свой объект-прототип, и интерпретатор JavaScript продолжит поиск по всей цепочке объектов-прототипов, пока не найдет нужный метод или свойство по имени или не достигнет корневого объекта `Object`. Если имя не будет найдено ни в одном из объектов-прототипов — результат обращения к свойству будет иметь значение `undefined`, а в результате обращения к несуществующей функции-методу будет сгенерирована ошибка, и выполнение скрипта может прерваться, если ошибку не перехватить в блоке `try-catch`.

Объекты-прототипы используются для организации наследования. Наследование не определено явно в спецификации JavaScript, но и не противоречит ей. У части программистов упоминание о наследовании в JavaScript вызывает бурную реакцию неприятия. Но, тем не менее, некоторые библиотеки используют такой подход, и мы разберем его:

```
function ParentObject(){}
function ChildObject(){}
```

```
ChildObject.prototype = new ParentObject();
ChildObject.prototype.constructor = ChildObject;
obj5 = new ChildObject();
alert(obj5 instanceof ChildObject);   // true
alert(obj5 instanceof ParentObject);  // true
```

В качестве объекта-прототипа для `ChildObject` задан экземпляр объекта `ParentObject`. Это позволяет унаследовать все методы объекта `ParentObject` для объекта `ChildObject`. Платой за такую вольность является то, что истинный объект-прототип функции-конструктора `ChildObject` будет заменен экземпляром объекта `ParentObject`. Свойство `constructor` объекта `ParentObject` сразу после создания будет указывать на функцию-конструктор `ParentObject`. Значит, мы явно должны установить свойство `constructor` для экземпляра объекта `ParentObject` в значение `ChildObject`, если собираемся использовать это значение в своих скриптах.

Рассмотрим использование функций JavaScript в качестве объектов JavaScript. Это мы делаем, чтобы подготовиться во всеоружии к написанию функции `sendRequest()`, где нам придется использовать весь спектр рассматриваемых в настоящей главе приемов работы с функциями.

Для обычного вызова функции необходимо указать имя функции со списком фактических параметров, который может быть и пустым. Это самый привычный для нас способ вызова функции. Но JavaScript реализует и более гибкие решения. Вспомним, что все функции JavaScript являются объектами и имеют свойства и методы. Два очень активно используемых Ajax-программистами метода объекта функции, `apply()` и `call()`, позволяют вызвать функцию альтернативным способом. Сразу же рассмотрим самый простой пример:

```
function someFunction(){
  alert("someFunction");
}
someFunction();
someFunction.call();
someFunction.apply();
```

В данном примере функция `someFunction()` будет вызвана три раза. И все эти три вызова не будут различаться (забегая вперед, скажу, что это связано с тем, что методы `call()` и `apply()` вызваны с пустым списком фактических параметров). Что же нового может дать вызов функции методом `call()` и `apply()` по сравнению с обычным вызовом функции?

Как мы уже знаем, функция может быть привязана в качестве функции-метода к объекту JavaScript. При этом объект становится доступным внутри своей функции-метода через неявный параметр `this`. Вызывая функцию при

помощи методов `call()` и `apply()`, мы принудительно выполняем функцию в качестве функции-метода объекта, который задан в качестве первого фактического параметра функции `call()` или `apply()`. Разберем простой пример:

```
function SomeObject(alertString){
  this.propertyString = alertString;
}
function someFunction(){
  if (this && this.propertyString)
    alert(this.propertyString);
  else
    alert("this.propertyString is undefined");
}
var obj6 = new SomeObject("test call");
var obj7 = new SomeObject("test apply");
someFunction.call(obj6);
someFunction.apply(obj7);
someFunction();
```

В приведенном примере мы не привязываем к создаваемым объектам типа `SomeObject` ни одной функции-метода. При помощи методов `call()` и `apply()` мы вызываем функцию `someFunction()` *в контексте объекта*, который задан в качестве первого фактического параметра функций `call()` и `apply()`. Более красноречивым является имя функции `apply()`, т. е. *применить* функцию к объекту, заданному в качестве первого фактического параметра функции `apply()`. В данном примере вызовы функции при помощи методов `call()` и `apply()` выполняют одинаковые действия — вызывают функцию в качестве метода объектов `obj6` и `obj7`. Но вот функция `someFunction()`, вызванная просто с пустым списком фактических параметров, работает иначе. В этом случае текущий объект для функции не задан. В зависимости от реализации Web-браузера объект `this`, вызванный внутри функции без привязки к конкретному объекту, может содержать ссылку на объект-функцию `someFunction()` или на глобальный объект `window`, но использовать такое поведение в логике приложения не стоит.

Метод `call()` вызывает функцию в контексте объекта, переданного функции в качестве первого параметра. Это означает, что неявный параметр `this` внутри функции будет содержать ссылку на объект, переданный функции `call()` в качестве первого фактического параметра. Аналогично ведет себя и функция `apply()`. Единственная разница заключается в том, что функции `call()` передаются фактические параметры функции как при обычном вызове функции (через запятую), но сдвинутые на один параметр вправо,

т. к. первым параметром всегда должен быть объект контекста. А функция `apply()` вызывается всегда с двумя параметрами: первый параметр — объект, в контексте которого вызывается функция, а второй параметр — массив типа `Array` фактических параметров, передаваемых в функцию:

```
function SomeObject(alertString){
  this.propertyString = alertString;
}
function someFunction(anotherString, moreString){
  if (this && this.propertyString)
    alert(this.propertyString);
  else
    alert("this.propertyString is undefined");
  alert(anotherString);
  alert(moreString);
}
var obj6 = new SomeObject("test call");
var obj7 = new SomeObject("test apply");
someFunction.call(obj6, "someString", "allwaysString");
var array1 = Array("someString", "allwaysString");
someFunction.apply(obj7, array1);
someFunction();
```

Сейчас вам может показаться, что использование метода `call()` более удобно и понятно. Но со временем вы оцените, что гораздо удобнее использовать как раз метод `apply()`, потому что при помощи метода `apply()` функции можно передать список параметров произвольной длины.

Сейчас мы уже готовы приступить к созданию функции `sendRequest()` для комфортной работы с объектом `XMLHttpRequest`. Итак, мы начинаем.

2.5. Создание простейшей функции-обертки для работы с объектом *XMLHttpRequest*

Пришло время применить наши знания о функциях-объектах и замыканиях на практике. Сейчас мы сделаем первое приближение к разработке функции `sendRequest()`, чтобы поскорее освоить возможности работы с объектом `XMLHttpRequest` в асинхронном режиме. Для начала разработаем функцию `callbackFunction(element)`, которая обработает асинхронный запрос `XMLHttpRequest`, отправленный на сервер. Мы будем передавать этой функции в качестве фактического параметра HTML-элемент, который должен

быть обновлен с учетом данных, полученных с сервера. Кроме того, мы сделаем так, чтобы объект `XMLHttpRequest` был доступен внутри функции `callbackFunction(element)` как неявный параметр `this`, независимо от того, обеспечивает ли Web-браузер такую возможность для `onreadystatechange`-функций.

Мы поставили перед собой достаточно сложные задачи. Напоминаю, что `onreadystatechange`-функции нельзя непосредственно передать фактические параметры, и неявный параметр `this` внутри `onreadystatechange`-функции не во всех Web-браузерах ссылается на объект `XMLHttpRequest`. Как я говорил, один из возможных способов решения поставленной задачи — использовать замыкание.

Для организации замыкания напишем функцию-обертку, основная задача которой — отделить программный код утилитного (вспомогательного) уровня, который обеспечивает передачу формальных параметров и доступ к объекту запроса `XMLHttpRequest` через неявный параметр `this`, от программного кода прикладного уровня. Такую функцию-обертку можно использовать неоднократно для вызова самых различных прикладных функций. Для этого в качестве одного из фактических параметров функции-обертке должна передаваться ссылка на "обворачиваемую" прикладную функцию. Сейчас мы создадим самую простейшую функцию-обертку для работы с объектом `XMLHttpRequest`, а в завершении напишем полноценную функцию `sendRequest()`, которая будет выполнять всю рутинную работу вместо Web-программистов: от создания объекта `XMLHttpRequest` и получения ответа с сервера до обработки ошибок и разрыва циклических связей объекта `XMLHttpRequest`.

Я употребляю в книге перевод термина *wrapper* как *функция-обертка*. В использовании такой функции нет ничего такого, чтобы требовало использовать более загадочный термин. Прежде чем разрабатывать функцию-обертку, рассмотрим прикладную, т. е. обворачиваемую функцию `callbackFunction(element)`. Код этой функции предельно прост:

```
function callbackFunction(element){
  element.innerHTML = this.responseText
}
```

Увы, как я и говорил, использовать эту функцию напрямую в качестве `onreadystatechange`-функции нельзя, поскольку нет возможности передать этой функции фактический параметр `element` и получить доступ к объекту запроса через неявный параметр `this`. Я сказал, что нельзя использовать напрямую. Но, использовать можно (с помощью замыканий)! Для этого и разработаем функцию-обертку с такой сигнатурой:

```
function callbackFunctionWrapper(request, callback, element)
```

Эта функция будет обеспечивать лишь малую часть из необходимых для промышленных приложение возможностей. Она вызовет функцию `callback`, переданную как второй параметр функции, в контексте объекта `request`, переданного как первый параметр функции, с фактическим параметром `element`, который будет предан как третий параметр функции. В программном коде все это выглядит не так страшно, как может показаться.

Для разработки функции используем наш сквозной пример, по загрузке фрагмента HTML-документа, который уже ранее использовался в этой главе. Соответствующий файл расположен в папке \bhvajax\xmlhttprequest и называется testasync.html.

В приведенном варианте функции-обертки мы выполнили поставленную задачу — передали в `onreadystatechange`-функцию фактический параметр и вызвали ее в контексте объекта запроса `XMLHttpRequest`. Как это происходит? В функцию `callbackFunctionWrapper()` в качестве фактических параметров мы передаем ссылку на созданный оператором `new` экземпляр объекта `XMLHttpRequest`, ссылку на callback-функцию `callbackFunction` и ссылку на HTML-элемент, который будет передан в качестве фактического параметра в `callbackFunction()`:

```
xmlHttpRequest.onreadystatechange = callbackFunctionWrapper(
        xmlHttpRequest, callbackFunction, element);
```

В теле функции-обертки `callbackFunctionWrapper()` создается внутренняя анонимная функция, которая возвращается оператором `return` при выходе из функции-обертки:

```
return function(){
  if (request.readyState == 4){
    callback.call(request, element);
    request = null;
  }
};
```

Ссылка на анонимную функцию сохраняется в свойстве `onreadystatechange` объекта `XMLHttpRequest`. Следовательно, локальный контекст функции-обертки замыкается и остается доступным, пока доступен объект `XMLHttpRequest` и пока его свойство `onreadystatechange` будет содержать ссылку на анонимную функцию. При генерации события `onreadystatechange` вызывается анонимная функция, созданная при вызове функции-обертки. В теле анонимной функции анализируется значение свойства `readyState` объекта запроса, и если свойство `readyState` равно `4`, т. е. COMPLETED — будет вызвана `callbackFunction()` в контексте объекта запроса `XMLHttpRequest`

и с фактическим параметром — HTML-элементом, внешний вид которого мы хотим обновить значением, полученным с Web-сервера:

```
if (request.readyState == 4){
  callback.call(request, element);
  request = null;
}
```

Далее, переменной `request` присваивается значение `null` для разрыва циклической ссылки и облегчения работы "сборщика мусора". Такой вариант разрыва циклической ссылки я не встречал в профессиональных библиотеках. Возможно, он работает неэффективно. Поэтому разберем более часто встречающийся вариант разрыва циклической ссылки, который и будем использовать в дальнейшей работе.

Циклическая ссылка объектом запроса `XMLHttpRequest` на `onreadystatechange`-функцию создается в этот момент:

```
var xmlHttpRequest.onreadystatechange = callbackFunctionWrapper(
                xmlHttpRequest, callbackFunction, element);
```

В приведенном фрагменте видно, что свойству `onreadystatechange` объекта запроса присваивается ссылка на анонимную функцию, которую вернет функция-обертка `callbackFunctionWrapper()`. И на этот же самый объект будет ссылаться анонимная функция через формальный параметр `request`. В некоторых типах Web-браузеров циклическая ссылка будет обнаружена и разорвана, после того как объект станет недоступен. Но лучше об этом позаботиться самостоятельно:

```
function nullFunction(){}
function callbackFunctionWrapper(request, callback, element){
  if (typeof callback == "function")
    return function(){
      if (request.readyState == 4) {
        callback.call(request, element);
        request.onreadystatechange = nullFunction;
      }
    };
  else
    return nullFunction;
}
```

В приведенном фрагменте, после обработки ответа Web-сервера, свойству `onreadystatechange` объекта `XMLHttpRequest` присваивается значение пустой функции `nullFunction`. Почему мы не использовали для этой цели значе-

ние null? Дело в том, что в некоторых реализациях объекта XMLHttpRequest, свойству onreadystatechange можно присвоить только значение типа Function. Если мы попробуем присвоить ему значение null — будет сгенерирована ошибка.

Теперь мы уже овладели всем необходимым, чтобы создать функцию sendRequest(), заявленную в начале главы. Осталось добавить всего несколько штрихов — кроссбраузерно создать объект запроса XMLHttpRequest и обработать ошибки, если таковые возникнут при прохождении запроса. Этим мы и займемся в следующем разделе.

2.6. Разработка функции *sendRequest()*

Разрабатываемая в данном разделе функция sendRequest() станет основой для дальнейшей работы с примерами книги. Поэтому поместить эту функцию следует в общую папку, откуда она и будет загружаться приложениями в каждом последующем разделе. В качестве такой папки я выбрал папку \bhvajax\bhv. В этой папке создан файл \bhvajax\bhv\util.js, в который и будет помещена функция sendRequest().

Если вы прямо сейчас откроете файл \bhvajax\bhv\util.js с прилагаемого к книге компакт-диска, то найдете там не только код функции sendRequest(). Например, в \bhvajax\bhv\util.js определены константы с кодами клавиш и реализация функций, определяющих координаты left и top HTML-элемента. В этот файл мы будем помещать код, который почти наверняка будет использоваться в любом нашем приложении. Поэтому файл \bhvajax\bhv\util.js следует всегда подключать первым во всех HTML-документах, рассматриваемых в книге. И делать это необходимо всегда в элементе HEAD HTML-документа. Расположение элемента SCRIPT в теле элемента HEAD гарантирует, что до начала отображения документа в Web-браузере скрипт будет полностью загружен.

В настоящее время широко используются готовые библиотеки JavaScript, которые позволяют быстро и качественно разрабатывать Web-приложения. Иногда используют одновременно несколько библиотек от разных разработчиков. Например, одна библиотека поддерживает кроссбраузерность, другая — работу с асинхронными запросами, третья — работу XML-документами, четвертая — компоненты пользовательского интерфейса. Некоторые библиотеки пишутся с использованием базовых библиотек, поэтому, подключив такую библиотеку, вы должны подключить и соответствующую базовую библиотеку. Теперь задумаемся, как можно быть уверенным, что мы, непродуманно используя имена глобальных переменных, не переопределим переменную или функцию из готовой библиотеки, программный код которой

может быть нам неизвестен во всех подробностях? Такие ошибки могут стать обычным делом, если не используется специально разработанная система именования, о которой мы сейчас поговорим. Чтобы дать вам почувствовать всю серьезность этой проблемы скажу, что некоторые очень хорошие и популярные библиотеки нельзя использовать совместно, потому что они могут переопределять глобальные переменные с тривиальными именами: `Class` или `Method`, уже не говоря о переменных `Ajax` и `Request`. Поэтому мы сразу обеспечим в наших приложениях защиту от проблем подобного рода.

Итак, давайте поговорим об использовании глобальных имен в JavaScript. Пока я разрабатывал небольшие скрипты, я не задумывался об использовании глобальных имен. Когда же я начал разрабатывать свои первые компоненты, то проблемы с засорением глобально пространства имен сразу же дали о себе знать. И я перешел к использованию так называемых *объектов пространства имен*. Чтобы понять, о чем идет речь, разберем небольшой пример. Давайте создадим в глобальном пространстве имен (т. е. вне определения функции) пустой объект и сохраним его значение переменной `bhv`:

```
var bhv = {};
```

Теперь условимся, что `bhv` будет объектом, в свойствах которого мы будем сохранять все необходимые нам глобальные значения:

```
var bhv = {};
bhv.sendRequest = function (){...};
bhv.key = {};
bhv.key.ESC = 27;
```

Объект, который используют для хранения глобальных значений, часто называют объектом пространства имен. Практически все библиотеки JavaScript используют такой объект для хранения глобальных имен. Давайте оценим положительные стороны такого подхода:

- создается всего одна глобальная переменная, в нашем случае `bhv`;
- система имен образует дерево, в котором естественно классифицируются все глобальные объекты, например, все коды клавиш хранятся в объекте `bhv.key`;
- совместное использование двух и более библиотек разных разработчиков не приводит к конфликту имен.

Но есть и отрицательные стороны такого подхода:

- каждая точка в имени объекта — это дополнительная операция доступа к свойству объекта, которая занимает время;
- объект пространства имен сложно инициализировать вручную, т. к. можно случайно переопределить объект повторной инициализацией, или наоборот,

пытаться работать с несуществующими свойствами объекта пространства имен.

Оба эти недостатка легко преодолимы. Чтобы не делать лишней операции доступа, ссылки на часто используемые объекты запоминают в локальных переменных, а для надежного создания объектов используют вспомогательные функции. Об этом мы подробно поговорим в *главе 11*, когда будем рассматривать приемы модульного программирования в JavaScript.

Несмотря на приведенные недостатки, выгоды от хранения глобальных значений в объектах пространства весьма ощутимые. Поэтому их используют практически во всех профессиональных библиотеках JavaScript и Ajax.

Мы тоже будем использовать для хранения глобальных имен такой подход. Например, функцию `sendRequest()` мы сохраним в пространстве имен `bhv`, т. е. как `bhv.sendRequest()`. В то же время, функции, связанные с функцией `bhv.sendRequest()`, но не имеющие самостоятельного значения, будем сохранять в пространстве имен `bhv.util`, например, `bhv.util.nullFunction()`.

Наконец, мы готовы приступить к рассмотрению функции `sendRequest()`. В листинге 2.1 приведен код вспомогательных функций, которые связаны с работой функции `bhv.sendRequest()`. Поскольку этот код носит вспомогательный характер, и не имеет самостоятельного значения, функции `getXMLHttpRequest()`, `nullFunction()`, `defaultError()` и `registreCallbackFunction()` хранятся не в пространстве имен `bhv`, а в более далеком пространстве имен `bhv.util`. Как я уже говорил, такой способ хранения имен позволяет, кроме всего прочего, удобно классифицировать объекты и функции.

Листинг 2.1. Функции, используемые функцией `sendRequest()` в файле util.js

```
var bhv = {util: {}};

bhv.util.getXMLHttpRequest = function(){
  var xmlReq;
  if (window.XMLHttpRequest)
    xmlReq = new window.XMLHttpRequest();
  else
    try {
      xmlReq = new ActiveXObject("Msxml2.XMLHTTP");
    }catch(e) {
```

```
      try {
        xmlReq = new ActiveXObject("Microsoft.XMLHTTP");
      }catch (e) {
        xmlReq = null;
      }
    }
  return xmlReq;
}

bhv.util.nullFunction = function(){};

bhv.util.defaultError = function(){
if (typeof this.responseText != "undefined")
  alert("Ошибка:\n" + this.responseText);
else
  alert("Ошибка: XMLHttpRequest");
}

bhv.util.registreCallbackFunction =
  function(xmlHttpRequest, callback, onerror, callbackArgsArray){
  return function(){
    if(xmlHttpRequest.readyState == 4){
      if(! xmlHttpRequest.status
        || xmlHttpRequest.status >= 200
        && xmlHttpRequest.status < 300
        || xmlHttpRequest.status == 304)
        callback.apply(xmlHttpRequest, callbackArgsArray);
      else
        if (typeof onerror == "function")
          onerror.apply(xmlHttpRequest, callbackArgsArray);
        else
          throw new Error("Ошибка создания XMLHttpRequest")
      xmlHttpRequest.onreadystatechange = bhv.util.emptyFunction;
    };
  }
}
```

Код и назначение функций `getXMLHttpRequest()` и `nullFunction()` должен быть вам уже ясен из *разд. 2.5*. Первая из них кроссбраузерно создает новый объект `XMLHttpRequest` и возвращает ссылку на этот объект. Вторая создает пустую функцию, используемую для разрыва циклической ссылки. Функция `defaultError()` — новое, что появилось в листинге 2.1. При отправке запроса к Web-серверу могут произойти самые различные сбои, начиная от момента отправки запроса на сервер, обработки запроса Web-сервером и заканчивая возвратом результата работы в вызывающий скрипт. Например, адрес или список http-параметров может быть недопустимым, может нарушаться защита при обращении к ресурсу с чужого домена, запрошенный ресурс может отсутствовать, может истечь тайм-аут запроса. Ошибки необходимо обрабатывать всегда. Вы, скорее всего, будете разрабатывать свои обработчики ошибок, в зависимости от конкретных особенностей запроса. В случае если обработчик ошибок не будет явно назначен, будет вызвана функция `defaultError()`, задаваемая в качестве обработчика ошибок запросов по умолчанию. Функция `defaultError()` вызывает окно сообщений `window.alert()`, в котором выводит ответ сервера, если ответ будет доступен, или предупреждающее сообщение. Такой обработчик заведомо не годится для реальных приложений, но при изучении Ajax-технологии он поможет выявить ошибки в запросах. Эти запросы просто сами напомнят вам о себе. (Кстати, немного назойливое поведение этой функции натолкнет вас на мысль о написании собственных обработчиков ошибок запросов.)

Теперь разберем функцию `registreCallbackFunction()`. Эта функция представляет более универсальный вариант функции `callbackFunctionWrapper()` (*см. разд. 2.5*, файл \bhvajax\xmlhttprequest\testasync.html). Сравним эти функции и рассмотрим, что пришлось доработать. Для удобства сравнения приведу код функции:

```
function callbackFunctionWrapper(request, callback, element){
  if (typeof callback == "function")
    return function(){
      if (request.readyState == 4) {
        callback.call(request, element);
        request.onreadystatechange = nullFunction;
      }
    };
  else
    return nullFunction;
}
```

Очевидно, что функция `registreCallbackFunction()` повторяет основные моменты, реализованные в `callbackFunctionWrapper()`, но на более подробном уровне. Начнем с того, что в дополнение к основной прикладной функции в `registreCallbackFunction()` будет передана `onerror`-функция, которая будет вызвана для ошибочного запроса. Вместо одного скалярного параметра будет передан массив типа `Array` параметров, который позволит вызывать прикладную функцию уже не через метод `call()`, а через метод `apply()` с произвольным количеством параметров:

```
callback.apply(xmlHttpRequest, callbackArgsArray);
```

В условии `if` при вызове `callback`-функции кроме состояния запроса `readyState` анализируется и http-статус запроса из свойства `status`:

```
if(xmlHttpRequest.readyState == 4){
  if(! xmlHttpRequest.status
    || xmlHttpRequest.status >= 200
    && xmlHttpRequest.status < 300
    || xmlHttpRequest.status == 304)
    callback.apply(xmlHttpRequest, callbackArgsArray);
  else
    if (typeof onerror == "function")
      onerror.apply(xmlHttpRequest, callbackArgsArray);
```

Анализ свойства `status` объекта `XMLHttprequest` позволяет определить, был ли запрос успешным. Статус успешных запросов лежит в диапазоне от 200 до 299. Принято считать успешным и запрос со статусом 304, который говорит о том, что получен ответ из кэша.

Первое условие:

```
! xmlHttpRequest.status
```

касается запросов не к Web-серверу, а к локальной файловой системе, которые не имеют http-статуса по той простой причине, что не являются http-запросами.

В случае успешного запроса вызывается основная `callback`-функция, а в случае ошибочного — `onerror`-функция.

Теперь можно приступить к написанию функции `sendRequest()` (листинг 2.2). Основное назначение этой функции — спрятать подробности создания и вызова методов объекта `XMLHttpRequest` и задание свойств по умолчанию этого запроса. В табл. 2.1 приведено описание формальных параметров функции `sendRequest()`.

Таблица 2.1. *Формальные параметры функции* `sendRequest()`

Имя формального параметра	Краткое описание формального параметра	Тип значения формального параметра	Значение, задаваемое по умолчанию
`httpMethod`	Метод запроса к Web-сервера, как правило, `GET` или `POST`	`String`	`POST`
`url`	URL-адрес запроса	`String`	Не определено
`httpParams`	Параметры запроса или тело запроса	`String`	Не определено
`async`	Асинхронность запроса	`Boolean`	`False`
`callback`	Основная функция-обработчик запроса	`Function`	Не определено
`onerror`	Функция-обработчик ошибочного запроса	`Function`	`bhv.uti.defaultError()`
`callbackArgsArray`	Массив фактических параметров функции-обработчика запроса	`Array`	Пустой массив `[]`
`contentType`	Тип запроса	`String`	`application/x-www-form-urlencoded`
`headers`	Заголовки запроса	`Object`	Объект `{}`

В первых строках программы производится анализ значений фактических параметров, переданных в функцию `sendRequest()`, которым в некоторых случаях присваиваются значения по умолчанию. Хочу лишний раз подчеркнуть, что значения по умолчанию присваиваем только мы в своей функции `sendRequest()`, чтобы вы случайно не подумали, что эти значения по умолчанию предусмотрены в объекте `XMLHttprequest`.

Листинг 2.2. Функция `sendRequest()` в файле bhvajax\bhv\util.js

```
bhv.sendRequest = function(httpMethod, url, httpParams, async,
                           callback, onerror, callbackArgsArray,
                           contentType, headers) {

if (! onerror)
```

```
  onerror = bhv.util.defaultError;

if (! callbackArgsArray)
  callbackArgsArray = [];

if (! contentType)
  contentType = "application/x-www-form-urlencoded";

if (! headers)
  headers = {};

var xmlHttpRequest = bhv.util.getXMLHttpRequest();

if (async)
  xmlHttpRequest.onreadystatechange = registreCallbackFunction(
          xmlHttpRequest, callback, onerror, callbackArgsArray);

try {
  if (httpMethod.toLowerCase() == "get") {
    xmlHttpRequest.open("get", url + "?" + httpParams, async);
    xmlHttpRequest.setRequestHeader("Content-Type", contentType);
    for (var header in headers)
      xmlHttpRequest.setRequestHeader(header, headers[header]);
    xmlHttpRequest.send(null);
  } else {
  xmlHttpRequest.open("post", url, async);
  xmlHttpRequest.setRequestHeader("Content-Type", contentType);
  for (var header in headers)
    xmlHttpRequest.setRequestHeader(header, headers[header]);
    xmlHttpRequest.send(httpParams);
  }
} catch(e) {
  xmlHttpRequest.onreadystatechange = bhv.util.emptyFunction;
  if (typeof onerror == "function")
    onerror.apply(xmlHttpRequest, callbackArgsArray);
  else
    throw new Error("Ошибка XMLHttpRequest");
}

if (! async)
  if (! xmlHttpRequest.status
```

```
        || xmlHttpRequest.status >= 200
        && xmlHttpRequest.status < 300
        || xmlHttpRequest.status == 304)
    callback.apply(xmlHttpRequest, callbackArgsArray);
  else
    if (typeof onerror == "function")
      onerror.apply(xmlHttpRequest, callbackArgsArray);
    else
      throw new Error("Ошибка XMLHttpRequest");
}
```

После анализа параметров для асинхронных (и только для асинхронных) запросов назначается `onreadystatechange`-функция:

```
if (async)
  xmlHttpRequest.onreadystatechange = registreCallbackFunction(
          xmlHttpRequest, callback, onerror, callbackArgsArray);
```

Затем, для всех — и для асинхронных и для синхронных запросов — вызываются методы `open()`, `setRequestHeader()` и `send()` с соответствующими параметрами. Обратите внимание на разницу задания параметров при использовании http-методов `GET` и `POST`. Для http-метода `GET` http-параметры задаются в методе `open()`:

```
xmlHttpRequest.open("get", url + "?" + httpParams, async);
xmlHttpRequest.send(null);
```

Для http-метода `POST` http-параметры задаются в методе `send()`:

```
xmlHttpRequest.open("post", url, async);
xmlHttpRequest.send(httpParams);
```

Код отправки запроса заключен в блок `try-catch`, и в ветви `catch` происходит разрыв циклической ссылки и вызов `onerror`-функции. Дело в том, что ошибка может возникнуть еще до отправки запроса на сервер, и обработку ошибок такого запроса мы можем произвести только здесь, в блоке `try-catch`:

```
catch(e) {
  xmlHttpRequest.onreadystatechange = bhv.util.emptyFunction;
  if (typeof onerror == "function")
    onerror.apply(xmlHttpRequest, callbackArgsArray);
  else
    throw new Error("Ошибка XMLHttpRequest");
}
```

Наконец, осталось обсудить завершающие строчки кода функции `sendRequest()`, в которых вызывается функция-обработчик для синхронных запросов. Напоминаю, что `onreadystatechange`-функция вызывается автоматически только для асинхронных запросов и только в некоторых браузерах для синхронных запросов. Поэтому у синхронных запросов мы оставили свойство `onreadystatechange` пустым. Нам не остается другого выбора, как явно вызвать функцию-обработчик для синхронных запросов, что мы и делаем:

```
if (! async)
  if (! xmlHttpRequest.status
        || xmlHttpRequest.status >= 200
        && xmlHttpRequest.status < 300
        || xmlHttpRequest.status == 304)
    callback.apply(xmlHttpRequest, callbackArgsArray);
  else
    if (typeof onerror == "function")
      onerror.apply(xmlHttpRequest, callbackArgsArray);
    else
      throw new Error("Ошибка XMLHttpRequest");
```

Я думаю, что с этим кодом вы уже успели не раз познакомиться в тексте главы, так что он не потребует разъяснений, за исключением, может быть, одного условия:

```
if (! async)
```

которое означает не асинхронный запрос, т. е. синхронный запрос.

Заявленная в начале *главы 2* функция `sendRequest()` создана. И на этом можно было бы закончить изложение. Но мне показалось некорректным по отношению к уважаемым читателям, пробивавшимся через порой усложненный материал по основам JavaScript, оставить тех, кто все еще с нами, без реально работающего примера, который можно было бы прямо сегодня применить на практике. Я решил в качестве такого примера предложить модернизировать компонент "Аккордеон" из *главы 1*, применив функцию `sendRequest()` на практике немедленно.

2.7. Компонент "Аккордеон" с асинхронной загрузкой текста панелей

В *главе 1* мы разработали компонент "Аккордеон", который не использует асинхронные запросы к серверу для загрузки текста панелей. Давайте задумаемся, правильно ли это? Компонент "Аккордеон" может состоять из десятка

панелей (я думаю, оптимально, все же, не более семи). На каждой из панелей могут быть довольно объемный текст и изображения. Содержимое всех панелей компонента начинает загружаться сразу и загружается полностью, хотя пользователь может просмотреть один-два раздела и закрыть Web-страницу. Тут и стоит подумать о применении асинхронной загрузки содержимого панелей при помощи объекта `XMLHttpRequest` и только что разработанной нами функции `sendRequest()`.

Принцип работы компонента "Аккордеон" с асинхронной загрузкой содержимого панелей предельно прост. При создании компонента в Web-браузер загружаются только заголовки разделов и содержимое всего одной активной панели компонента "Аккордеон". Все остальные разделы в этот момент "отдыхают" на Web-сервере и ожидают, когда пользователь захочет прочитать их, и выберет соответствующий заголовок, щелкнув по нему кнопкой мыши.

Мы используем программный код, разработанный в *главе 1*, и внесем совсем незначительные изменения, чтобы все работало именно так, как мы хотим. Для начала работы, скопируйте из папки \bhvajax\accordion файлы accordion.js, accordion.css и accordion2.html в папку \bhvajax\xmlhttprequest и переименуйте файл accordion2.html в accordion3.html. Файл accordion.js переименовывать не надо, т. к. он будет поддерживать тот же функционал, что и версия из *главы 1*, к которому будут добавлены новые возможности. Поэтому вы можете для всех HTML-документов старую версию файла accordion.js заменить новой версией, не меняя кода самого HTML-документа.

Напоминаю, что первый вариант компонента создавался при помощи функции `accordionAddBehivior(accordion)`, которой в качестве параметра передавался в виде строки `String` идентификатор HTML-элемента `DIV`, в котором необходимо расположить компонент. Текст заголовков разделов и текст содержимого панелей компонента "Аккордеон" содержался прямо в тексте документа в HTML-элементах `DIV`. Для загрузки асинхронными запросами такой вариант разметки не подойдет. Поэтому в тексте HTML-документа останется всего один пустой элемент для всего компонента:

```
<div id="accordion1"> </div>
```

А всю информацию о заголовках разделов и содержимом панелей мы передадим функции `accordionAddBehivior()` как параметр в виде массива `Array`.

Сделаем еще одно небольшое отступление и поговорим об особенностях работы с типом `Array`, иначе дальнейший программный код может быть вам неясен.

Массив `Array` может быть создан двумя практически эквивалентными способами:

```
var arr1= Array();
var arr2 = [];
```

Можно сразу же создать массив, с инициализацией элементов:

```
var arr3 = Array(value1, value2, value3);
var arr4 = [value1, value2, value3];
```

Из двух приведенных нотаций сейчас принято пользоваться второй (с квадратными скобками). Но есть один случай, когда два способа не эквивалентны:

```
var arr5 = Array(N);
var arr6 = [N];
```

В этом случае `arr5` будет ссылаться на массив из `N` пустых элементов, а `arr6` на массив из одного элемента, равного `N`.

Количество элементов в массиве можно узнать из свойства `length`.

Массивы могут быть вложенными и нерегулярными (иметь разное количество элементов во вложенных массивах):

```
var arr7 = [
  [1, "Два", 3],
  [4, 5],
  [6, 7, 8, "девять"]
]
```

Индексация элементов массивов начинается с нуля. Обращаются к элементу массива так:

```
var arr8 = arr7[1];     // [4, 5]
var var9 = arr7[1][1];  // 5
```

Оператор `typeof` для массива возвращает строковое значение `"object"`, а операторы `instanceof` — логическое значение `true` и при сравнении с `Object`, и при сравнении с `Array`.

Для перебора элементов массива в цикле нельзя использовать цикл `for in`:

```
for (var ind10 in arr10)
  alert(arr10[ind10]);
```

Вы можете найти рекомендации поступать так в литературе, но попробуйте проделать эксперимент — протестируйте следующий код:

```
Array.prototype.toJSON= function() {
  var json="[";
  for (var i = 0; i<this.length-1; i++)
    json += (this[i] + ", ");
  json += this[i] + "]";
  return json;
}
```

```
var arr11=[5,6,7];
for (var ind11 in arr11)
  alert(arr11[ind11]);
alert(arr11.toJSON());
```

Что произошло? Вместо трех элементов цикл в дополнение вывел еще и четвертый элемент — функцию-метод `toJSON()`. Можно спорить о том, корректно ли подгружать свои методы к стандартным объектам: `Object`, `Array`, `String`, но правда заключается в том, что многие популярные библиотеки подгружают такие методы, и цикл `for in` в таких случаях будет работать неверно. Поэтому массивы `Array` всегда следует обходить в простом цикле `for` (не `for in`).

Мы рассмотрели основы работы с объектом `Array` и можем вернуться к разработке компонента "Аккордеон". Как мы и условились, всю информацию о заголовках разделов и содержимом панелей мы передадим функции `accordionAddBehivior()` в параметре в виде массива `Array`. Он будет иметь такой вид:

```
var accordionModel = [
  ["Первый раздел", "../text/text1.html"],
  ["Второй раздел", "../text/text2.html"],
  ["Третий раздел", "../text/text3.html"]
];
```

Массив `accordionModel` состоит из трех элементов типа `Array` по числу разделов компонента. То есть в нашем компоненте будут три раздела с заголовками: "Первый раздел", "Второй раздел" и "Третий раздел". Разумеется, разделов может быть и больше.

Каждый из элементов массива `accordionModel` представляет собой массив `Array` из двух элементов. В этом случае, количество элементов будет всегда жестко фиксированным и равно двум. Первый элемент (с индексом 0) содержит строку с заголовком раздела, а второй элемент (с индексом 1) содержит строку типа `String` с URL-адресом файла или Web-ресурса, который будет загружаться в панель содержимого раздела компонента "Аккордеон". В примере этот адрес задан относительно файла \bhvajax\xmlhttprequest\accordion3.html, т. е. файлы text1.html, trxt2.html, text3.html должны находиться в папке \bhvajax\test. Подробнее о задании адресов загружаемых средствами Ajax Web-ресурсов мы поговорим в *главе 3*.

Если вы теперь откроете код наших с вами компонентов, разработанных в *главе 1*, то сразу оцените, насколько проще стал код HTML-документа. Обратите внимание, что я добавил не только второй, но и третий параметр к функции `accordionAddBehivior()`. Этот параметр указывает, какой раздел

необходимо активизировать после создания компонента. Возможно, не всегда это будет первый раздел. Теперь осталось реализовать, вернее, дополнить код компонента в файле \bhvajax\xmlhttprequest\accordion.js.

Если функция вызвана со вторым формальным параметром `model` типа `Array`, этот параметр анализируется и на его основе формируется объектная модель компонента. Эта объектная модель полностью соответствует той модели, которую мы создавали вручную для компонента "Аккордеон" в *главе 1*, только создается при помощи методов `createElement()` и `createTextNode()` предопределенного объекта `document` Web-браузера.

Для каждого раздела компонента создается элемент `DIV` заголовка раздела:

```
var div = document.createElement("DIV");
```

В этот элемент помещается текст заголовка:

```
div.appendChild(document.createTextNode(model[i][0]));
```

Инициализируются два новых свойства элемента `DIV`:

```
div.contentURL = model[i][1];
div.isLoaded = false;
```

Первое свойство задает URL-адрес ресурса, загружаемого в панель раздела, а второе свидетельствует о том, что ресурс еще не загружен с Web-сервера. Наконец, элемент заголовка раздела присоединяется к компоненту "Аккордеон", и за ним присоединяется пустой элемент `DIV` для панели содержимого раздела:

```
accordion.appendChild(div);
div = document.createElement("DIV");
accordion.appendChild(div);
```

Начиная с этого момента, объектная модель компонента, созданного программно, ничем не отличается от объектной модели из *главы 1*, которую мы создавали вручную при помощи HTML-разметки. Поэтому дальнейший код компонента не изменился, за исключением кода загрузки активного раздела компонента:

```
if (i == activate*2){
  div[i].className = "accordionCurrentTitle";
  div[i+1].className = "accordionCurrentPane";
  if (model instanceof Array)
    div[i].onclick();
}else {
  div[i].className = "accordionOtherTitle";
  div[i+1].className = "accordionOtherPane";
}
```

Во-первых, мы проверяем, является ли текущий раздел тем разделом, который будет активизирован после создания компонента (это определяется третьим параметром функции `accordionAddBehivior()`):

```
if (i == activate*2)
```

И затем для программно созданных компонентов, для которых выполняется условие:

```
if (model instanceof Array)
```

программно вызываем обработчик щелчка мышью:

```
div[i].onclick();
```

Вот и все изменения. Естественно, что программный код обработчика события от мыши тоже должен быть изменен, но только для первого выбора раздела. Если раздел активизируется второй раз — он уже инициализирован и не требует повторной загрузки (если, разумеется, речь идет о статическом Web-ресурсе, содержимое которого не изменяется). Приведу только программный код, добавленный в функцию `accordionTitleOnClick()`:

```
if (this.contentURL && !this.isLoaded){
  this.isLoaded = true;
  bhv.sendRequest("get",this.contentURL,null,
    true,function(pane){pane.innerHTML = this.responseText},
    null,[nextPane]);
}
```

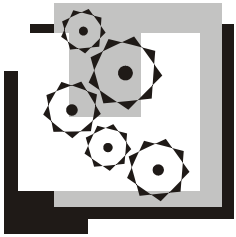
Для обработчика события `onclick` неявный параметр `this` ссылается на HTML-элемент, к которому привязан обработчик, в данном случае это будет заголовок раздела. Далее, для заголовков раздела, у которых есть свойство `contentURL`, и которые еще не были загружены (`!this.isLoaded`), вызывается функция `sendRequest()` со следующими фактическими параметрами:

- `"get"` — запрос http-методом `GET`;
- `this.contentURL` — URL-адрес загружаемого ресурса;
- `null` — http-параметры запросу не передаются;
- `true` — запрос асинхронный;
- `function...` — код функции-обработчика успешного запроса;
- `null` — функция-обработчик ошибочного запроса будет назначена по умолчанию;
- `[nextPane]` — массив из одного элемента `DIV`, в который будет помещен ответ сервера.

* * *

Итак, мы получили работающий компонент. Давайте проанализируем, все ли было сделано правильно, и наметим пути его совершенствования.

В данном случае, компонент был реализован путем внесения изменений в компонент "Аккордеон" из *главы 1*. Я пытался сделать это с минимальными усилиями, поэтому программный код не всегда получился обоснованным. Во-первых, не реализован шаблон проектирования "Модель — Вид — Контроллер". Во-вторых, функция-обработчик запроса неоправданно создается как анонимная функция при каждом вызове функции `sendRequest()`. В-третьих, в тексте загружаемых в панели компонента "Аккордеон" файлов могут находиться ссылки на Web-ресурсы, например, на изображения, таблицы стилей, скрипты, которые не будут загружаться, если они заданы относительным URL-адресом. В *главе 3* мы создадим компонент "Панель с закладками", в котором постараемся дать более красивое и универсальное решение. А пока можете попробовать самостоятельно усовершенствовать код компонента в качестве домашнего задания.



Глава 3

Разработка компонента "Панель с закладками"

"Панель с закладками" представляет собой набор панелей, из которых одна панель является активной, а остальные скрыты от пользователя. В верхней части панели размещаются закладки. Закладки служат примерно для того, что и обычные книжные закладки. Выбирая закладку, пользователь выбирает следующую активную панель. По функциональности "Панель с закладками" немного напоминает компонент "Аккордеон". Но мы не будем повторяться. Как вы помните, в *главе 1* мы разработали компонент "Аккордеон", который не работает с асинхронными запросами. В *главе 2* функциональность компонента "Аккордеон" была расширена. Была добавлена возможность асинхронно загружать содержимое панелей с Web-сервера по требованию пользователя. Но некоторые вопросы остались нерешенными. Например, при загрузке HTML-документа в панель компонента "Аккордеон", мы загружаем текст документа. В тексте могут содержаться HTML-элементы `IMG`, у которых URL-адреса в атрибуте `src` заданы в относительной форме. Будут ли правильно загружаться изображения, зависит от того, в какой папке расположен загружаемый асинхронным запросом документ. Если загружаемый документ расположен в одной папке с текущим документом, то изображения будут загружаться правильно. Если загружаемый документ расположен в любой другой папке, изображения, заданные относительными адресами, не будут загружаться. Почему так происходит, что такое абсолютные и относительные URL-адреса, абсолютные и относительные пути, вы узнаете, прочитав эту главу. Кроме того, вы познакомитесь с возможностью создавать рисованные компоненты при помощи изображений и свойств стиля `background`, `background-image`, `background-repeat`, `background-position`.

Код компонента "Панель с закладками" вы можете найти в папке \bhvajax\tabbedpane на прилагаемом к книге компакт-диске.

3.1. Реализация интерфейса компонента "Панель с закладками"

Стандартные HTML-элементы имеют одну особенность, которая ограничивает возможности Web-дизайнеров. Все HTML-элементы реализованы в виде прямоугольников. Но, как гласит пословица, нет худа без добра. Если вы немного поработаете со стилями своих компонентов, в частности со свойством стиля `background-image`, то сможете создавать рисованные компоненты самой различной формы, и ваша Web-страница получит больше шансов стать запоминающейся. Когда я впервые увидел Web-страницу с рисованными компонентами, моя первая мысль была — как же это сделано? А когда я понял, что передо мной всего лишь комбинация элементов с фоновыми изображениями, то поразился, насколько простыми средствами можно достичь эффектного результата. Конечно, использование фоновых изображений и рисованных компонентов — это еще не залог хорошего Web-дизайна. Речь идет о том, что возможности Web-дизайнера существенно расширяются. Мы же, Web-программисты, должны знать, как обеспечить необходимый Web-дизайнеру функционал. Разрабатываемый нами компонент с художественной точки зрения не будет совершенен. Мы больше сосредоточимся на программном решении вопроса. Для своего реального приложения вы, поработав над файлами с определениями стилей, сделаете так, что ваш компонент приобретет не только необходимый функционал, но и хорошие эстетические свойства. Для этого, разумеется, необходимо изначально договориться, что весь код, связанный со стилевым оформлением документа, будет сосредоточен в css-файлах.

Использование JavaScript открывает новые возможности для разработки рисованных компонентов. Один рисованный компонент может быть реализован двумя, тремя и большим количеством HTML-элементов `DIV` и `SPAN`. Но это еще не все. Рисованный компонент должен реагировать изменением своего внешнего вида на обычные для HTML-элементов события — наведение мыши, щелчок кнопкой мыши. Реализовать такое поведение рисованного компонента без помощи JavaScript довольно сложно. Вы можете вспомнить наш недавний опыт с компонентом "Аккордеон", который при помощи HTML-разметки создавался с большим объемом ручной работы, а в окончательном варианте — двумя строчками кода JavaScript.

Посмотрим, как будет выглядеть компонент "Панель с закладками" в окне Web-браузера. На рис. 3.1 одна панель этого компонента активна, а все остальные, с точки зрения пользователя, скрыты за активной панелью и представлены только своими закладками. Внешний вид закладки активной панели отличается от внешнего вида закладок неактивных панелей. Активная

закладка как бы образует единое целое с активной панелью. Пожалуй, наиболее сложно при разработке компонента сделать так, чтобы пользователь интуитивно ассоциировал активную панель с активной закладкой, а неактивные закладки — с возможностью выбрать новую активную панель.

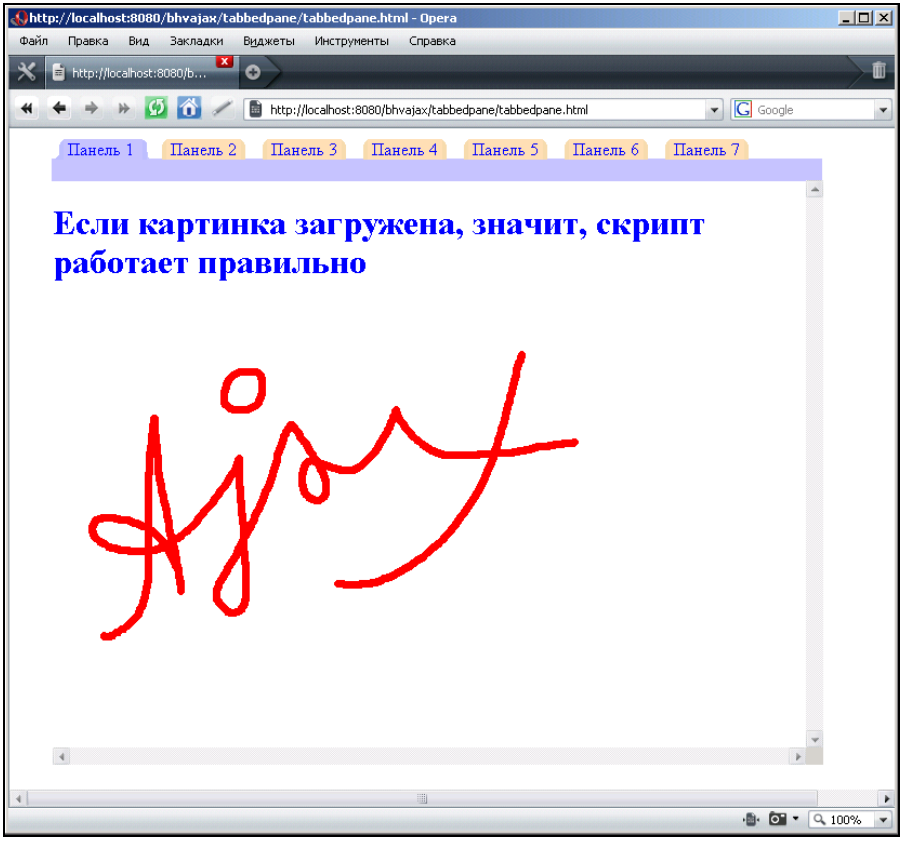


Рис. 3.1. Внешний вид компонента "Панель с закладками"

Вы можете сами теперь подумать о том, как сделать компонент "Панель с закладками" интуитивно понятным для пользователя. В нашем компоненте интуитивность реализована следующими приемами. Закладка сливается по цвету фона с верхней границей рамки активной панели. Для того чтобы еще усилить эффект слияния, у активной закладки переход в верхнюю границу рамки активной панели происходит с небольшим закруглением, а у неактивных закладок — под прямым углом. Кроме этого, активная закладка увеличивается на два пиксела по сравнению с неактивной, что создает иллюзию едва уловимого приближения закладки при выборе.

На рис. 3.2 представлена схема HTML-элемента, по которой создана активная закладка. Активная закладка состоит из пяти HTML-элементов SPAN. Для большей наглядности представим эти элементы в виде фрагмента HTML-документа:

```
<span class="tabLeftBefor">    </span>
<span class="tabLeft">          </span>
<span class="tabCenter">        </span>
<span class="tabRight">         </span>
<span class="tabRightAfter">    </span>
```

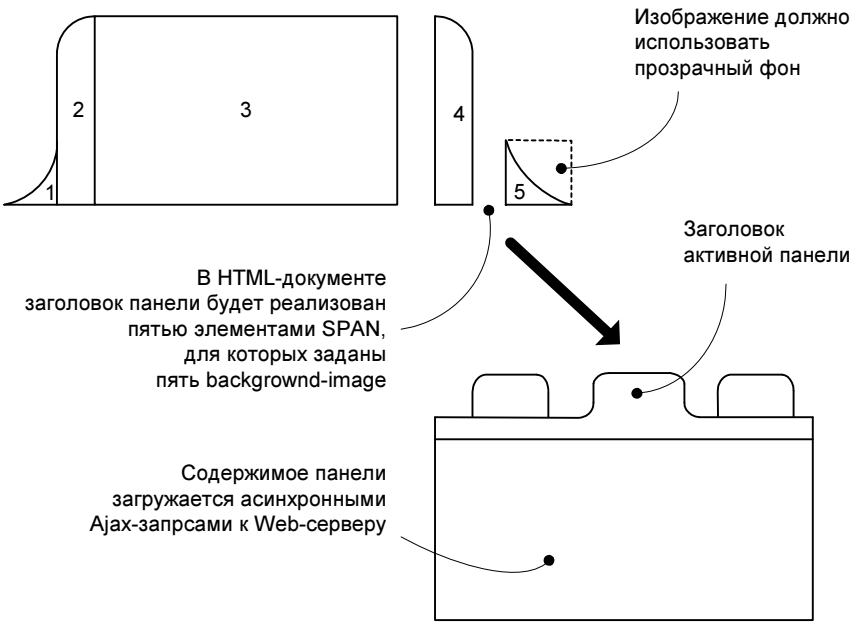


Рис. 3.2. Схема активной закладки компонента "Панель с закладками"

Разумеется, при создании компонента "Панель с закладками" мы не будем кодировать такой фрагмент HTML-документа вручную. Создание HTML-элементов будет осуществляться методами объектной модели документа (DOM) HTML-документа и JavaScript.

Для того чтобы превратить выделенную закладку в невыделенную и наоборот, можно менять при помощи JavaScript стилевые классы у всех пяти элементов SPAN. Но существует и более простой путь. В CSS можно определять

контекстно-зависимые стили. Поместим наши пять элементов SPAN в один охватывающий элемент SPAN:

```
<span class="tabSelected">
  <span class="tabLeftBefor">    </span>
  <span class="tabLeft">         </span>
  <span class="tabCenter">       </span>
  <span class="tabRight">        </span>
  <span class="tabRightAfter">   </span>
</span>
```

Наши пять элементов находятся в теле элемента SPAN, с именем стилевого класса tadSelected. Теперь, для того чтобы изменить внешний вид этих элементов, достаточно поменять имя класса только у охватывающего элемента SPAN:

```
<span class="tab">
  <span class="tabLeftBefor">    </span>
  <span class="tabLeft">         </span>
  <span class="tabCenter">       </span>
  <span class="tabRight">        </span>
  <span class="tabRightAfter">   </span>
</span>
```

Рассмотрим, как будут заданы стилевые правила для компонента "Панель с закладками" в файле tabbedpane.css.

Контекстно-зависимые стили задаются следующим образом:

```
span.tab span.tabLeft {
    margin: 0px 0px 0px 0px;
    padding: 0px 0px 0px 0px;
    border: none;
    text-align: center;
    overflow: visible;
    background-image: url("tableft.gif");
    background-repeat: no-repeat;
    background-position: left top
}

span.tabSelected span.tabLeft {
    margin: 0px 0px 0px 0px;
    padding: 0px 0px 0px 0px;
```

```
    border: none;
    text-align: center;
    overflow: hidden;
    background-image: url("tabselectleft.gif");
    background-repeat: no-repeat;
    background-position: left top;
    width: 5px
}
```

Что означает такая запись? Когда элемент `SPAN` с заданным стилевым классом `tabLeft` находится в теле элемента `SPAN` с заданным стилевым классом `tab`, для его отображения в Web-браузере используется фоновый рисунок tableft.gif. В файле CSS с определениями стилей ссылка на файл изображения задается как `url("tableft.gif")`. Если программно изменить имя стилевого класса охватывающего компонента с `tab` на `tabSelected`, начнет действовать другое стилевое правило, и в качестве фонового имиджа будет использоваться tabselectleft.gif.

Кроме свойства стиля `background-image`, важными при создании рисованных компонентов являются свойства:

- `background-position`, которое может принимать значения `left`, `right`, `top`, `bottom` и `center`. Это свойство может быть задано раздельно для `background-x` и `background-y` в абсолютных величинах или процентах;
- `background-repeat`, которое может принимать значения `repeat` (значение по умолчанию), `no-repeat`, `repeat-x`, `repeat-y`;
- `background-color`, которое определяет цвет фона.

Как всегда, в CSS существует возможность задать все эти свойства стиля в сокращенной форме в одном свойстве `background`.

Свойство `background-color` обычно подбирают в тон фонового изображения, на случай если оно будет долго загружаться или пользователь отключит загрузку изображений.

Свойство `background-repeat` задает многократное повторение изображения по вертикали или по горизонтали. Использование многократного повторения позволяет, во-первых, уменьшить сетевой трафик и ускорить загрузку документа и, во-вторых, обеспечивает гибкость при создании рисованных компонентов. Например, для центрального элемента используется повторение по горизонтали фонового изображения:

```
span.tabSelected span.tabCenter {
    margin: 0px 0px 0px 0px;
```

```
    padding: 0px 8px 0px 4px;
    border: none;
    text-align: center;
    overflow: visible;
    background-image: url("tabselectcenter.gif");
    background-repeat: repeat-x;
    background-position: left top
}
```

Многократное повторение позволяет использовать изображение шириной в один пиксел для прорисовки элемента произвольной ширины.

Выравнивание элемента при помощи свойства `background-position` по правой, левой, верхней или нижней границе HTML-элемента активно используется при создании рисованных компонентов. Как правило, размер компонента заранее неизвестен при создании Web-страницы, использующей компонент, и даже при загрузке страницы в Web-браузер с учетом разрешения монитора. Поэтому фоновое изображение либо задается повторением `repeat`, либо делается с запасом, вроде припуска при кройке и шитье. Лишняя часть фонового изображения не выводится Web-браузером, но некоторые части изображения, в нашем случае это закругления, должны быть отображены и выровнены по границе элемента:

```
span.tabSelected span.tabLeftBefor {
    margin: 0px 0px 0px 0px;
    padding: 0px 0px 0px 0px;
    border: none;
    text-align: center;
    overflow: hidden;
    background-image: url("tabselectleftbefor.gif");
    background-repeat: no-repeat;
    background-position: left bottom;
    width: 5px
}

span.tabSelected span.tabCenter {
    margin: 0px 0px 0px 0px;
    padding: 0px 8px 0px 4px;
    border: none;
    text-align: center;
    overflow: visible;
    background-image: url("tabselectcenter.gif");
```

```
    background-repeat: repeat-x;
    background-position: left top
}
```

Для первого элемента SPAN задано выравнивание по левой нижней границе, а для второго — по левой верхней границе. Именно такое задание background-position позволяет правильно нарисовать компонент *"Выделенная закладка"*, чтобы формообразующие части изображения всегда были на своем месте.

Мы углубились в рассмотрение стилевого оформления закладок, т. к. этот материал новый для нас. Теперь вернемся к файлу tabbedpane.css, в котором заданы стилевые правила для всех элементов компонента "Панель с закладками". На рис. 3.3 изображена схема, в которой видно, как будут располагаться панели компонента и какие стилевые классы будут применяться HTML-элементам, при помощи которых реализован компонент. Переведем эту схему на язык HTML-документа:

```
<div class="tabbed">
  <div class="tabbedBar">
    <span class="tabSelected"></span>
    <span class="tab"></span>
    <span class="tab"></span>
    <span class="tab"></span>
  </div>
  <div class="tabbedSpace">
  </div>
  <div style="tabbedPane">
    <div style="paneSelected">
       Панель 1 (активная)
    </div>
    <div style="pane">
       Панель 2
    </div>
    <div style="pane">
       Панель 3
    </div>
    <div style="pane">
       Панель 4
    </div>
  </div>
</div>
```

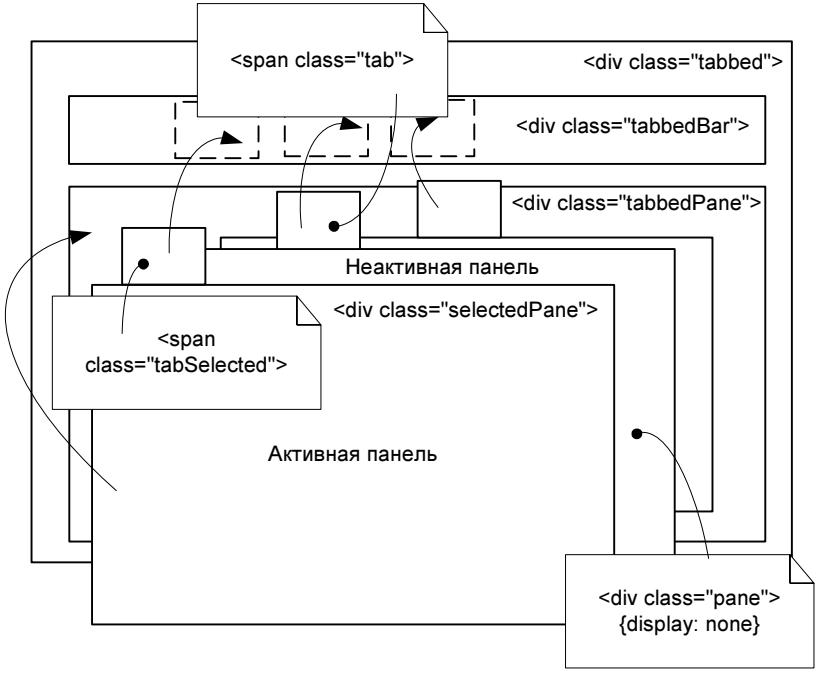


Рис. 3.3. Схема компонента "Панель с закладками"

Этот фрагмент HTML-документа мы рассмотрели только для наглядности. Как и компонент "Аккордеон", компонент "Панель с закладками" будет создаваться программно. Для этого, в конце концов, мы и решили изучать Ajax-технологию.

3.2. Разработка JavaScript-кода компонента "Панель с закладками"

В данном разделе мы напишем код JavaScript, который шаг за шагом создаст компонент "Панель с закладками". Как и в *главе 2*, создадим массив типа `Array`, содержащий текст закладок и URL-адреса, с которых будет загружаться содержимое панелей:

```
var tabbedpaneModel = [
  ["Закладка 1", "../text/text1.html"],
  ["Закладка 2", "../text/text2.html"],
  ["Закладка 3", "../text/text3.html"]
];
```

Код создания компонента "Панель с закладками" будет очень похож на код создания компонента "Аккордеон" с асинхронной загрузкой панелей из *главы 2*. Пусть вас не смущает такое повторение. Далее в этой главе мы поговорим и о более интересных вещах. Но теперь необходимо проделать работу по созданию компонента (см. файл tabbedpane.js).

В этом JavaScript-коде почти все вам должно быть уже знакомым. Первые строки кода проверяют наличие объекта, в котором мы договорились запоминать глобальные имена — `bhv`, и в случае отсутствия такого объекта создают его:

```
if (typeof bhv == "undefined")
  bhv = {};
if (typeof bhv.tabbedpane == "undefined")
  bhv.tabbedpane = {};
```

Все объекты, связанные с компонентом "Панель с закладками", будут сохраняться в объекте `bhv.tabbedpane`. Вас может заинтересовать такой вопрос. Переменная `bhv` инициализируется в файле util.js, который мы договорились всегда загружать первым в `HEAD` HTML-документа. Не делаем ли мы лишнюю работу, проверяя наличие объекта `bhv` в памяти? Оказывается, что нет. Дело в том, что скрипты могут загружаться в Web-браузер совсем не в том порядке, в котором они были указаны в HTML-документе. Скрипт, который был указан раньше в тексте HTML-документа, будет раньше запрошен на Web-сервере. Но момент, в который Web-сервер закончит передачу скрипта и Web-браузер будет готов к его выполнению, зависит от размера скрипта, настроек кэширования и других, практически, случайных, факторов, а не от порядка, в котором HTML-элементы `SCRIPT` указаны в HTML-документе. Этот момент может стать неожиданностью для разработчика и привести к неправильной работе приложения, особенно если приложение тестируется и отлаживается в локальной сети, а затем перемещается в Интернет. За счет разной скорости соединения с Web-сервером в Интернете и в локальной сети, порядок загрузки скриптов может измениться. Поэтому проверять наличие объекта пространства имен необходимо в каждом скрипте. В противном случае вы можете или обратиться к несуществующему объекту, или повторно создать пустой объект и утратить при этом часть свойств и методов, определенных в существующем объекте.

В компоненте "Панель с закладками" используется реализация отложенной загрузки содержимого панелей. На самом деле, некоторые панели могут быть и не востребованы пользователем. Поэтому загрузка панелей происходит только при выборе соответствующей закладки:

```
bhv.tabbedpane.selectTab = function(){
  if (! this.loaded) {
```

```
    this.loaded = true;
    bhv.sendRequest("GET", this.address, null, true,
             bhv.tabbedpane.loadTab, bhv.tabbedpane.errorTab, [this]);
  }
```

Загрузка происходит только при первом выборе закладки, для чего анализируется свойство `this.loaded`. При первом же обращении к панели этому свойству присваивается значение `this.loaded = true`. В данном случае `this` указывает на HTML-элемент `SPAN`, в котором содержится закладка.

Существуют, по крайней мере, две возможности хранить информацию о модели компонента на стороне клиента (Web-браузера) — в пользовательском объекте JavaScript или прямо в свойствах HTML-элементов.

В объекте JavaScript типа `Array` мы задали информацию, используемую при создании объекта:

```
var tabbedpaneModel = [
  ["Закладка 1","../text/text1.html"],
  ["Закладка 2","../text/text2.html"],
  ["Закладка 3","../text/text3.html"]
];
```

После инициализации компонента этот объект не используется. Можно было бы пойти путем развития функциональности модели данных, заданной в объекте JavaScript, и привязывать к нему обработчики событий и свойства, такие как `loaded`. Так мы и будем поступать при создании более сложных компонентов — Lookup Combobox и "Таблица данных". Сейчас же мы привяжем свойства прямо к HTML-элементу `SPAN`:

```
div = document.createElement("SPAN");
div.address = paneModel[i][1];
div.loaded = false;
div.onclick = bhv.tabbedpane.selectTab;
var pane = document.createElement("DIV");
pane.className = "pane";
div.pane = pane;
```

Недостатки такого подхода понятны. Модель данных смешивается с представлением данных. Но сейчас мы разрабатываем достаточно простой компонент. Его модель должна содержать информацию о URL-адресе загружаемого в панель HTML-документа `address`, о том, была ли страница уже загружена `loaded`, и ссылку на HTML-элемент, в котором отображается содержимое панели `pane`.

Мы сказали, что у подхода с хранением модели компонента в свойствах HTML-элемента есть недостатки. Но есть и преимущества, ради которых мы и будем использовать такой подход. В чем же состоят эти преимущества? Дело в том, что в теле функции-обработчика события `onclick`, в нашем случае это функция `bhv.tabbedpane.selectTab`, HTML-элемент `SPAN` будет доступен как неявный параметр `this`, следовательно, и модель данных будет доступна как свойства неявного параметра `this`, и это для несложных моделей позволяет создавать более простой код:

```
bhv.tabbedpane.selectTab = function() {
  if (! this.loaded) {
    this.loaded = true;
    bhv.sendRequest("GET", this.address, null, true,
      bhv.tabbedpane.loadTab, bhv.tabbedpane.errorTab, [this]);
  }
  var barPane = this.parentNode;
  var divs = barPane.childNodes;
  for(var I = 0; i < divs.length; i++)
    if (divs[i].tagName == "SPAN")
      bhv.tabbedpane.unSelectTab.call(divs[i]);
  this.className = "tabSelected";
  this.pane.className = "paneSelected";
  this.pane.style.height =
    (parseInt(this.parentNode.parentNode.style.height)-
     this.parentNode.offsetHeight -
     this.parentNode.parentNode.tabbedBar.offsetHeight -
     this.parentNode.parentNode.space.offsetHeight) + "px";
}
```

Вы видите, что все данные, необходимые для работы компонента, доступны через неявный параметр `this`. Объект `this` передается функции `sendRequest()`, а функция `sendRequest()` передает этот объект в массиве аргументов функциям-обработчикам правильного и ошибочного запроса:

```
bhv.tabbedpane.loadTab = function(thi){
  thi.pane.innerHTML = bhv.relocateSRC(this.responseText, thi.address )
}

bhv.tabbedpane.errorTab = function(thi){
  thi.pane.innerHTML = "<h1>Страница недоступна</h1>"
}
```

Обе функции обновляют содержимое активной панели, ссылка на которую хранится в свойстве `pane` закладки. Хранение модели в пользовательском объекте JavaScript, а не прямо в свойствах HTML-элемента привело бы к более сложному программному коду.

Если вы обратили внимание, перед присваиванием свойству `innerHTML` текст запроса обрабатывается функцией `relocateSRC()`. Для чего необходимо обрабатывать загружаемый документ, мы разберем на примере.

Давайте подумаем над тем, что произойдет, если в одну из панелей нашего компонента загружается содержимое HTML-документа из произвольной папки. Текст документа будет отображен правильно, изображения, заданные абсолютными URL-адресами и абсолютными путями, будут загружены правильно. Но изображения, заданные относительными URL-адресами и относительными путями, будут загружаться правильно только как исключение. Дело в том, что для загружаемого в "Панель с закладками" HTML-документа текущая папка может не совпадать с папкой текущего HTML-документа. Предположим, текущим HTML-документом является \bhvajax\tabbedpane\tabbedpane.html. В компонент "Панель с закладками", расположенный в текущем документе, загружается HTML-документ \bhvajax\text\text1.html, в котором задано изображение с относительным URL-адресом:

```
<img src="img/image1.gif">
```

Путь к изображению задан относительно папки \bhvajax\text, и файл с изображением расположен по адресу \bhvajax\text\img\image1.gif. В то же время для текущего HTML-документа \bhvajax\tabbedpane\tabbedpane.html текущей папкой является \bhvajax\tabbedpane, и Web-браузер будет искать это изображение по адресу \bhvajax\tabbedpane\img\image1.gif. Это означает, что текст загружаемого HTML-документа будет правильно отображен Web-браузером, а вот прочие ресурсы, такие как изображения, не будут загружены, если они заданы относительными URL-адресами и относительными путями. Следующий раздел мы как раз и посвятим решению этой важной и интересной проблемы.

3.3. Способы задания URL-адресов в HTML-документах

Адрес Web-ресурса может быть задан как абсолютный URL-адрес, который начинается с имени протокола:

http://localhost:8080/bhvajax/tabbedpane/tabbedpane.html

Вы, наверное, редко задаете URL-адрес в такой полной форме. Имя протокола для Web-браузера часто пропускается, т. к. неявно подразумевается http-

протокол. Имя хоста задается только для ссылок на хосты, отличные от хоста, в котором расположен текущий документ. Номер порта тоже может быть пропущен, если используется порт по умолчанию. Для http-протокола это порт 80.

После того как первая страница загружена, вы можете записывать адрес в виде относительного URL-адреса, который может начинаться с косой черты, что означает задание пути относительно корневого каталога хоста:

/bhvajax/text/img/image1.gif

или задаваться относительно текущей папки. Вариант задания относительно текущей папки может выглядеть как:

./img/image1.gif

или как

img/image1.gif

В некоторых случаях задается папка относительно текущей папки вверх по иерархии папок:

../text/img/image1.gif

Каждая серия символов **../** означает переход в папку на один уровень вверх по иерархии папок. Например, на две папки вверх по иерархии:

../../bhvajax/text/img/image1.gif

В данном случае два последних URL-адреса ссылаются на один и тот же ресурс, только первый путь задан через родительскую папку, а второй — через папку, родительскую для родительской папки.

Вы должны хорошо ориентироваться в терминологии относительных URL-адресов, т. к. они используются очень часто. Задание относительных URL-адресов позволяет перемещать ваши приложения из папки в папку и даже с одного хоста на другой хост с минимальными изменениями в HTML-документах.

Как я уже обращал ваше внимание, относительные адреса могут создавать и неудобства при работе Ajax-приложений. Например, когда мы загружали в панели компонентов "Аккордеон" и "Панель с закладками" документы из текущего каталога, все работало правильно, т. к. все относительные адреса оставались правильными и для загружаемого документа, и для текущего документа. Ведь для обоих документов текущей папкой была одна и та же папка. Теперь мы усложнили задачу и стали загружать в панели текущего документа документы из соседних папок, и сразу же столкнулись с проблемой. Рисунки перестали загружаться, т. к. их адрес задан относительно папки, в которой расположен загружаемый асинхронным запросом документ,

а Web-браузер пытается вычислить их URL-адрес относительно текущей папки основного документа. Давайте рассмотрим, как можно решить эту проблему на простом примере. Мы загружаем в "Панель с закладками" документ **../text/text1.html**, в котором задано изображение:

```
<img src="img/image1.gif">
```

Для того чтобы получить файл с изображением, мы должны привести адрес, заданный относительно файла **../text/text1.html**, к адресу, заданному относительно текущего документа. В итоге должен получиться такой адрес: **../text/img/image1.gif**.

Можно использовать приведение пути к текущему каталогу, а можно привести его и к корневому каталогу хоста. Исходя из того, что текущий документ имеет абсолютный путь **/bhvajax/tabbedpane/tabbedpane.html**, путь **../text/img/image1.gif** приводится к абсолютному пути **/bhvajax/text/img/image1.gif**.

Только что мы выполнили приведение папки к текущему каталогу и к корневому каталогу хоста вручную. А теперь разработаем функцию, которая могла бы в любом месте кода правильно преобразовать относительный путь в абсолютный.

Функция `getAbsolutePath()` будет приводить все относительные пути к абсолютному пути, заданному от корневого каталога хоста. Такой путь не будет включать имя протокола, имя домена и порт. Поэтому, строго говоря, это будет тоже относительный URL-адрес, заданный относительно корневого каталога Web-сервера. Сравните абсолютный URL-адрес:

http://localhost:8080/bhvajax/tabbedpane/tabbedpane.html

и абсолютный путь, являющийся относительным URL-адресом, который задан от корневого каталога хоста:

/bhvajax/tabbedpane/tabbedpane.html

с относительным URL-адресом, заданным с помощью относительного пути относительно папки, в которой содержится текущий документ:

../text/text1.html

img/image1.gif

То есть абсолютный URL-адрес и абсолютный путь — это разные понятия. Абсолютный URL-адрес однозначно определяет Web-ресурс в Интернете, а абсолютный путь однозначно определяет Web-ресурс относительно корневого каталога хоста. Теперь, когда мы разобрались с терминологией, перейдем к разработке функции.

Цель нашей функции `getAbsolutePath()` — преобразовать относительный путь, который задан в качестве параметра функции, в абсолютный путь отно-

сительно корневого каталога хоста. Или на нашем сквозном примере, если мы вызовем функцию с одним параметром:

```
getAbsolutePath("../text/text1.html")
```

из HTML-документа **/bhvajax/tabbedpane/tabbedpane.html**, то функция должна преобразовать заданный относительный путь к абсолютному пути:

/bhvajax/text/text1.html

А если мы вызовем функцию с двумя параметрами:

```
getAbsolutePath("img/image1.gif", "../text/text1.html")
```

из HTML-документа **/bhvajax/tabbedpane/tabbedpane.html**, то функция должна привести первый параметр, в котором путь задан относительно второго параметра, к абсолютному пути, заданному от корневого каталога хоста:

/bhvajax/text/img/image1.gif

Ведь именно по этому адресу и находится интересующий нас Web-ресурс — изображение image1.gif.

Прежде чем приступить к преобразованию адресов, необходимо определить точку, в которой находится текущий документ. Это можно сделать при помощи объекта `window.location`. Объект `window.location` содержит ряд свойств, в которых содержится информация о URL-адресе текущего документа, загруженного в Web-браузер (табл. 3.1).

Таблица 3.1. *Свойства объекта* `window.location`

Свойство	Значение свойства	Пример
`href`	Абсолютный URL-адрес текущего документа	`http://localhost:8080/bhvajax/test.php?page=1#first`
`protocol`	Протокол	`http:`
`host`	Имя и порт хоста	`localhost:8080`
`hostname`	Имя хоста	`localhost`
`port`	Порт	`8080`
`pathname`	Путь относительно корневого каталога хоста	`/bhvajax/test.php`
`search`	Часть URL после знака `?`, включая знак `?`, которая обычно содержит http-параметры запроса типа `GET`	`?page=1`
`hash`	Часть URL после знака `#`, включая знак `#`, которая, как правило, содержит идентификатор `id` элемента или имя `name` якоря (`<A>`), на который производится автоматическая прокрутка загружаемого документа	`#first`

Из всех свойств, для решения нашей задачи лучше всего подходит свойство `window.location.pahtname`, которое содержит путь к текущему документу, заданный относительно корневого каталога хоста. Для того чтобы проследить работу нашего алгоритма преобразования относительных путей в абсолютные, рассмотрим в табл. 3.2, как будут преобразовываться конкретно заданные пути по шагам алгоритма.

Таблица 3.2. *Алгоритм преобразования относительных путей в абсолютные пути*

Номер шага	Текущий документ	Документ, относительно которого задан искомый путь	Искомый путь
0	window.location.pathname	../text/text1.html	img/image1.gif
1	/bhvajax/tabbedpane/	../text/	img/image1.gif
2	/bhvajax/tabbedpane/../text/img/image1.gif		

В начале работы алгоритма мы имеем путь `window.location.pathname` текущего документа и два относительных пути, переданных в качестве фактических параметров функции, которую нам еще предстоит разработать. На первом шаге алгоритма (см. табл. 3.2) мы из путей, заданных как формальные параметры, выделяем путь к папке и исключаем имя файла, получая два пути к папкам **/bhvajax/tabbedpane/** и **../text/**. На втором шаге достаточно просто сделать конкатенацию полученных путей, в результате чего получим немного непривычный путь **/bhvajax/tabbedpane/../text/img/image1.gif**. Спецификация допускает использование таких URL-адресов. При этом фрагмент адреса **tabbedpane/../** указывает на то, что был осуществлен заход в папку tabbedpane и возврат обратно в родительскую папку bhvajax, т. е. пути **/bhvajax/tabbedpane/../text** и **/bhvajax/text** — это один и тот же путь. Продолжим этот пример и зададим адрес с использованием двух подъемов вверх по иерархии папок **../../**. Путь **/bhvajax/tabbedpane/../../** ведет к корневому каталогу хоста. Поэтому наш путь к папке text пришлось бы задать как **/bhvajax/tabbedpane/../../bhvajax/text/**.

Когда я разрабатывал первый вариант функции `getAbsolutePath()`, то приводил все пути, которые содержат **../**, к простому виду, но потом сообразил, что делать так не имеет смысла, и окончательный вариант функции в листинге 3.1 возвращает не совсем привычные, но полноценно работающие во всех случаях пути, заданные от корневого каталога хоста.

Листинг 3.1. Преобразование относительного пути в абсолютный

```
bhv.getAbsolutePath = function(path, relative) {

  path = path.replace(/\\/g, "/");
  if (path.substring(0, 1) == "/")
    return path;

  var current = document.location.pathname;
  current = current.replace(/\\/g, "/");
  current = current.substring(0, current.lastIndexOf("/") + 1);

  if (relative) {
    relative = relative.replace(/\\/g, "/");
    relative = relative.substring(0, relative.lastIndexOf("/") + 1);
    if (relative.substring(0, 1) == "/")
      current = relative;
    else
      current = current + relative;
    }

  return current + path;
}
```

Давайте разберем некоторые особенности работы функции. Как и обычно, функцию мы привязываем к объекту пространства имен `bhv` и помещаем определение функции в файле util.js, поскольку такая функция может нам понадобиться в других приложениях. В функцию может передаваться один или два параметра. В параметре `path` задается искомый путь, который необходимо преобразовать к абсолютному пути. Во втором параметре передается путь, относительно которого задан параметр `path`. Если второй параметр не задан, преобразование происходит относительно пути текущего документа.

В различных операционных системах для разделения имен папок используются символ косой черты или обратной косой черты. Чтобы упростить код функции, символ обратной косой черты в пути заменим символом косой черты:

```
path = path.replace(/\\/g, "/");
var current = document.location.pathname;
current = current.replace(/\\/g, "/");
relative = relative.replace(/\\/g, "/");
```

Приведенный фрагмент программного кода использует регулярные выражения. Наверное, работа с регулярными выражениями — тема для отдельного и очень серьезного разговора, но не при первом знакомстве с JavaScript. Если вы уже знакомы с регулярными выражениями по Perl или PHP 5, для вас будет новым только то, что некоторые методы работы с регулярными выражениями, в частности метод `replace()`, являются методами объекта типа `String`, которым в качестве параметра передается объект типа регулярное выражение. Регулярное выражение можно создавать при помощи обычного для Perl-программистов синтаксиса:

```
/.../g
```

Функцию `replace()` можно применять как к переменным, так и к строковым литералам:

```
var str = "Часть строки будет заменена".replace(/будет/g, "уже");
// "Часть строки уже заменена"
```

Символ `g` (global) в конце шаблона называют модификатором, который говорит о том, что необходимо произвести замену всех вхождений строки поиска. Без модификатора будет произведена только замена первого вхождения строки поиска.

Путь, заданный для преобразования в абсолютный путь, может и без преобразования оказаться абсолютным путем. Следующий фрагмент кода проверяет, не является ли искомая строка абсолютным путем:

```
if (path.substring(0, 1) == "/")
  return path;
```

Если искомая строка задана абсолютным путем, ее не следует преобразовывать. Возникает вопрос: зачем вызывать функцию преобразования для абсолютных путей? Ответ следующий: функция будет вызываться для обработки документов, о которых неизвестно, будут ли адреса в них заданы абсолютными или относительными путями. Можно было бы проверку на абсолютность пути осуществлять перед вызовом функции. Но проще заложить проверку в саму функцию. Тогда и для абсолютных, и для относительных путей функция будет работать правильно.

Ну и, наконец, в следующих строках функции от путей отсекаются имена файлов:

```
var current = document.location.pathname;
current = current.replace(/\\/g, "/");
current = current.substring(0, current.lastIndexOf("/") + 1);

relative = relative.replace(/\\/g, "/");
relative = relative.substring(0, relative.lastIndexOf("/") + 1);
```

Все, что остается сделать, — вернуть в вызывающую программу абсолютный путь Web-ресурса:

```
return current + path;
```

Итак, функция `getAbsolutePaht()` разработана. Но это еще не все. Необходимо в загружаемом в панель с закладками HTML-документе выделить пути к Web-ресурсам и преобразовать их к абсолютному виду. Наша задача упрощается тем, что такие пути будут заданы в тегах HTML-элементов в атрибутах `src` или `href`.

Мы уже познакомились с простейшими регулярными выражениями, когда заменяли знак обратной косой черты в функции `getAbsolutePath()`. Функция `relocateSRC()` из листинга 3.2 использует более сложные регулярные выражения и только что рассмотренную нами функцию `getAbsolutePath()` для преобразования HTML-документа.

Листинг 3.2. Преобразование путей в HTML-документе регулярными выражениями

```
bhv.relocateSRC = function(htmlText, relative) {
  var newText = htmlText.replace(
    /(<[^>]*\s(src|href)\s*=\s*(\"|\'))(.*)(\3[^>]*>)/gi,
    "$1" + bhv.getAbsolutePath("$4", relative) + "$5");
  newText = newText.replace(
    /(<[^>]*\s(src|href)\s*=\s*)([^\s\"\'>]+([^>]*>)/gi,
    "$1" + bhv.getAbsolutePath("$3", relative) + "$4");
  return newText;
}
```

Я должен обратить ваше внимание, что использование регулярных выражений — очень мощное средство в руках JavaScript-программиста. Но для тех, кто не знаком с синтаксисом регулярных выражений, приведенный в листинге 3.2 программный код может показаться непонятным. Поэтому я сделаю несколько комментариев, а затем приведу программный код функции, которая проделывает ту же самую работу без использования регулярных выражений. Это я сделаю для пользы тех читателей, которые только приступили к изучению JavaScript, и с моей стороны было бы просто невежливо решать задачу двумя строчками кода, более похожими на шифрограмму, чем на листинг JavaScript.

В функции используются два шаблона регулярных выражений. Первый шаблон — для атрибутов `src` и `href`, значения которых заданы с использованием

одинарных или двойных ограничивающих кавычек, а второй шаблон — для атрибутов, значения которых заданы без ограничивающих кавычек.

Признаюсь, что я не сразу нашел решение задачи при помощи регулярных выражений. Поэтому в один прекрасный момент я сказал себе: ищу решение ровно три минуты и, если не нахожу, пишу обычную функцию с посимвольным чтением документа. Иначе вы могли бы и не дождаться этой книги. К счастью такая функция, с которой вы можете познакомиться в листинге 3.3, оказалась намного проще, чем я сначала мог это предположить. Но вот вопрос: является ли эта функция проще для восприятия, чем функция из листинга 3.2? Я думаю, возможности, которые получает JavaScript-программист, освоивший работу с регулярными выражениями, с лихвой окупает время, потраченное на изучение.

Листинг 3.3. Преобразование путей в HTML-документе без регулярных выражений

```
bhv.simpleRelocateSRC = function(htmlText, relative) {
  var newText = "";
  var symbol = "";
  var beforURL = "";
  var someURL = "";
  var isTag = false;
  var isSRC = false;
  var isAttr = false;
  var isURL = false;
  for (var i = 0; i < htmlText.length; i++) {
    var symbol = htmlText.substring(i, i + 1);
    if (symbol == "<"){
      isTag = true;
      isSRC = false;
      isAttr = false;
      newText += symbol;
    } else if (symbol == ">") {
      isTag = false;
      isSRC = false;
      isAttr = false;
      newText += symbol;
    } else if (isTag
        && htmlText.substring(i, i + 4).toLowerCase() == " src") {
```

```
    isSRC = true;
    isAttr = false;
    newText += htmlText.substring(i, i + 4);
    i += 3;
  } else if (isTag
     && htmlText.substring(i, i + 5).toLowerCase() == " href") {
    isSRC = true;
    isAttr = false;
    newText += htmlText.substring(i, i + 5);
    i += 4;
  } else if (isTag && isSRC && symbol == "=") {
    isAttr = true;
    newText += symbol;
  } else if (isTag && isSRC && isAttr && symbol == '"') {
    var index = htmlText.indexOf('"', i + 1);
    someURL = htmlText.substring(i + 1, index);
    someURL = bhv.getAbsolutePath(someURL, relative);
    newText += ('"' + someURL + '"');
    i = index;
    isSRC=false;
    isAttr = false;
  } else if (isTag && isSRC && isAttr && symbol == "'") {
    var index = htmlText.indexOf("'", i + 1);
    someURL = htmlText.substring(i + 1, index);
    someURL = bhv.getAbsolutePath(someURL, relative);
    newText += ("'" + someURL + "'");
    i = index;
    isSRC = false;
    isAttr = false;
  } else if (isTag && isSRC && isAttr && symbol != " ") {
    var index = Math.min(
     (htmlText + " ").indexOf(" ", i + 1), htmlText.indexOf(">", i + 1));
    someURL = htmlText.substring(i, index);
    someURL = bhv.getAbsolutePath(someURL, relative);
    newText += (" " + someURL + " ");
    i = index - 1;
    isSRC = false;
    isAttr = false;
  } else
```

```
      newText += symbol;
  }
  return newText;
}
```

Как вы можете заметить, чтобы обеспечить функциональность, эквивалентную двум строчкам кода с использованием регулярных выражений, нам пришлось выполнять довольно сложные проверки и устанавливать значения всевозможных флагов — логических переменных. Основные моменты в работе функции `simpleRelocateSrc()` следующие. Обработка строки, содержащей HTML-документ, происходит в цикле, который просматривает текст по одному символу:

```
for (var i = 0; i < htmlText.length; i++) {
  var symbol = htmlText.substring(i, i + 1);
```

Если текст не содержит путь к Web-ресурсу, он просто посимвольно переписывается в новую переменную:

```
newText += symbol;
```

Как же выявляются и преобразуются пути к Web-ресурсам? Такие пути могут находиться только внутри тега HTML-элемента. Поэтому отслеживаются символы < и >, которые служат ограничителями тегов:

```
if (symbol == "<")
  isTag = true;
if (symbol == ">")
  isTag = false;
```

Если мы находимся внутри тега, переменная `isTag` принимает логическое значение `true`. Внутри тега производится поиск атрибутов `src` и `href`:

```
if (isTag && htmlText.substring(i, i + 4).toLowerCase() == " src")
```

Найдя атрибут `src` или `href`, функция находит значение этих атрибутов, которое может быть заключено между парой одинарных или двойных кавычек, либо (в HTML это допускается) без кавычек после знака "равно" до конца слова. Разберем код для случая, когда значение атрибута выделено парой двойных кавычек:

```
if (isTag && isSRC && isAttr && symbol == '"') {
  var index = htmlText.indexOf('"', i + 1);
  someURL = htmlText.substring(i + 1, index);
  someURL = bhv.getAbsolutePath(someURL, relative);
  newText += ('"' + someURL + '"');
```

```
  i = index;
  isSRC = false;
  isAttr = false;
}
```

Сначала ищем индекс закрывающей кавычки:

```
var index = htmlText.indexOf('"', i + 1);
```

Затем запоминаем значение атрибутов переменной и преобразуем путь к абсолютному виду:

```
someURL = htmlText.substring(i + 1, index);
someURL = bhv.getAbsolutePath(someURL, relative);
```

Далее остается записать преобразованный путь в новый документ, прокрутить индекс цикла до индекса закрывающей кавычки и присвоить переменным `isSRC` и `isAttr` логическое значение `false`, в знак того, что обработка атрибута `src` и `href` закончена:

```
newText += ('"' + someURL + '"');
i = index;
isSRC = false;
```

Теперь можно обрабатывать функцией `simpleRelocateSCR()` загружаемый в панель с закладками HTML-документ. При этом все относительные адреса будут правильно преобразованы, и HTML-элементы `IMG`, `A`, т. е. изображения и гиперссылки, будут иметь правильные адреса в атрибутах `src` и `href`.

Косвенной пользой от изучения кода этой функции будет то, что вы можете оценить, насколько ограничение значений атрибутов кавычками может облегчить разбор текста HTML-документа, и с этого момента будете все атрибуты заключать в кавычки.

Из первых глав книги вы уже знаете, что атрибуты `src` есть также у HTML-элементов `LINK` и `SCRIPT` (и еще у некоторых других, о которых мы пока не говорили). Все эти HTML-элементы будут содержать правильные абсолютные пути в атрибутах `src` и `href`, но это не означает, что соответствующие таблицы стилей и скрипты будут загружены в Web-браузер при загрузке текста HTML-документа асинхронным запросом. Ведь мы не открываем новый HTML-документ в Web-браузере, а всего лишь изменяем свойство `innerHTML` у одного из элементов, загруженного в Web-браузер HTML-документа:

```
bhv.tabbedpane.loadTab = function(thi) {
  thi.pane.innerHTML = bhv.relocateSRC(this.responseText, thi.address )
}
```

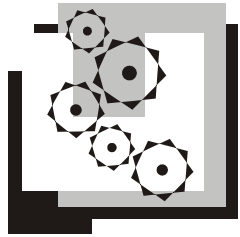
Для HTML-элемента `LINK`, содержащего ссылку на css-файл с определением стилей, соответствующий файл будет автоматически загружаться лишь некоторыми Web-браузерами. А код JavaScript, заданный в элементе `SCRIPT`, не будет загружен вовсе. Поэтому загружать таким способом можно только текст HTML-документов и изображения `IMG`.

Существуют хорошие библиотеки, которые производят разбор загружаемого асинхронными запросами текста HTML-документа, и в случае, если текст содержит HTML-элементы `SCRIPT` — могут загрузить и выполнить скрипты. Есть возможность кроссбраузерно загружать элементы `LINK`, которые содержат ссылки на css-файлы с определениями стилей. Но мы не будем касаться этой более сложной темы и закончим на этом разработку компонента "Панель с закладками".

* * *

Подведем итоги. Мы создали компонент "Панель с закладками". Стилевое оформление компонента вынесено в отдельный файл tabbedpane.css, поэтому вы можете изменять по мере необходимости стилевое оформление компонента. При этом код в файле tabbedpan.js менять не придется.

При разработке компонента мы научились создавать рисованные при помощи свойства стиля `background-image` компоненты, которые расширяют возможности Web-дизайнера. И, наконец, мы научились правильно преобразовывать относительные пути в атрибутах `src` HTML-элементов, таких как `IMG`, в загружаемых асинхронными запросами документах.



Глава 4

Работа с XML-документами средствами JavaScript

XML-документы в Ajax-технологии вы будете использовать в каждом приложении. Мы уже обсуждали в *главе 2*, что наиболее универсальным способом обмена между Web-браузером и Web-сервером является вариант, в котором ответ Web-сервера содержит XML-документ. Структура такого документа может быть согласована на самых ранних этапах разработки приложения, после чего разработчики серверной и клиентской части Ajax-приложения могут относительно независимо заниматься каждый своей частью работы, имея достаточные гарантии, что после соединения воедино серверной и клиентской частей приложения все будет работать как и предполагалось.

Для удобства работы с XML-документами, объект `XMLHttpRequest` имеет свойство `responseXML`, в котором ответ сервера содержится в виде объекта типа `Document` (`DOMDocument`), представляющего собой корневое дерево объектов типа `Node`. На самом деле мы уже начали знакомиться с методами объектов `Document` и `Node`, работая с объектной моделью (DOM) HTML-документа. Современные Web-браузеры позволяют работать одинаковыми методами и с узлами XML-документа, и с элементами HTML-документа. Если вы хотите познакомиться со всеми подробностями, могу рекомендовать детально изучить спецификацию *"Document Object Model (DOM) Level 1 Specification Version 1.0 W3C Recommendation 1 October, 1998"*, в которой определена объектная модель XML- и HTML-документов, реализованная в современных Web-браузерах. С наиболее часто используемыми приемами работы с XML- и HTML-документами мы познакомимся в этой главе.

Текст примеров данной главы вы можете найти в папке \bhvajax\xml на прилагаемом к книге компакт-диске.

4.1. Структура XML-документа

XML-документ состоит из двух частей — пролога и тела документа. Пролог является необязательным и состоит из двух необязательных разделов. Первый необязательный раздел пролога XML-документа называется *объявлением XML*:

```
<?xml version="1.0" encoding="Windows-1251"?>
```

Объявление XML, если оно присутствует в документе, всегда должно содержаться в первой строке. В атрибуте `encoding` задается кодировка символов. По умолчанию используется кодировка UTF-8. Задавать кодировку в объявлении XML можно рекомендовать всегда, если не используется кодировка по умолчанию UTF-8. Это помогает Web-браузеру правильно распознать кодировку XML-документа.

Вторая необязательная часть пролога — объявление типа документа, которое может содержать определение типа документа или ссылку на ресурс с таким определением. Определение типа (схемы) документа мы не будем использовать в этой книге по двум причинам. Во-первых, потому что это отвлекло бы нас от основной темы повествования и, во-вторых, потому что мы будем использовать сравнительно простые XML-документы, для которых тип (схему) документа будем задавать на интуитивном уровне. Определение типа (схемы) XML-документа задает, какие элементы (теги) будут использоваться, их имена, атрибуты, допустимый порядок вложенности и следования элементов друг за другом. Мы уже использовали определение типа документа для HTML-документов:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
```

Да, это то самое объявление типа документа. А определение типа документа HTML Transitional расположено по адресу **http://www.w3.org/TR/html4/loose.dtd**. Вы можете прямо сейчас обратиться к этому ресурсу и посмотреть на определение типа документа DTD HTML Transitional.

После пролога документа следует тело документа. Разметка XML-документа похожа на разметку HTML-документа. Но есть и некоторые отличия. Разметка XML более строгая и не допускает пропуска отдельных элементов, например закрывающих тегов. Каждый открывающий тег должен иметь свой закрывающий тег:

```
<test>
  <element0>
    <element1/>
```

```
  </element0>
  <element0>
    <element1/>
  </element0>
</test>
```

Все элементы XML-документа находятся в теле единственного корневого элемента, имя которого совпадает с именем XML-документа. В приведенном примере это корневой элемент `test`. Если элемент не имеет тела, он называется *пустым элементом* и записывается в специальной форме:

```
<element1/>
```

Пустой элемент записывается одним тегом и не требует наличия закрывающего тега.

Если в HTML множество имен элементов (тегов) строго определено, то в XML разработчик сам для себя задает множество имен элементов. Делается это в определении типа документа, которое мы договорились не использовать. Для документов без объявленного типа документов можно использовать произвольные имена тегов, которые являются допустимыми именами в XML. Имена должны начинаться с буквы, содержать буквы, цифры, символ подчеркивания или символ двоеточия. Символ двоеточия используется специальным образом для задания пространств имен, и мы не будем применять его в нашей книге.

XML-элементы могут иметь атрибуты, которые задаются в открывающем теге элемента в форме *имя_атрибута*="*значение_атрибута*". В отличие от HTML, в XML атрибуты всегда должны быть ограничены кавычками:

```
<test>
  <element0 attr1="value1">
    <element1 attr2="value2"/>
  </element0>
  <element0>
    <element1/>
  </element0>
</test>
```

Элементы XML-документа могут содержать другие элементы и текст:

```
<test>
  <element0 attr1="value1">
    Элементы могут содержать текст.
    <element1 attr2="value2"/>
```

```
    Текст в теле элемента может использоваться
    наряду с другими элементами
  </element0>
  <element0>
    Элемент содержит только текст
  </element0>
  <element0>
    <element1/>
  </element0>
</test>
```

При разметке XML-документа необходимо соблюдать правильную вложенность элементов. И открывающий, и закрывающий теги элемента должны полностью находиться в теле одного элемента.

Текст, который не является разметкой, в XML называют *символьными данными*. К разметке относятся открывающие и закрывающие теги элементов, теги пустых элементов, ограничители разделов `CDATA`, символьные мнемоники (`&`, `<`, `>`, `&quot;`, `&apos;`) и некоторые другие средства разметки.

С открывающими и закрывающими тегами элементов мы уже познакомились (да, собственно, были давно уже знакомы по HTML), теги пустых элементов немного отличаются от того, что нам известно из HTML. С остальными, из перечисленных выше, видами разметки мы сейчас познакомимся.

Мнемоники `&`, `<`, `>`, `&quot;`, `&apos;` используются вместо символов `&` (амперсанд), `<` ("меньше чем" или левая угловая скобка), `>` ("больше чем" или правая угловая скобка), `"` (кавычка или двойная кавычка), `'` (одинарная кавычка или апостроф). Мнемоника начинается знаком `&` и заканчивается точкой с запятой. Для того чтобы применять мнемоники, необходимо понимать, зачем они используются.

Знак `<` ("меньше чем" или левая угловая скобка) в XML-документе имеет совершенно особое значение, поскольку используется для выделения в документе тегов элементов и других средств разметки. Поэтому, если знак `<` не является элементом разметки, вместо него необходимо использовать мнемонику `<`. Это обязательно нужно делать в текстовых разделах и в атрибутах элементов:

```
<test>
  <element0 attr1="Использование &amp;lt; вместо &lt;">
    В текстах и атрибутах вместо символа &lt;
    необходимо использовать мнемонику &amp;lt;
  </element0>
```

```
  <element0 attr1="Использовать &amp;amp; вместо &amp;">
    В текстах и атрибутах вместо символа &amp;
    необходимо использовать мнемонику &amp;amp;
  </element0>
</test>
```

Для обозначения мнемоник в качестве первого символа выбран символ `&` (амперсанд). Следовательно, как и мнемоника `<`, мнемоника `&` должна использоваться вместо символа `&` в тестовых разделах и атрибутах элементов.

В строке атрибутов элементов вместо кавычек нужно использовать `&quot;` или `&apos`. В то же время, использовать мнемоники `&quot;` или `&apos` в текстовых данных нет необходимости:

```
<test>
  <element0 attr1="Использование &quot;">
    В текстах использовать &amp;quot; нет необходимости.
    Можно использовать непосредственно знак " и знак '.
  </element0>
</test>
```

Если в XML-документе находится текст программы JavaScript, ограничение на использование знака `<` может оказаться неприемлемым. В текстовых данных его нужно заменить на мнемонику, но это не всегда допустимо. Специально для тех случаев, когда в тексте надо указывать знаки `&`, `<` и другие средства XML-разметки, используют разделы `CDATA`. Раздел `CDATA` открывается последовательностью символов `<![CDATA[` и закрывается последовательностью символов `]]>`. Поэтому сама последовательность `]]>` в теле секции `CDATA` не может применяться. Только для этой цели служит мнемоника `>` ("больше чем" или правая угловая скобка). Использовать мнемонику `>` в других случаях, кроме секций `CDATA`, и в иных сочетаниях, кроме `]]>`, нет необходимости:

```
<test>
  <element0 attr1="Использование &quot;">
    <![CDATA[
      Разделы CDATA могут содержать любые символы,
      в том числе <, &, > и <![CDATA[,
      но не сочетание ]]&gt;, которое является
      признаком окончания раздела CDATA
    ]]>
  </element0>
</test>
```

Те из вас, кто использует XHTML, т. е. такой стиль разметки HTML-документа, который одновременно отвечает спецификации XML, должны применять такую разметку для своих скриптов:

```
<html>
<head>
<script type="text/javascript">
<![CDATA[
String.prototype.isEmpty = function() {
  if (this.replace(/[\r|\n]/g,"").search(/\S+/) < 0)
    return true;
  else
    return false;
}
//]]>
</script>
</head>
<body>
</body>
</html>
```

Вы можете заметить, что приведенный фрагмент является одновременно и HTML-документом, и XML-документом. Для этого мы поместили весь документ в корневой элемент `HTML`, который в чистом HTML допускается пропускать (в этом случае он будет создан Web-браузером автоматически). Далее, мы заключили текст JavaScript, содержащий символ < ("меньше чем") в секцию `CDATE`.

Вторым способом включить текст программы JavaScript в текст XML- и XHTML-документа является использование комментариев. *Комментарии* в XML имеют синтаксис, похожий на HTML-комментарии:

```
<!-- Это комментарий в XML -->
```

Внутри комментария допускается использование любых символов, но для обеспечения совместимости запрещено указывать два подряд идущих знака - (минус), т. е. --. Так что любители использовать декремент в JavaScript должны помнить об этой особенности. Фрагмент XHTML-документа с использованием комментариев в элементе `script` приведен ниже:

```
<script type="text/javascript">
<!--
String.prototype.allTrim = function() {
  return this.replace(/^\s+|\s+$/g, "");
```

```
};
//-->
</script>
```

Кстати, в секции `CDATA` тоже можно случайно использовать недопустимую последовательность `]]>`:

```
<script type="text/javascript">
<!--
<![CDATA[
var arr1 = [1, 2, 3];
var arr2 = [[1, 2, 3], [4], [5], [6]];
if (arr2[0, arr1[0]]>=1)
  alert("В секции CDATA недопустимо использовать ]]>")
}
//]]>
//-->
</script>
```

Конечно, я дополнительно взял секцию `CDATA` в комментарии, так что код остался правильным. Но если бы комментариев не было, код содержал бы с точки зрения XML две ошибки. Использовать ли для кода JavaScript секции `CDATA` или комментарии — зависит от вашей задачи. Недостатком использования комментариев является то, что программные средства, которые работают с XML-документами, могут, но не обязаны извлекать текст комментариев из XML-документа (хотя спецификация и не запрещает этого).

Мы рассмотрели следующие средства, использующиеся в XML-документах: элементы, пустые элементы, атрибуты элементов, текстовые данные, секции `CDATA`, текстовые мнемоники и комментарии. Это наиболее простые и часто применяемые программистами средства XML. Программистам, знакомым с HTML-разметкой, основы работы с XML не покажутся чем-то новым. В табл. 4.1 приведены в сводном виде основные точки соприкосновения и различия HTML и XML.

Таблица 4.1. *Сравнение языков разметки текста HTML и XML*

HTML	**XML**
HTML-разметка определяет внешний вид документа	XML-разметка определяет данные документа и не предусматривает определенного визуального отображения

Таблица 4.1 *(продолжение)*

HTML	XML
HTML-документ используется для отображения в конкретной программе — Web-браузере	XML-документ не ограничен использованием совместно с конкретным программным обеспечением
Для разметки документа используются теги элементов	То же
Документ расположен в теле корневого элемента	То же
Имя корневого элемента всегда `HTML`	Имя корневого элемента всегда совпадает с именем документа
Имена элементов закреплены в спецификации HTML и являются нечувствительными к регистру, т. е. `HTML` и `html` — это один и тот же тег. Традиционно имена HTML-тегов записывались в верхнем регистре, но с появлением спецификации XHTML, в котором имена тегов заданы строго в нижнем регистре, в HTML-документах имена тегов стали записывать преимущественно в нижнем регистре	Имена тегов определяются в DTD (определении типа документа), если таковое задано в прологе документа. Имена тегов чувствительны к регистру, традиционно используются имена в нижнем регистре
Непарные теги записываются как открывающий тег. Например, ` `. Для совместимости с XML в спецификации XHTML такие теги записываются в формате ` `. В настоящее время, для совместимости с Web-браузерами, не распознающими XHTML, наличие пробела после имени тега обязательно	Пустые элементы записываются с помощью специального тега пустого элемента *<имя_тега/>*
Некоторые HTML-элементы могут быть пропущены и автоматически добавляются к объектной модели документа Web-браузером. У некоторых элементов могут быть пропущены закрывающие теги, если это не приводит к неоднозначности в определении документа. Отрывающие и закрывающие теги должны располагаться в теле родительского элемента, между его открывающим и закрывающим тегом. В случае некритической ошибки вложенности тегов Web-браузер пытается корректно отобразить документ, поэтому такие ошибки часто остаются незамеченными разработчиками	Требования к разметке строгие. Все непустые теги должны иметь закрывающий тег. И открывающий, и закрывающий теги элемента должны находиться в теле одного родительского элемента, между его открывающим и закрывающим тегами, обеспечивая правильную вложенность тегов. В случае ошибки вложенности тегов обрабатывающие программы, как правило, выдают сообщение об ошибке
В открывающих тегах элементов могут находиться атрибуты, которые задаются в виде *имя_атрибута*="*значение_атрибута*"	Атрибуты могут располагаться в открывающих тегах и в пустых тегах элементов

Таблица 4.1 *(окончание)*

HTML	XML
Имена атрибутов являются нечувствительными к регистру. ***Внимание!*** Если вы используете имена атрибутов HTML-элементов в программах на языке программирования JavaScript, то атрибуты элементов являются чувствительными к регистру, т. к. JavaScript является чувствительным к регистру в написании имен. При этом имена событий, как правило, задаются в нижнем регистре: `onclick`, `onmouseover`, а имена свойств "камелизируются" (от англ. *camel*) и задаются в виде `className`, `httpEquiv`	Имена атрибутов являются чувствительными к регистру
Значение атрибута может быть заключено в ограничивающие кавычки. Если значение атрибута не содержит пробельных символов, использовать ограничивающие кавычки необязательно. Для совместимости с XHTML рекомендуется всегда использовать ограничивающие кавычки в значениях атрибутов	Значение атрибутов всегда должно быть заключено в ограничивающие кавычки
Комментарии могут располагаться в теле элемента и обозначаются как `<!-- текст комментария -->` ***Внимание!*** В комментарии не допускается использование двух символов "минус", идущих подряд	То же

Если вы проанализируете табл. 4.1 (последний раз я делал это, сравнивая образы Онегина и Печорина), то можете сделать вывод, что правила, используемые для XML-документов, проще, чем правила для HTML-документов. Во всяком случае, это так, пока мы имеем дело с такими средствами XML, как элементы, атрибуты, текстовые данные и секции `CDATA`. И это не случайность. Если вы откроете спецификацию XML, то можете прочитать, что одной из целей создания XML являлась краткость, ясность, простота использования. И вы убедитесь, что мы именно так и будем использовать этот язык разметки в наших приложениях.

4.2. Варианты использования технологии XML в Ajax-приложениях

Как гласит народная мудрость — сколько ни говори XML, слаще от этого не станет. Давайте посмотрим, как технологию XML можно использовать в Ajax-приложениях.

Вспомним, что основной движущей силой Ajax-приложения является объект `XMLHttpRequest`, который отправляет запросы (request) на Web-сервер и получает ответы (response) с Web-сервера. Название объекта можно перевести как: XML-запросы к Web-серверу по протоколу HTTP. То есть предполагалось, что запросы к Web-серверу будут иметь тип `application/xml` (или устаревший тип `text/xml`). Давайте посмотрим, как может выглядеть запрос к Web-серверу типа `application/xml`:

```
<ajax_response>
  <user>11</user>
  <product>1111</product>
    <count>1</count>
  <session_id>abc123456</session_id>
</ajax_response>
```

Рассмотрим программный код JavaScript для отправки запроса типа `application/xml` на Web-сервер:

```
var ajax_response = form2xml();
var xmlHttrRequest = new XMLHttpReqyest();
xmlHttpRequest.open("POST", "order_product.php", true);
xmlHttprequest.setHeader("Content-Type: application/xml");
xmlHttpRequest.send(ajax_response);
```

Как видите, по сравнению с отправкой запроса типа `application/x-www-form-urlencoded`, изменился заголовок запроса, устанавливаемый в `setHeader()`. Все сложности формирования XML-документа для запроса скрыты от нас в функции `prepareAjaxResponse()`, которая создает необходимый XML-документ. Как вы понимаете, код такой функции может быть достаточно сложным. Поэтому созданы готовые библиотеки, которые специализируются на работе с XML-запросами, отправляемыми на Web-сервер. Если посмотреть на возможности, предлагаемые такими библиотеками, то их можно разделить на две категории. Первая категория — это библиотеки, которые на основании значений, введенных пользователем в элементы `INPUT`, расположенных в элементе `FORM` текущего HTML-документа, формируют XML-документ.

При этом атрибуты `NAME` или `ID` HTML-элементов `INPUT` становятся именами тегов XML-элементов. Например, для HTML-формы:

```
<form name="ajax_response">
  <input type="text" value="11" name="user" id="user">
  <input type="text" value="1111" name="product" id="product">
  <input type="text" value="1" name="count" id="count">
  <input type="hidden" value=" abc11111" name="session_id"
         id="session_id">
<form>
```

формируется XML-документ:

```
<ajax_response>
  <user>11</user>
  <product>1111</product>
  <count>1</count>
  <session_id>abc11111</session_id>
</ajax_response>
```

Иногда такой процесс не совсем точно называют сериализацией. Мы будем использовать более скромный термин — *преобразование*.

Аналогично преобразованию форм можно проводить преобразование объектов JavaScript:

```
var ajax_response = {};
ajax_response.user = 11;
ajax_response.product = 1111;
ajax_response.count = 1;
ajax_response.session_id = "abc11111";
xmlHttprequest.send(object2xml(ajax_response));
```

В данной главе мы создадим вспомогательный код для преобразования HTML-форм и JavaScript-объектов в XML-документы, а также для работы с DOM XML-документа.

4.2.1. Преобразование объектов JavaScript в XML-документ

JavaScript имеет очень удобные средства для преобразования объектов JavaScript в XML-документы. К свойствам и функциям-методам объектов JavaScript можно обращаться в двух нотациях: в привычной для многих объектно-ориентированных языков точечной нотации как к членам объекта и как

к элементам ассоциативного массива (при помощи квадратных скобок), что, в свою очередь, знакомо программистам Perl и PHP:

```
var obj0 = {
  property0: "string value";
  getProperty0: function() {
    return this.property0;
  }
}
object0.getProperty();
object0["getProperty"]();
var getName = "getProperty";
object0[getName]();
```

В JavaScript нет ассоциативных массивов, все это синтаксис доступа к членам (свойствам и методам) объектов. Наличие двух форм записи для доступа к членам объекта делает работу с объектами в JavaScript очень удобной и гибкой. Сравним две записи одного и того же свойства объекта:

```
object1.prop0.prop1.prop2.prop3;
object1["prop0"]["prop1"]["prop2"]["prop3"];
```

Первая форма записи через точку более привычна для программистов, работающих с объектно-ориентированными языками, а вторая форма записи, несмотря на некоторую непривычность, открывает широкие возможности для программной работы с объектами. Например, можно очень просто перебрать в цикле все свойства объектов:

```
for (var p in object3)
  alert("Свойство" + p + " имеет значение " + object3[p]);
```

В таком "двойным" синтаксисе можно получать доступ не только к свойствам, но и к функциям-методам объекта:

```
object1.prop0.prop1.prop2.prop3.method0();
object1["prop0"]["prop1"]["prop2"]["prop3"]["metod0"]();
```

В квадратных скобках можно использовать как строковые литералы, так и строковые переменные, и два синтаксиса доступа к членам объекта могут быть смешаны:

```
object1["prop0"].prop1["prop2"].prop3["metod0"]();
```

То есть любые сочетания употребления доступа к членам объекта допускаются и работают эквивалентно. Как можно использовать такую возможность, мы поймем прямо сейчас, когда будем преобразовывать JavaScript-объект

в XML-документ. Давайте создадим простейший JavaScript-объект и преобразуем его в XML-документ:

```
var ajax_response = {};
ajax_response.user = "11";
ajax_response.product = "1111";
ajax_response.count = "1";
ajax_response.session_id = "abc11111";
var xml_response = "<ajax_response>";
for (p in ajax_response)
  if (typeof ajax_response[p] == "string" )
    xml_response += "<" + p + ">" + ajax_response[p] + "</" + p + ">";
xml_response += "</ajax_response>";
```

Как вы можете убедиться, XML-документ из JavaScript-объекта можно сформировать достаточно просто. Но мы сделали это не только просто, но и достаточно грубо. Давайте покритикуем данный фрагмент кода. Основной недостаток заключается в том, что мы не заменили недопустимые в текстовых данных XML-документов символы `&` и `<` на их представление в виде мнемоник `&` и `<`. Это можно сделать при помощи регулярных выражений и метода `replace()` объекта `String`:

```
xml_response += "<" + p + ">"
    + ajax_response[p].replace(/&/g, "&amp;").replace(/</g, "&lt;")
    + "</" + p + ">";
```

В качестве альтернативы, можно было бы оформить строку в качестве секции `CDATA`:

```
xml_response += "<" + p + "><![CDATA["
    + ajax_response[p].replace(/]]>/g, "]]&gt;") + "]]></" + p + ">";
```

Следующим недостатком нашего кода является то, что обрабатываются только свойства объекта типа `String`. Объект может иметь более сложную структуру и содержать в свойствах ссылки на другой объект:

```
var ajax_response = {
  session_id: "abc123456";
  obj_ref: {
    user = "11",
    product = "1111",
    count = "1"
  }
};
```

Для такого объекта должен быть сформирован такой XML-документ:

```
<ajax_response>
  <session_id>abc123456</session_id>
  <obj_ref>
    <user>11</user>
    <product>1111</product>
    <count>1</count>
  </obj_ref>
</ajax_response>
```

То есть для свойств объектов, содержащих ссылку на объект, построение XML-документа должно проводиться рекурсивно для каждого такого свойства. И это порождает дополнительную проблему. Объект может содержать циклические ссылки, и рекурсия может никогда не завершиться, если для этого не предусмотреть анализ посещенных объектов. Кроме того, если вы создаете пользовательский объект, то можете обнаружить, что он имеет часть свойств и методов, которые вы даже нигде не определяли. В чем тут дело? Некоторые свойства и методы будут унаследованы от базового объекта `Object`. Имена методов объекта `Object` хорошо известны и могут быть проанализированы в коде. Но некоторые профессиональные библиотеки JavaScript расширяют функциональность встроенного типа `Object`. Так что, перебирая в цикле свойства пользовательского объекта, вы можете обнаружить самые неожиданные свойства. Я не могу вам сказать, правильно ли подгружать новые методы к встроенному типу `Object`. Но правда заключается в том, что практически все библиотеки делают это. Мы можем с легкостью избежать проблем с добавленными библиотеками свойствами, просто создав новый объект типа `Object` и проверяя, не было ли унаследовано свойство от встроенного объекта.

С учетом тех подробностей, которые мы должны учесть, когда преобразуем объект JavaScript в XML-документ, лучше это преобразование перенести в функцию. Код такой функции представлен в файле \bhvajax\xml\dom.js. Функция называется `bhv.dom.object2xml()` и при вызове программного кода имеет сигнатуру `bhv.dom.object2xml(object, name)`.

Из вызывающей программы функция `xml2object()` всегда вызывается с двумя параметрами. При этом третий и четвертый параметры будут иметь тип `undefined`. В первом параметре содержится ссылка на преобразуемый объект, во втором параметре — имя документа (и корневого элемента документа). Третий и четвертый параметры используются при рекурсивном вызове функции. В третьем параметре хранятся ссылки на уже пройденные объекты,

а в четвертом параметре — имена свойств, в которых хранятся посещенные объекты:

```
path.push(object);
pathNames.push(name);
```

Эти объекты имеют тип `Array` и создаются при первом обращении к функции:

```
if (typeof path == "undefined") {
  path = [];
  pathNames = [];
}
```

А при выходе из каждого рекурсивного вызова, последний посещенный объект удаляется из стека:

```
path.pop();
pathNames.pop();
return xmlString;
```

Конечной точкой рекурсивных вызовов является тот момент, когда очередной рекурсивный вызов обращается к типу, для которого оператор `typeof` возвращает значение, отличное от `"object"` и `"function"`:

```
var xmlString += "<" + name + ">";
if (typeof object == "object"){
... // Код сокращен
} else if (typeof object != "function")
  xmlString += (""+object).replace(/&/g, "&amp;").replace(/</g, "&lt;");
xmlString += "</" + name + ">\n";
```

Для свойств типа "объект" производится наиболее сложная обработка, которая выполняет перебор свойств объекта в цикле, и содержит рекурсивный вызов функции для каждого свойства:

```
if (typeof object == "object"){
  for (var p in object)
    if (typeof object[p] != "function" &&
       (! bhv.proto[p] || p == "value"))
      if (bhv.IN(object[p], path))
        xmlString += bhv.dom.object2xml(
          "<![JSON[" + pathNames.join(".") + "." + p + "]]>",
          p, path, pathNames);
      else
        xmlString += bhv.dom.object2xml(object[p], p, path, pathNames);
```

Все свойства объекта перебираются в цикле:

```
for (var p in object)
```

Этот цикл выдаст не только нужные нам свойства, но и унаследованные от встроенного JavaScript типа `Object` свойства и методы. Не хранится ли в свойстве ссылка на функцию-метод и не является ли свойство унаследованным, проверяется в условии:

```
if (typeof object[p] != "function" && (! bhv.proto[p] || p == "value"))
```

Для унаследованных свойств имеется одно исключение. Если свойство имеет имя `value`, оно все равно включается в XML-документ. Далее проверяется наличие в объекте циклической ссылки:

```
if (bhv.IN(object[p], path))
  xmlString += bhv.dom.object2xml(
    "<![JSON[" + pathNames.join(".") + "." + p + "]]>",
    p, path, pathNames);
```

Если ссылка содержит цикл, очередному рекурсивному вызову передается не объект, что привело бы к бесконечной рекурсии, а строковое представление этого объекта в виде строки JSON:

```
"<![JSON[" + pathNames.join(".") + "." + p + "]]>"
```

Строковое значение прервет выполнение рекурсии, в то же время всегда можно будет восстановить ссылку на объект по строке JSON. Для того чтобы можно было автоматизировать разбор объекта JSON, использована нотация, похожая на секцию `CDATA`, но на самом деле является обычными текстовыми данными, поэтому при обработке этой строки, как и положено, символ < будет заменен на мнемонику `<` при выполнении очередного вызова функции в строке кода:

```
xmlString += (""+object).replace(/&/g, "&amp;").replace(/</g, "&lt;");
```

Если вы занимались когда-нибудь преобразованием объектов в XML-документы в языках программирования, отличных от JavaScript, вас может даже удивить, насколько просто это можно реализовать в JavaScript, за которым успела несправедливо закрепиться слава игрушечного языка программирования. Оставайтесь с нами, дальше будет не менее интересно.

4.2.2. Преобразование HTML-форм в XML-документ

Традиционно, в Web-приложениях данные на Web-сервер отсылались при помощи HTML-форм, в которых данные вводились в элементы `INPUT` и `SELECT`. В Ajax-технологии, для ввода данных обычно используются те же элементы `INPUT` и `SELECT`, которые не обязательно включать в тело HTML-

формы. Однако использование HTML-форм может оказаться полезным, если вы захотите отсылать данные на Web-сервер в виде XML-документов. Вот такая HTML-форма:

```
<form name="ajax_request">
  <input type="text" value="11" name="user" id="user">
  <input type="text" value="1111" name="product" id="product">
  <input type="text" value="1" name="count" id="count">
  <input type="hidden" value=" abc11111" name="session_id"
         id="session_id">
<form>
```

может быть легко преобразована в XML-документ:

```
<ajax_request>
  <user>11</user>
  <product>1111</product>
  <count>1</count>
  <session_id>abc11111</session_id>
</ajax_request>
```

После того как мы разработали функцию `object2xml()`, вам уже понятно, как это можно сделать. Кстати, к HTML-форме можно получить доступ как к объекту, и у вас, возможно, появится желание проверить работу нашей функции `object2xml()` на объекте HTML-форма. Во всяком случае, я именно так и поступил. Оказалось, что для работы с объектом "HTML-форма" функцию пришлось еще немного доработать, в частности, взять все вызовы оператора `typeof` в блоки `try-catch`. Это оказалось необходимым, потому что объект "HTML-форма", как и прочие HTML-элементы, не является встроенным объектом JavaScript. В большинстве случаев это можно не учитывать и работать с HTML-элементами как с объектами JavaScript, но в данном конкретном случае выполнение оператора `typeof` над некоторыми свойствами HTML-элементов может вызвать ошибку разрешения доступа на выполнение этого оператора. После дополнения блоками `try-catch` функция оказалась полностью работоспособной. Но получившийся XML-документ оказался слишком громоздким. Дело в том, что HTML-элементы содержат большое количество свойств. Кроме того, они содержат ссылки не только на дочерние элементы, но и на родительские элементы, и на объект HTML-документа. Поэтому в результирующий XML-документ были включены все HTML-элементы документа и все свойства этих элементов. Для преобразования HTML-форм в XML-документ необходимо разработать специальную функцию, чем мы и займемся прямо сейчас.

Доступ к объекту HTML-форма можно получить по имени формы, заданному в элементе `FORM` в атрибуте `name`, или, как к любому HTML-элементу, при помощи метода `document.getElementByID()`:

```
<form name="someFormName" id="someFormId">
</form>
<script type="text/javascript">
  var formObj0 = document.forms.someFormName;
  var formObj1 = document.forms["someFormName"];
  var formObj2 = document.getElementById("someFormId");
</script>
```

Все три переменные ссылаются на один и тот же объект формы. Коллекция `document.forms` содержит ссылки на все HTML-формы документа, которые можно извлечь в двух эквивалентных нотациях (через точку и как к элементу ассоциативного массива). Кроме того, к элементам коллекции `document.forms` можно получить доступ и как к элементам обычного массива по числовым индексам и даже перебрать их в цикле:

```
for (var i; i < document.forms.length; i++)
  alert(document.forms[i].name)
```

В настоящее время рекомендуют использовать элемент `id` вместо элемента `name`, но пока эта рекомендация трудно выполнима, т. к. на использование атрибута `name` ориентировано много существующего кода клиентских и серверных библиотек. Атрибут `id` любого HTML-элемента имеет еще одну практически не используемую особенность. В глобальном пространстве имен JavaScript в некоторых Web-браузерах создается глобальная переменная с тем же именем, которое указано в атрибуте `id`. На самом деле больших удобств это не сулит по той причине, что засорение глобального пространства имен в JavaScript сопряжено с опасностью затереть нужную переменную. Иногда вы даже можете ничего не подозревать, поскольку такая переменная может быть определена в готовой библиотеке, которую вы используете. Тогда все, что остается, — писать унылые посты на форумы по HTML и JavaScript.

Аналогично тому, как объект `document` содержит коллекцию `document.forms`, каждая форма содержит коллекцию `elements`. С учетом этого, создание функции `form2xml()` является задачей практически решенной, и мы представляем это решение в файле \bhvajax\xml\dom.js.

Очевидно, что код нашей функции `form2xml()` значительно проще, чем код функции `object2xml()`, и не содержит рекурсии. Это связано с тем, что формы не могут быть вложенными и все свои элементы содержат в коллекции

`elements`. Мы обработали HTML-элементы двух видов: `INPUT` и `SELECT`. В элементах `INPUT` значения находятся в свойстве `value`. Но для элементов `INPUT` типа `radio` и `checkbox` в формировании XML-документа принимают участие только выбранные элементы, у которых свойство `checked` равно логическому значению `true`. HTML-элемент `SELECT` имеет немного другое строение, и сам включает коллекцию `options`, которая состоит из HTML-элементов `OPTION`. HTML-элементы `OPTION` имеют значение в свойстве `value`, как и элементы `INPUT`, но выбранные элементы `OPTION` имеют свойство `selected`, равное логическому значению `true` (а не `checked`, как `INPUT` типа `radio` и `checkbox`).

Как и всегда для XML-документов, мы заменяем недопустимые в текстовых данных XML-документа символы их мнемониками:

```
replace(/&/g, "&amp;").replace(/</g, "&lt;");
```

Имя XML-тега формируется по такому правилу: если задан атрибут `name` HTML-элемента, то имя тега берется из атрибута `name`. Если атрибут `name` не задан — из атрибута `id`. Если и атрибут `id` не задан, имя XML-тега берется из атрибута `type` HTML-элемента `INPUT`.

Теперь обратим внимание на первые строчки функции:

```
bhv.dom.form2xml = function(form) {
  if (typeof form == "string") {
    var name = form;
    form = document.forms[name] || document.getElementByID(name);
  } else
    var name = form.name || form.id;
```

Для удобства использования, в качестве параметра функции может быть передана ссылка на объект "HTML-форма" либо имя формы, заданное в атрибуте `name`, либо идентификатор формы, заданный в атрибуте `id`. Это достаточно часто применяемый подход, который упрощает использование функции. Вне зависимости от того, имеете ли вы в распоряжении ссылку на HTML-элемент, как объект DOM, или его имя в виде строки типа `String` — вызывается одна и та же функция. Если вы еще не оценили удобство такого подхода, после написания не одной сотни строк кода на JavaScript, преимущества станут для вас более очевидными.

Для того чтобы перейти к тематике следующего раздела, вы должны задать мне один вопрос: на каком основании нами используются такие свойства (коллекции) HTML-элементов, как `document.forms`, `form.elements`, `select.options`? Если такие свойства HTML-элементов, как `id`, `title`, `name`, `class` (которое в JavaScript используется в виде исключения как `className`),

style определены в спецификации HTML, следовало бы предположить, что и для прочих свойств, которые активно используются для программного доступа к HTML-элементам, должна быть спецификация. И такая спецификация создана. Но об этом поговорим в следующем разделе.

4.3. Спецификация Document Object Model Level 1

Технология Ajax базируется на технологии Dynamic HTML (DHTML). Возможности Dynamic HTML были реализованы Internet Explorer 4.0 и стали публично доступны в сентябре 1997 года. Через год появилась спецификация *"Document Object Model (DOM) Level 1 SpecificationVersion 1.0 W3C Recommendation 1 October, 1998"*. На эту спецификацию, как указано в ее тексте, повлиял (influenced) Dynamic HTML, но спецификация не реализует всех возможностей Dynamic HTML. В частности, в ней не определены события.

Спецификация DOM Level 1 должна стать основным документом для программиста, работающего с DOM HTML-документа, который собирается разрабатывать кроссбраузерные приложения, потому что эту спецификацию поддерживают все наиболее распространенные Web-браузеры. Спецификация DOM Level 1 состоит из двух основных разделов: *"DOM Level 1 Core"* и *"DOM Level 1 HTML"* и нескольких приложений. Из приложений к спецификации нам необходимо познакомиться с *"ECMA Script Language Binding"*, в котором определен интерфейс доступа к свойствам и методам DOM из языка программирования JavaScript.

В разделе "DOM Level 1 Core" определен интерфейс доступа к DOM-модели документа, общий для XML и HTML-документов, в частности, такие свойства, как document.documentElement, parentNode, коллекции childNodes, attributes, такие методы, как document.createElement(), document.createTextNode(), getElementsByTagName(), appendChild(), insertBefor(), removeChild(), replaceChild() и др.

В разделе "DOM Level 1 HTML" определено, что HTML-элементы обладают всеми базовыми свойствами и методами из раздела "DOM Level 1 Core" и имеют ряд специфических для HTML свойств и методов. Это, например, такие свойства HTML-документа, как document.forms, document.images, document.links, методы document.getElementById(), document.getElementByName() (метод document.getElementsByTagName() тоже доступен, но специфицирован в разделе "DOM Level 1 Core").

Из краткого и далеко неполного перечня свойств и методов, специфицированных в DOM Level 1, вы уже поняли, что мы с самого начала пользовались

этой спецификацией, хотя явно и не оговаривали это. При разработке кроссбраузерных приложений необходимо строго соблюдать эту спецификацию. Но все же есть одно свойство HTML-элементов, которое очень часто используют Ajax-программисты, но которое имеет корни в Dynamic HTML и не включено в спецификацию DOM Level 1. Это свойство `innerHTML`. Понятно, что свойство `innerHTML` не было включено в спецификацию DOM Level 1, потому что присваивание нового значения этому свойству влечет за собой серьезную работу: разбор фрагмента HTML-документа, создание на основе этого свойства новых HTML-элементов, обновление вида HTML-документа в окне Web-браузера. Но именно поэтому, программисты и любят использовать свойство `innerHTML`, ведь одним присваиванием можно проделать достаточно серьезную работу. Плата за использование этого свойства — нарушение кроссбраузерности приложений и разное поведение Web-браузера в зависимости от того, какие HTML-элементы содержатся в тексте, который присваивается свойству `innerHTML`.

Вам решать, использовать ли свойство `innerHTML` в своих приложениях. Наверное, для самых серьезных ваших разработок лучше пользоваться только средствами, специфицированными в DOM Level 1. Хотя бы потому, что применение свойства `innerHTML` неявно подразумевает, что ответ Web-сервера содержит фрагмент HTML-документа. А этот способ мы уже подвергли критике в *главе 2*. Но вот ряд аналогичных свойств из Dynamic HTML, которые также не включены в спецификацию DOM Level 1 — `innerText`, `outerHTML` и `outerText` — точно не стоит использовать, поскольку их не поддерживают многие из наиболее распространенных версий Web-браузеров (я не называю конкретно — какие, поскольку на момент, когда вы будете читать эти строки, ситуация может измениться).

Спецификация DOM Level 1 имеет прямое отношение и к использованию объекта `XMLHttpRequest`. При рассмотрении этого объекта в *главе 2* мы указывали, что ответ Web-сервера содержится в двух свойствах объекта `XMLHttpRequest`. До сих пор мы использовали только одно из этих свойств — `responseText`. Свойство `responseText` объекта `XMLHttpRequest` содержит ответ Web-сервера в виде строки текста. Как уже было сказано, такая строка может содержать фрагмент разметки HTML-документа, который можно присвоить свойству `innerHTML`, или текст программы JavaScript, который можно выполнить встроенной функцией JavaScript `eval()`. Но наиболее перспективным мы признали использование в качестве ответа Web-сервера XML-документа. В случае если ответ Web-сервера содержит XML-документ, работать с ним при помощи свойства `responseText` будет неудобно.

Рассмотрим пример. Пусть ответ Web-сервера содержит сведения о товаре, их цене и ассортименте с учетом реальных остатков на складе, которые

может использовать, например, Ajax-приложение интернет-магазина (см. файл \bhvajax\xml\ajax_response.xml).

Как вы заметите, в коде содержится уже не фрагмент HTML-документа, который можно присвоить свойству `innerHTML`, а XML-документ. Часть данных в приведенном XML-документе будет отображена в HTML-документе. Это такие данные, как название `name`, цена `price` и цвет `color`. Идентификатор `id` служит для идентификации товара при взаимодействии Web-браузера с Web-сервером и не будет отображаться в окне Web-браузера. Вполне очевидно, что получить доступ к этим свойствам, применяя методы типа String и даже регулярные выражения, достаточно сложно. Для этого и служит свойство объекта `XMLHttpRequest`, в котором содержится ответ Web-сервера в виде объекта, реализующего интерфейс `Document` из спецификации DOM Level 1 — `responseXML`. Для определенности, тип `Document` иногда не совсем строго называют `DOMDocument`.

Давайте сразу рассмотрим простой пример. Как мы можем получить доступ к данным, полученным с Web-сервера в приведенном XML-документе? Создадим в той же папке, \bhvajax\xml\, HTML-документ \bhvajax\xml\ajax_response.html. В этом документе мы получаем в качестве ответа Web-сервера файл ajax_response.xml. Если файл не содержит ошибок с точки зрения спецификации XML, на основании разбора текста этого файла будет создан объект типа `Document`, который будет доступен как свойство `responseXML` объекта `XMLHttpRequest`. В приведенном коде, почти все методы мы уже использовали при работе с HTML-документами в предыдущих главах книги.

Список элементов `product` мы получаем с помощью метода `getElementsByTagName("product")` и сохраняем в переменной `products`. Значение, которое возвращает метод `getElementsByTagName()`, реализует интерфейс `NodeList`, определенный в спецификации DOM Level 1. Этот интерфейс определяет свойство `length` — количество элементов в коллекции `NodeList`, и метод `item()`, при помощи которого можно получить доступ к члену коллекции по номеру от `0` до `letgth - 1`. Вместо доступа к члену коллекции при помощи метода `item()` в JavaScript можно получить доступ по индексу как к элементу массива. Обе строки кода возвращают один и тот же элемент `id`:

```
products[0].getElementsByTagName("id").item(0);
products[0].getElementsByTagName("id")[0];
```

Мы получили вот этот элемент `id` из файла ajax_respponse.xml:

```
<id>11</id>
```

Нас же интересует не элемент `id`, а значение `id`, равное в данном случае 11. Для того чтобы получить это значение, мы должны получить тестовые данные из тела элемента `products[0].getElementsByTagName("id")[0]`. Для этого

следует приложить еще немного усилий. Текстовые данные содержатся в свойстве элемента `firstData`. Но это еще не строка типа `String`, а объект, реализующий интерфейс `CharacterData`, определенный в спецификации DOM Level 1. Объект, реализующий интерфейс `CharacterData`, имеет два свойства: `length` и `data`. В свойстве `data` и содержатся интересующие нас данные:

```
var id = products[0].getElementsByTagName("id")[0].firstChild.data;
```

Свойство `data` содержит значение типа `String`. При этом символы, заданные `&` и другими мнемониками, преобразованы в соответствующие им символы, так что никаких дополнительных преобразований текста производить нет необходимости.

Если вам показалось, что использовать эти методы для доступа к XML-документу достаточно громоздко из JavaScript, то вы не ошиблись. Дело в том, что спецификация DOM Level 1 была разработана с учетом применения не только в таком удобном во всех отношениях языке, как JavaScript.

После первого знакомства с объектами, определенными в спецификации DOM Level 1, рассмотрим более последовательно эту спецификацию.

Наиболее общим объектом, определенным в спецификации DOM Level 1, является `Node`, который обычно переводится на русский язык как *узел*. Этот термин пришел из *теории графов* и в русском языке в теории графов чаще переводится как вершина. DOM Level 1 использует представление XML-документа в виде ориентированного дерева. Требование спецификации XML о соблюдении правильной вложенности тегов элементов, когда и открывающий, и закрывающий теги элемента должны содержаться в теле одного охватывающего элемента, позволяет трактовать охватывающий элемент как родительский элемент (узел, вершину графа), а элементы, которые содержатся в теле охватывающего тега, как дочерние элементы (узлы, вершины графа). Текстовые данные, которые содержатся в теле элемента, трактуются как дочерний узел этого элемента. Атрибуты XML-элемента также трактуются как дочерние узлы этого элемента. Для ориентированного дерева характерно, что все узлы кроме корневого имеют ровно одного родителя. Корневой узел не имеет родительского узла. На рис. 4.1 представлен XML-документ из файла ajax_response.xml в виде ориентированного дерева.

В модели DOM все объекты либо реализуют интерфейс `Node`, либо являются коллекциями элементов типа `Node`. Свойства и методы интерфейса `Node` реализуют все необходимые операции для работы с представлением документа в виде графа: доступ к родительскому узлу, к коллекции дочерних узлов, к первому и последнему дочернем узлам коллекции, добавление нового дочернего узла в конец коллекции или в определенное место коллекции, замена и удаление существующего узла коллекции дочерних узлов новым узлом.

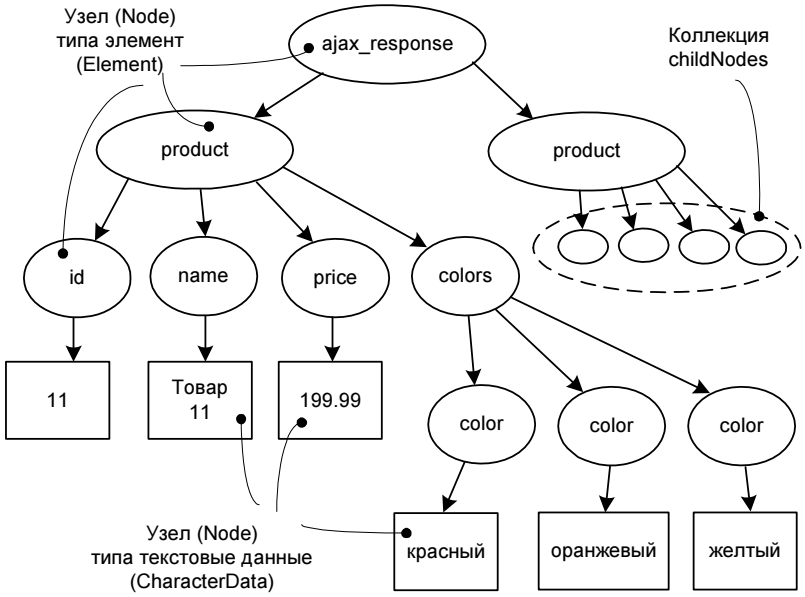


Рис. 4.1. Графическое представление XML-документа

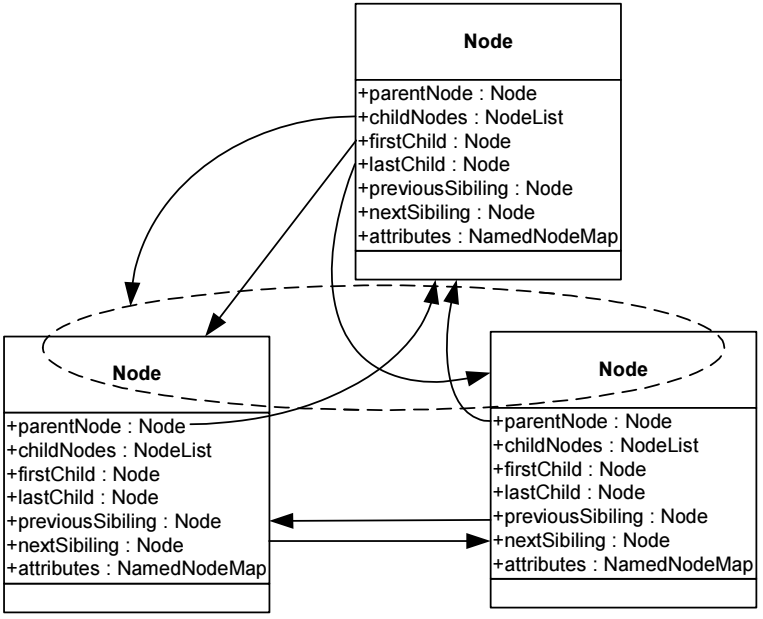


Рис. 4.2. Связи между объектами типа `Node`

В табл. 4.2 представлены все свойства и методы, реализованные в интерфейсе `Node`, а на рис. 4.2 показаны связи между свойствами объектов `Node`.

Таблица 4.2. *Интерфейс* `Node`*, определенный в DOM Level 1*

Имя свойства или метода	Тип значения	Краткое описание
`nodeName`	`String`	Имя узла
`nodeValue`	`String`	Значение узла
`nodeType`	`short (number)`	Тип узла (числовой)
`parentNode`	`Node`	Родительский узел
`childNodes`	`NodeList`	Коллекция дочерних узлов
`firstChild`	`Node`	Первый дочерний узел
`lastChild`	`Node`	Последний дочерний узел
`previousSibiling`	`Node`	Предшествующий смежный узел (в коллекции дочерних узлов)
`nextSibiling`	`Node`	Последующий смежный узел (в коллекции дочерних узлов)
`Attributes`	`NamedNodeMap`	Коллекция атрибутов
`ownerDocument`	`Document`	Документ
`insertBefore(newChild, refChild)`	`Node`	Вставить новый дочерний узел (`newChild`) перед существующим дочерним узлом (`refChild`)
`replaceChild(newChild, oldChild)`	`Node`	Заменить новым дочерним узлом (`newChild`) существующий дочерний узел (`oldChild`)
`removeChild(oldChild)`	`Node`	Удалить существующий дочерний узел
`appendChild(newChild)`	`Node`	Добавить новый дочерний узел в конец коллекции дочерних узлов
`hasChildNodes()`	`Boolean`	Проверяет наличие у текущего узла дочерних узлов
`cloneNode(deep)`	`Node`	Создает клон (копию) узла, параметр `deep` типа `Boolean` указывает, должно ли быть клонирование глубоким

Node — это абстрактный базовый тип (интерфейс). В документе будут присутствовать конкретные подтипы, реализующие интерфейс Node. С некоторыми из них мы уже познакомились — это Element (элемент), Attr (атрибут), CharacterData (текстовые данные), Text (текст), CDATASection (CDATA-секция). Для того чтобы явно указать, какой тип имеет узел, в интерфейсе предусмотрен целочисленный атрибут nodeType. В листинге 4.1 приведены все возможные значения атрибута nodeType.

Листинг 4.1. Значение nodeType интерфейса Node в соответствии с DOM Level 1

```
bhv.dom.ELEMENT_NODE            = 1;
bhv.dom.ATTRIBUTE_NODE          = 2;
bhv.dom.TEXT_NODE               = 3;
bhv.dom.CDATA_SECTION_NODE      = 4;
bhv.dom.ENTITY_REFERENCE_NODE   = 5;
bhv.dom.ENTITY_NODE             = 6;
bhv.dom.PROCESSING_INSTRUCTION_NODE = 7;
bhv.dom.COMMENT_NODE            = 8;
bhv.dom.DOCUMENT_NODE           = 9;
bhv.dom.DOCUMENT_TYPE_NODE      = 10;
bhv.dom.DOCUMENT_FRAGMENT_NODE  = 11;
bhv.dom.NOTATION_NODE           = 12;
```

Вы видите, что узлами могут быть не только элементы, атрибуты и текстовые данные. Мы не будем работать в данной книге с узлами типа Entity, EntityReference, ProcessingInstruction, Notation. В нашем контексте их возможности не будут необходимы. Вы также обратили внимание, что в списке нет узла типа CharacterData (текстовые данные). Как и Node, CharecterData является абстрактным базовым типом для двух производных от CharacterData типов: Text (текст) и CDATASection.

Узлы типа Element представляют собой элементы XML-документа. Узлы типа Element имеют все свойства и методы из интерфейса Node. Давайте вспомним, что XML-элементы отличает наличие у них имени тега и атрибутов. В соответствии с этим, свойства и методы для узлов типа Element будут дополнены свойством tagName, очень часто используемым методом getElementsByTagName(), и методами для работы с атрибутами (табл. 4.3).

Таблица 4.3. *Интерфейс* `Element`*, определенный в DOM Level 1*

Имя свойства или метода	Тип значения	Краткое описание
`tagName`	`String`	Имя тега элемента
`getAttribute(name)`	`String`	Получить строковое значение атрибута по имени атрибута (параметр типа `String`)
`setAttribute(name, value)`	`Void`	Изменить значение атрибута или создать новый атрибут (параметры типа `String`)
`removeAttribute(name)`	`Void`	Удалить атрибут с заданным именем (параметр типа `String`)
`getAttributeNode(name)`	`Attr`	Получить атрибут в виде объекта `Attr` по имени (параметр типа `String`)
`setAttributeNode(newAttr)`	`Attr`	Задать новое значение атрибута или создать новый атрибут (параметр типа `Attr`)
`removeAttributeNode(oldAttr)`	`Attr`	Удалить заданный атрибут (параметр типа `Attr`)
`getElementsByTagName(name)`	`NodeList`	Получить список всех потомков с заданным именем тега (параметр типа `String`)
`normalize()`	`Void`	Нормализовать представление дочерних узлов, имеющих тип `Text` (объединить смежные узлы типа `Text`)

В определении интерфейсов `Node` и `Element` мы использовали типы данных `NodeList`, `NamedNodeMap` и `Attr`. Рассмотрим работу с ними.

Тип `NodeList` имеют коллекция `childNodes` и значение, которое возвращает метод `getElementsByTagName()`. Доступ к членам коллекции `NodeList` похож на доступ к элементам массива. В интерфейсе `NodeList` определены:

- свойство `length` типа `int` (`number`) — количество членов (узлов) коллекции;
- метод `item(index)`, который возвращает член (узел) типа `Node` коллекции `NodeList` по заданному индексу типа `unsigned long` (`number`).

Чаще для доступа к члену коллекции используется не метод `item(index)`, а обращение в стиле массива типа `Array` (при помощи квадратных скобок).

Для точности изложения, я описываю типы индексов как `int` и `unsigned long`, как это определено в спецификации DOM Level 1. Для пользователей JavaScript это все будут параметры числового типа с учетом правил приведения типа JavaScript. (Описание типов индексов как `int` и `unsigned long` реально должны учитывать разработчики Web-браузеров и расширений для Web-браузеров.)

Коллекция `NamedNodeMap` использует для доступа к своим членам и числовые индексы, и строковые имена. В ней также определено свойство `length` и метод `item()`. Кроме того, в ней определен ряд методов для работы с членами коллекции по имени: `getNamedItem(name)`, `removeNamedItem(name)` и `setNamedItem(node)`. У последнего из перечисленных методов параметр имеет тип `Node`, и его имя задается свойством `nodeName`. В качестве членов коллекции `NamedNodeMap` могут выступать любые объекты типа `Node`, но чаще всего это объекты типа `Attr` (атрибуты). Мы не будем пользоваться этими методами, т. к. более удобно применять для доступа к атрибутам методы объекта типа `Element` `getAttribute()`, `setAttribute()` и `removeAttribute()`. Здесь приведено описание методов коллекции `NamedNodeMap` для того, чтобы у вас сложилось более адекватное восприятие DOM-модели документа.

Точно так же, мы не станем напрямую обращаться к объекту `Attr` (атрибут), а будем использовать для работы с атрибутами методы объекта типа `Element`. Но для полноты приведем и интерфейс объекта `Attr`. Объекты `Attr` имеют все свойства и методы, определенные в интерфейсе `Node`, и, кроме того, имеют свойства:

- `name` — имя атрибута типа `String`;
- `value` — значение атрибута типа `String`;
- `specified` — свойство логического типа, которое принимает значение `true`, если атрибут задан явно (специфицирован).

Для того чтобы завершить наш краткий обзор спецификации DOM Level 1, рассмотрим объект типа `Document`. Несмотря на то, что объект типа `Document`, как и большинство объектов из спецификации DOM Level 1, реализует интерфейс `Node`, чаще используются методы, характерные для объекта `Document` и не определенные в интерфейсе `Node`. Это свойство `documentElement`, которое содержит ссылку на корневой элемент XML-документа. Подчеркну, что `Document` и `documentElement` — это разные объекты. Объект `Document` представляет XML-документ в целом, в то время как `documentElement` — это просто элемент, находящийся на вершине иерархии элементов. Объект `Document`

содержит ряд так называемых *фабричных методов*, которые возвращают новые объекты DOM-модели — элементы, атрибуты и др. Наиболее часто мы будем использовать метод `ctreateElement(tagName)`, который создает и возвращает новый узел типа `Element` по заданному в строковом параметре имени тега элемента, и метод `createTextNode(data)`, который создает узел типа `Text`, принимая в качестве параметра строку текста. Еще один метод, который мы будем часто использовать, — `getElementsByTagName()`. Метод с таким именем определен также для объектов типа `Element`. Использование этого метода с объектом `Document` позволяет получить элементы с заданными именем тега во всем документе.

Мы рассмотрели объекты, их свойства и методы, которые будем наиболее часто использовать при работе с XML-документами, представленными в виде DOM-модели в соответствии со спецификацией DOM Level 1. Для того чтобы попрактиковаться в работе с DOM-моделью и получить более полное представление о работе методов, я предлагаю реализовать часть объектов и методов средствами JavaScript. Думаю, что несколько строчек кода могут сказать программисту больше, чем несколько страниц книги. В файле \bhvajax\xml\dom.js я ограничился реализацией подмножества интерфейсов `Document`, `Element`, `Attr` и `Text`.

Как всегда, мы размещаем глобальные имена в объекте пространства имен `bhv`.

Для объектов DOM-модели определяем функции-конструкторы, а функции-методы и некоторые свойства присоединяем к прототипу объекта. Присоединить методы можно непосредственно к существующему объекту прототипа:

```
bhv.dom.Text.prototype.name = "#text";
```

Другой возможностью является использование нового объекта в качестве прототипа, заданного своим литералом (в JSON-нотации):

```
bhv.dom.Element.prototype = {
nodeType: bhv.dom.ELEMENT_NODE
,
getAttrribute: function(name){
  for (var i = 0; i < this.attributes.length; i++)
    if (this.attributes[i].name == name)
      return this.attributes[i].value;
  return undefined;
}
}
```

Вторая форма задания методов и свойств выглядит более компактно, поэтому такая запись очень часто используется. Недостатком такой записи является

то, что существующий объект-прототип заменяется на новый объект-прототип, в частности, из-за этого становится недоступным свойство `constructor` объекта-прототипа, которое содержит ссылку на функцию-конструктор.

Реализация всех объектов в файле \bhvajax\xml\dom.js предусматривает метод `toString()`. Мы используем этот метод для формирования текста XML-документа по его представлению в виде объекта. Возможно, этот код вы будете использовать при отправке запроса к Web-серверу и дополните уже имеющиеся у нас в арсенале функции `object2xml()` и `form2xml()`.

4.4. Использование XML-документов для реализации слайд-шоу

Весь материал по XML, который нам понадобится для работы с книгой, был уже рассмотрен. Но я обещал каждую главу книги подкреплять реально работающим примером. Это служит, по крайней мере, двум целям: читатель получает в свое распоряжение образец работающего кода, и на реальном материале лучше воспринимается и запоминается изучаемый материал.

В качестве такой задачи я выбрал реализацию слайд-шоу средствами Ajax. Поскольку в этой главе и так уже было приведено с избытком программного кода, реализация слайд-шоу будет очень простой. Это будет скорее набросок для компонента, который вы можете реализовать в качестве домашнего задания.

Давайте подумаем, какие проблемы вам придется решить при замене классической Web-страницы на Ajax-приложение. Кроме некоторых технических проблем, может возникнуть еще вопрос о том, что делать с баннерной рекламой. В отличие от классического Web-приложения, когда пользователю время от времени приходится менять текущую страницу, и вы можете предложить ему отведать очередную порцию рекламы, пользователь Ajax-приложения работает с одной и той же страницей. Естественно, придется решать вопрос о том, как будет обновляться реклама на ваших страницах. Решение лежит на поверхности — это можно сделать средствами Ajax.

На самом деле, смена рекламных баннеров на ваших страницах средствами Ajax может принести положительный эффект с самых различных точек зрения, в том числе и с учетом интересов пользователя, рекламодателя, ну и ваших личных интересов. Динамически сменяемый баннер будет лучше замечен пользователем, вы можете оговаривать с рекламодателем время, на которое будет виден баннер, и порядок, в котором будут появляться баннеры. Будет доволен и пользователь, поскольку баннеры будут подгружаться в фоновом режиме после загрузки основного, интересующего пользователя

содержимого Web-страницы, и баннеры будут появляться последовательно в одном секторе страницы, а не занимать большую часть экрана, оставляя минимум пространства для полезной информации.

Мы будем последовательно загружать изображения в HTML-элемент `DIV` для того, чтобы познакомиться с программной частью слайд-шоу. Конечно, о художественной стороне вопроса речь идти не будет. В файле \bhvajax\xml\image_rotator.xml представлен XML-документ, в котором прописаны адреса изображений, которые будут последовательно загружаться в текущую страницу Web-браузера.

В элементе `image_item` XML-документа image_rotator.xml расположены элементы, содержащие текст и URL-адрес изображения. Таких элементов может быть достаточно много. Поэтому загружать картинки мы должны только по мере необходимости перед их выводом в окне браузера. В реальном приложении вы, скорее всего, предусмотрите еще такие атрибуты изображения, как время, которое он будет виден на экране в зависимости от привилегированности рекламодателя, дисциплина очереди, в которой изображение будет появляться на экране.

Вы уже, наверное, догадались, что XML-документ будет загружаться в Web-браузер асинхронным запросом к Web-серверу при помощи разработанной в *главе 2* функции `sendRequest()`. После этого изображения будут загружаться с сервера с заданным интервалом времени. Для этого будет использоваться функция `window.setInterval()`. Для удобства тестирования весь код расположен в файле \bhvajax\xml\image_rotator.html.

Обратим внимание на некоторые особенности кода из файла image_rotator.html.

Функция `sendRequest()` загружает асинхронным запросом файл image_rotator.xml и при окончании загрузки вызывает функцию `loadImage()` со строковым параметром `"rotator"`, в котором передается идентификатор HTML-элемента `DIV`, служащий контейнером для загружаемых имиджей.

Функция `loadImage()` ищет HTML-элемент `DIV` по его идентификатору, заданному в атрибуте `id`, и помещает в него вновь созданный HTML-элемент `IMG`:

```
element = document.getElementById(element);
var imageElement = document.createElement("IMG");
imageElement.style.height = "100px";
element.appendChild(imageElement);
```

Вы, наверное, обратили внимание, что для работы с DOM-моделью HTML-документа используются методы, определенные в спецификации DOM Level 1.

В следующих строках мы переписываем ответ Web-сервера в объект JavaScript, работать с которым, как правило, удобнее, чем с DOM-моделью документа:

```
var imageList = [];
imageList.nextLoad = 0;
var imageTags = this.responseXML.getElementsByTagName("image_item");

for (var i = 0; i < imageTags.length; i++) {
  imageList[i] = {};
  imageList[i].imageText =
    imageTags[i].getElementsByTagName("image_text")[0].firstChild.data;
  imageList[i].imageURL =
    imageTags[i].getElementsByTagName("image_url")[0].firstChild.data;
}
```

Обратите внимание на свойство `imageList.nextLoad`, в котором мы будем запоминать индекс изображения в коллекции и которое мы будем загружать на следующем шаге. Загрузку изображения выполняет функция `innerLoad()`. Эта функция внутренняя по отношению к функции `loadImage()`. Именно то, что эта функция является внутренней, объясняет то, что локальные имена функции `loadImage()` не удаляются "сборщиком мусора" и остаются доступными во время работы приложения. Ссылка на функцию `innerLoad()` передается в функции `window.setInterval()`:

```
innerLoad();
window.setInterval(innerLoad,5000);
```

Но перед этим функция `innerLoad()` вызывается явно, чтобы загрузка первого изображения началась немедленно.

При первом обращении к изображению функция устанавливает функцию-обработчик `onload`, которая будет вызвана при полной загрузке изображения. Поэтому изображение будет видно только после полной загрузки:

```
imageList[currentLoad].imageElement = document.createElement("IMG");
imageList[currentLoad].imageElement.onload = innerLoad;
imageList[currentLoad].imageElement.src = imageL-
ist[currentLoad].imageURL;
```

После полной загрузки изображения будет вызвана эта же самая функция `innerLoad()`, но будет выполнена другая ветка условия:

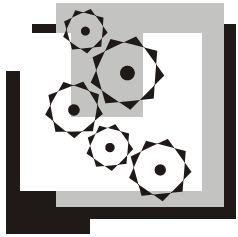
```
if (imageList[currentLoad].imageElement) {
  imageList.nextLoad = (imageList.nextLoad + 1) % imageList.length;
  imageElement.src = imageList[currentLoad].imageElement.src;
}
```

Для прохождения всех индексов коллекции изображений в цикле используется как раз в нужном месте оператор взятия остатка по модулю `%`. Спасибо, что разработчики спецификации JavaScript не забыли предусмотреть такую возможность.

Теперь можно загрузить страничку и насладиться полученным эффектом. Если вы заметили, URL-адрес изображений задан от корневого каталога хоста (начинается с косой черты). Если вы попробуете загрузить Web-страницу без Web-сервера (из файловой системы), то изображение не буден найдено. Подумайте, какой относительный адрес тут должен быть использован.

* * *

Итак, мы рассмотрели еще одну очень важную тему — работу с XML-документами и подмножество спецификации DOM Level 1. Имеющихся сведений достаточно для дальнейшей работы с примерами книги, но, конечно, недостаточно для профессиональной работы с XML-технологией. Если это было ваше первое знакомство с XML — я постарался, чтобы это знакомство было для вас интересным и плодотворным.



Глава 5

Разработка компонента "Полоска меню"

В этой главе мы разработаем компонент *"Полоска меню"*. Он не будет использовать асинхронные запросы к Web-серверу при помощи объекта `XMLHttpRequest`. На самом деле Ajax-приложение — это не приложение, которое обменивается с Web-сервером асинхронными запросами, хотя дословный перевод аббревиатуры Ajax звучит как *асинхронный JavaScript и XML*. Это очень удачное с маркетинговой точки зрения название может немного увести от понимания сути технологии Ajax, которая заключается в создании дружественных пользователю приложений с "богатым" пользовательским интерфейсом. Поэтому большинство Ajax-библиотек поддерживают не только удобную работу с объектом `XMLHttpRequest` и XML-документами, но и обеспечивают разработчиков Web-приложений привычными для настольных (desktop) приложений компонентами пользовательского интерфейса.

Одним из ставших привычным компонентов пользовательского интерфейса стала *полоска меню* (или строка меню), которую можно видеть в верхней части вашего Web-браузера и практически любого современного приложения. Полоска меню удобна, потому что она стала привычной для пользователей и потому, что она удобна. Поэтому разработчики Web-приложений часто используют компонент "Полоска меню". На рис. 5.1 вы можете ознакомиться с внешнем видом компонента "Полоска меню" в окне Web-браузера.

При выборе пункта меню выполняется заданное действие или активизируется *подчиненное выпадающее меню*. При выборе пункта подчиненного выпадающего меню также выполняется заданное действие. Разумеется, для простейших случаев можно прямо в разметке HTML-документа задавать текст меню и выполняемые действия. Но для более сложных случаев, удобнее реализовать компонент "Полоска меню" в соответствии с паттерном "Модель — Вид — Контроллер" (Model — View — Controller). Сокращенно этот паттерн обозначают как MVC. В некоторых источниках вы можете встретить перевод на русский язык названия паттерна MVC как "Модель — Представление — Контроллер" или "Модель — Представление — Поведение". Даже по этим

разночтениям в переводе вы можете косвенно судить о том, что этот паттерн все больше входит в практику, поскольку название "Модель — Вид — Контроллер" можно воспринимать не как перевод, а как транскрипцию англоязычного термина. Только слово "Модель" уже давно используется как научный термин, слово "Контроллер" начало использоваться сравнительно недавно, а слово "Вид" является калькой слова View, которое, может быть, и не соответствует во всех контекстах своему англоязычному собрату, но выявляет глубинные корни наших языков.

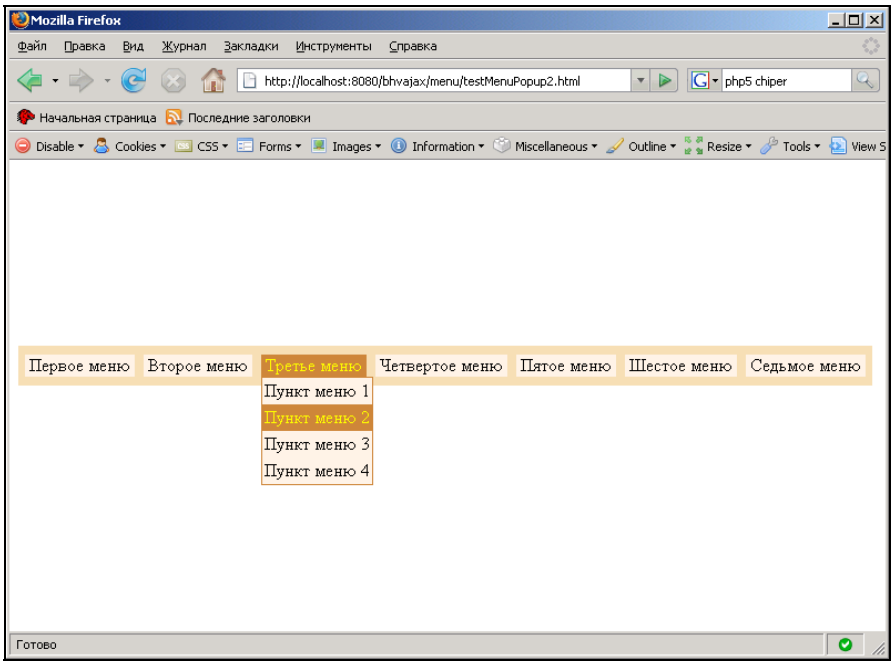


Рис. 5.1. Внешний вид компонента "Полоска меню" в окне Web-браузера

Текст примеров данной главы вы можете найти в папке \bhvajax\menu на прилагаемом к книге компакт-диске.

5.1. Использование паттерна "Модель — Вид — Контроллер" при разработке программ

Если вы еще ничего не читали о паттернах — часто их называют шаблонами проектирования — могу посоветовать обязательно познакомиться с соответствующей литературой. Использование паттернов как метода проектирова-

ния было предложено архитектором Кристофером Александером и адаптировано его последователями для разработки программного обеспечения. Кристофер Александер (родился в 4 октября 1936 года в Вене) получил образование в области математики и архитектуры. Его книга "Notes on the Synthesis of Form" (1964) вызвала интерес у специалистов в области компьютерных наук и была рекомендована Марвином Мински для подготовки студентов в Массачусетском технологическом институте, специализировавшихся в области компьютерных наук и искусственного интеллекта. Идеи Кристофера Александера повлияли на развитие компьютерных технологий в 60—70-х годах XX века. В 1977 году вышла наиболее широко известная книга Кристофера Александера "A Pattern Language". Книга содержит 253 паттерна проектирования. Каждый паттерн описывает проблему, с которой проектировщику приходится сталкиваться снова и снова в практической деятельности, и способ успешного решения проблемы. В настоящее время с паттернами из книги Кристофера Александера "A Pattern Language" вы можете ознакомиться в Интернете.

Паттерны Кристофера Александера описывают проблемы и способы их решения в области архитектуры, хотя сам язык паттернов относится не к конкретной области проектирования, а к методам проектирования и может применяться в любой предметной области в процессе проектирования. Разработчики программного обеспечения адаптировали идеи Кристофера Александера для решения своих специфических проблем. В 1987 году Кент Бэки и Вард Каннигем разработали паттерны для проектирования графического интерфейса для языка программирования Smalltalk. В 1988 году Э. Гамма начал работу над адаптацией языка паттернов в применении к разработке программного обеспечения. В 1991 году выходит книга Д. Коплин (James Coplien) "Advanced C++ Idioms". 21 октября 1994 года Э. Гамма, Р. Хелм, Р. Джонсон и Дж. Влиссидес публикуют книгу "Design Patterns. Elements of Reusable Object-Oriented Software", которая в настоящее время издана на русском языке[1]. В этой книге описаны 23 паттерна, и в *главе 1* дается обзор паттерна "Модель — Вид — Контроллер". Книга приобрела неоспоримый авторитет среди разработчиков программного обеспечения и стала отправной точкой для широкого использования паттернов программистами.

Паттерны (шаблоны проектирования, образцы проектирования) представляют собой описания способов решения проблем, созданные на основе анализа существующих удачных решений. Определяющим для паттернов является то, что они основаны на обобщении реальных решений проблем, а не на спекулятивных построениях, базирующихся на умозрительных предпосылках.

[1] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2001.

Поэтому паттерн нельзя придумать и нельзя улучшить. Можно улучшить только решение конкретной проблемы. Если это решение будет достаточно успешным и получит широкое распространение, такое решение может быть рекомендовано к применению в других аналогичных случаях как паттерн, образец для проектирования.

Примерно в таком же направлении проводили исследование советские и российские ученые Г. С. Альтшуллер, а также А. И. Половинкин, который работал над созданием Межотраслевого Фонда эвристических приемов. В Межотраслевом Фонде эвристических приемов все приемы имеют название и примеры использования, как это имеет место и для паттернов. В то же время есть и существенные отличия. Паттерны Кристофера Александера направлены на решение проблем, и в центре проблем стоит человек в архитектурной среде помещения, здания, комплекса, города. Межотраслевой Фонд эвристических приемов в большей степени ориентирован не на проблемы, а на каталог средств, которыми располагает проектировщик для решения проблем. И решаемые проблемы в большей степени ориентированы не на человека в технической системе, а на собственно технические системы. Такие различия в подходах имеют объективные предпосылки, т. к. Кристофер Александер обобщил паттерны в области архитектуры, которая имеет за плечами опыт тысячелетий. А Межотраслевой Фонд эвристических приемов обобщил опыт в сравнительно молодой области техники и технологии, что нисколько не умаляет значимость и высоту достижений ученых, работавших над созданием Фонда. В настоящее время Межотраслевой Фонд эвристических приемов доступен в Интернете. Я думаю, что, познакомившись с эвристическими приемами из Фонда, вы обогатите свой арсенал решений задач повышенной сложности, с которыми сталкиваетесь в своей ежедневной деятельности, и с благодарностью вспомните авторов. И это послужит поводом для более глубокого знакомства с их идеями и работами, а так же с работами их учеников и последователей.

> ***Замечание***
>
> Начиная с 60-х годов XX века, начали активно разрабатываться методы проектирования. Среди них можно выделить общие и частные методики проектирования, методологии проектирования, методы организации процесса проектирования, методы организации коллектива проектировщиков и методы активизации интуиции проектировщиков. От идеи превратить проектирование в формализованный процесс и реализовать проектирование при помощи искусственного интеллекта, идеи разработчиков методов проектирования мигрировали в сторону более эффективного использования человеческого фактора при проектировании.

Паттерн "Модель — Вид — Контроллер" (далее — MVC) был применен разработчиками пользовательских интерфейсов для языка программирования Smalltalk и прочно вошел в практику разработчиков программного обеспечения.

Приступая к описанию паттерна MVC, я хочу, чтобы у вас сложилось правильное впечатление о роли и месте паттернов при проектировании программного обеспечения.

- Паттерн — это еще не решение проблемы. И мы очень скоро убедимся в этом, когда сравним применение паттерна MVC к настольным (desktop) приложениям и к Web-приложениям. Поэтому использование паттерна — это не завершение, а начало работы над проблемой. Роль паттерна можно сравнить с ролью маяка в бескрайних просторах океана Ajax.
- Используя паттерны для Ajax-приложений, мы попробуем найти наиболее успешные реализации Web-приложений и Web-интерфейсов, которые будут соответствовать общей установке паттерна, а не попытаемся реализовать среду Smalltalk средствами JavaScript и PHP 5.
- Применение паттернов не сужает наше пространство поиска решений проблемы, а расширяет его. На самом деле паттерны можно рассматривать как материал, который должен, с одной стороны, помочь нам сформулировать проблемы, а с другой стороны, активизировать нашу интуицию за счет возникновения ассоциаций при анализе существующих образцовых решений (паттернов).

В соответствии с паттерном MVC, в приложениях, которые взаимодействуют с пользователями, выделяются три составляющие: Модель, Вид и Контроллер. Основная цель использование этого паттерна — сделать эти составляющие приложения (прежде всего Модель) относительно независимыми друг от друга, хотя сделать их полностью независимыми невозможно. На рис. 5.2 приведена наиболее общая схема приложения, разработанного в соответствии с паттерном MVC.

Модель содержит данные и методы доступа к данным, в то время как Вид отвечает за визуальное представление Модели. Роль Контроллера заключается в управлении Моделью и Видом. Для обеспечения независимости Модели от Вида принято наиболее строгое ограничение: Вид не может напрямую обращаться к методам, которые модифицируют Модель. В то же время, Вид может иметь прямой доступ к Модели для получения данных Модели. Существует противоположное мнение, что Вид не может напрямую обращаться к Модели для получения данных и должен запрашивать данные Модели через Контроллер. Я специально привел эти два взаимоисключающие мнения, для того чтобы дать вам полную картину происходящего. Не имеет смысла из чисто умозрительных соображений пытаться поддержать или опровергнуть любое из этих противоречивых суждений. Как я уже говорил, паттерны — это практика, которая нашла свою форму в виде экстракта из тысяч строк программного кода. Более убедительная реализация, успешно примененная

в различных приложениях, и даст ответ на вопрос, какое из двух противоречивых суждений имеет право быть принятым за образец (образцы проектирования — еще одно название паттернов).

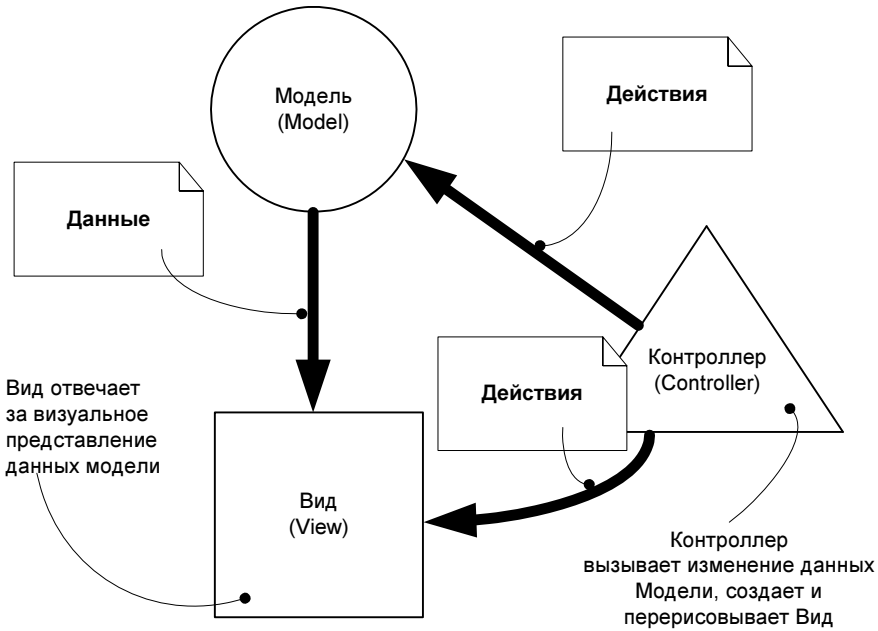


Рис. 5.2. Общая схема использования паттерна MVC

В первоначальном варианте под Моделью понималась не бизнес-логика, а модель пользовательского интерфейса приложения. Например, кнопка, которая имеет состояния "нажата", "не нажата", "доступна", "недоступна", может быть представлена такой Моделью:

```
function ButtonModel (pressed, enabled) {
  this.pressed = pressed;
  this.enabled = enabled;
}
ButtonModel.prototype = {
  setPressed: function(pressed) {
    if (! this.enabled)
      return;
    this.pressed = pressed;
  },
```

```
  setEnabled: function(enabled) {
    this.enabled = enabled;
    if (! this.enabled)
      this.pressed = false;
  }
  isPressed: function() {
    return this.pressed;
  }
  isEnabled: function() {
    return this.enabled;
  }
}
```

Мы определили Модель кнопки, которая может быть нажата, если является доступной. Вы видите, что наша Модель не содержит указаний на то, как она будет выглядеть на экране. Заданы только состояния, которые может иметь кнопка и логика, по которой эти состояния могут изменяться. Например, недоступную кнопку нельзя нажать, и недоступная кнопка автоматически становится не нажатой кнопкой. Вид, который будет определен далее в этой главе, не может вызывать методы `setPressed()` и `setEnabled()`, но может вызывать методы `isPressed()` и `isEnabled()`, которые не изменяют данные или состояние Модели.

Наша модель содержит данные, необходимые для отображения кнопки. При этом Модель не зависит от того, как именно будет эта кнопка прорисована Web-браузером. За это отвечает Вид:

```
function ButtonView(model, container) {
  this.model = model;
  this.span = document.createElement("SPAN");
  container.appendChild(this.span);
}
ButtonView.prototype.paint = function() {
  if (! this.model.isEnebled())
    this.span.className = "disabledButton";
  else if ( this.model.isPressed())
    this.span.className = "pressedButton";
  else
    this.span.className = "unpressedButton";
}
```

Вы видите, что Модель является независимой от Вида, а Вид может отображать Модель на основании данных Модели, полученных при помощи мето-

дов `isPressed()` и `isEnabled()`. Теперь остается разработать Контроллер, который бы вызывал изменения Модели и заставлял Вид перерисовываться в соответствии с новым состоянием Модели. Как правило, код Контроллера получается достаточно сложным. Я приведу упрощенный вариант Контроллера, чтобы легче было рассмотреть функциональность Контроллера и те проблемы, которые возникают в связи с разработкой Контролера средствами JavaScript и DOM-модели документа:

```
function ButtonController(model, view){
  this.model = model;
  this.view = view;
  var my = this;
  this.view.span.onclick = function() {my.viewOnClick()}
}
ButtonController.prototype.viewOnClick() = function() {
  if (! this.model.isEnabled())
    return;
  this.model.setPressed(! this.model.isPressed());
  this.view.paint();
}
```

Как вы можете наблюдать, Контроллер хранит ссылку на Модель и Вид:

```
this.model = model;
this.view = view;
```

Для обработки события `onclick` мы использовали обработку встроенного в Web-браузер события HTML-элемента `onclick`. Это событие привязано к HTML-элементу `SPAN`, поэтому для правильного вызова обработчика события как метода объекта Контроллера необходимо использовать замыкание:

```
var my = this;
this.view.span.onclick = function() {my.viewOnClick()}
```

Как мы уже неоднократно говорили, замыкания усложняют работу "сборщика мусора" и определяются при помощи сравнительно сложных конструкций языка JavaScript. Необходимость использовать замыкания при назначении методов Контроллера в качестве обработчиков событий от мыши и клавиатуры является в JavaScript фактором, усложняющим разработку Контроллера.

Модель является наиболее независимой частью приложения. Как вы можете убедиться, наш вариант Модели вообще "ничего не знает" о конкретной реализации Вида и Контроллера. Вид должен знать только об интерфейсе доступа к данным Модели. Но в реальных приложениях картина вырисовывается более сложная.

Основным вопросом является следующий вопрос: когда необходимо перерисовывать Вид? В приведенном примере Контроллер перерисовывает Вид сразу после вызова методов, изменяющих Модель. Но это простейший случай. Задача существенно усложняется тем, что к одной Модели может быть присоединено несколько Контроллеров или Видов, и изменение Модели может происходить асинхронно.

Наиболее примитивным способом, который применяется для обновления Вида с учетом изменившейся Модели — это периодический запрос данных Модели, который осуществляется по таймеру. Если вы работали с настольными базами данных, то должны быть знакомы с таким решением. Не всегда такой запрос по таймеру реализован достаточно корректно. Поэтому в некоторых конкретных реализациях перерисовка данных сопровождается миганием экрана, блокировкой пользовательского интерфейса, а в отдельных, наименее удачных случаях, сбросом текущего положения скроллинга и переходом на первую запись таблицы базы данных. Такое решение может приводить к существенному повышению сетевого трафика, если повторно запрашиваются все данные большой таблицы (которая может содержать миллионы строк), хотя на экране в данный момент всегда будет отражено не более 10—20 строк таблицы. Это примеры неудачного решения. Но существуют и достаточно хорошие решения, когда периодически запрашивается только отображаемая часть данных, и перерисовываются только изменившиеся значения. В ряде случаев, хорошо продуманный периодический запрос данных Модели по таймеру может быть признан обоснованным решением.

С учетом того, что Модель может иметь несколько Видов и Контроллеров и обновляться асинхронно, только Модель может извещать Контроллер и Вид о своем изменении. При этом Модель уже становится зависимой от реализации Контроллера. Для того чтобы Контроллер мог узнать о том, что данные Модели изменились и необходимо перерисовать Вид, Модель должна послать сообщение Контроллеру о своем изменении. Для этого Модель должна обеспечивать рассылку сообщений о своем изменении всем объектам, которым необходимо отслеживать изменения Модели. Существует соответствующий паттерн, который называют Обозреватель, Слушатель или Подписка/Рассылка. Вот вариант реализации паттерна Слушатель на JavaScript:

```
function AbstractModel() {
  this.listeners = [];
}
AbstractModel.prototype.addListener = function(listener) {
  this.listeners.push(listener)
}
```

```
AbstractModel.prototype.stateChanged = function(event) {
  for (var i = 0; i < this.listeners.length; i++)
    this.listeners[i].stateChanged(event)
}

function AbstractController(model, view){
  this.model = model;
  model.addListener(this);
  this.view = view;
}
AbstractController.prototype.stateChanged = function(event)
  this.view.paint();
}
```

Контроллер может взаимодействовать асинхронно не только с Моделью, но и с Видом. Как правило, асинхронное взаимодействие Контроллера и Вида реализуют в настольных (desktop) приложениях для того, чтобы пользовательский интерфейс не блокировался на время выполнения операций Контроллера и Модели. Поэтому Вид и Котроллер также могут взаимодействовать в соответствии с паттерном Слушатель. На рис. 5.3 приведена общая схема взаимодействия MVC в настольных приложениях. Контроллер прослушивает события, которые поступают от Модели и пользовательского интерфейса. Конкретная реализация этой схемы в классе `JButton` приведена на рис. 5.4.

Класс `JButton` реализован в соответствии с паттерном MVC. Как вы можете убедиться, реально работающая реализация сложнее общей схемы. Она решает целый ряд задач, среди которых есть и такие, как обратная совместимость с раннее разработанным программным кодом, кросс-платформенность и эффективная многопоточная работа. Модель Кнопки `JButton` — `DefaultButtonModel` — имеет модифицирующий метод `setPressed()` и метод-наблюдатель (по классификации Барбары Лисков и Джона Гатега) `isPressed()`. Модель отправляет сообщение о своем изменении `stateChanged()` объекту `JButton`. В качестве Вида выступает объект `BaseButtonUI`. В классе `JButton` реализована концепция сменных видов для прорисовки пользовательских интерфейсов в различных стилевых оформлениях. В связи с этим, объект класса `BaseButonUI`, как конкретная реализация вида, создает свой Контроллер — `BaseButtonListener`, который прослушивает события, поступающие из объекта `JButton`. Событие от пользовательского интерфейса при нажатии кнопки мыши в области Кнопки поступает к объекту `JButton`. Тонкости реализации обработки поступивших событий теряются в глубине иерархии классов и в данном случае нас не интересуют.

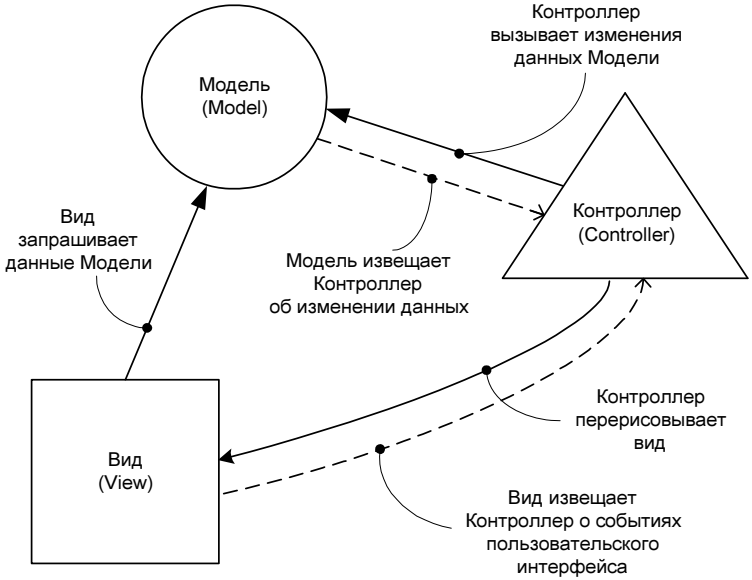


Рис. 5.3. Схема использования паттерна MVC в настольных приложениях

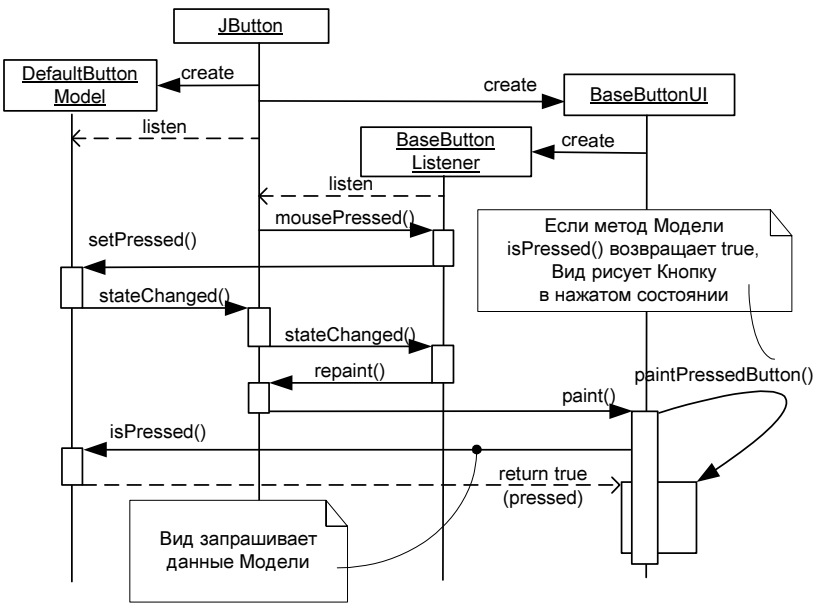


Рис. 5.4. Диаграмма последовательностей работы класса `JButton`

Событие от объекта JButton mousePressed() поступает к объекту BaseButtonListener, который модифицирует Модель, вызывая метод setPressed(). Изменившаяся Модель рассылает сообщение stateChanged() всем зарегистрированным слушателям, среди которых объект класса JButton. Этот объект пересылает сообщение stateChanged() объекту BaseButtonListener. В данном случае этот объект не производит никаких дополнительных действий по обработке сообщения, а просто вызывает метод repaint() объекта JButton. Объект JButton вызывает метод paint() объекта BaseButtonUI. Пересылка сообщений от класса JButton к BaseButtonListener может показаться лишним холостым ходом, но это не так. Во-первых, могут существовать реализации Видов, отличных от BaseButtonUI, которые потребуют в методе stateChanged() дополнительную обработку, а во-вторых, на этапе разработки, не зная всех подробностей реализации, сложно предвидеть, что при реализации BaseButtonUI не потребуется дополнительная обработка события ctateChanged() объектом BaseButtonListener и будет достаточно стандартной обработки, предусмотренной в классе JButton. Метод paint() объекта Вида BaseButtonUI обращается непосредственно к Модели и запрашивает состояние при помощи метода isPressed(). Только получив от Модели текущее состояние Кнопки isPressed(), равное логическому значению true, Вид перерисовывает Кнопку в нажатом состоянии методом paintPressedButton().

В разобранном примере мы говорили о классе DefaultButtonModel как о Модели и BaseButtonUI как о Виде. Что касается Контроллера, такого однозначного соответствия не существует. Часть функциональности Контроллера реализована в классе JButton, часть функциональности делегирована классу BaseButtonListener (например, вызов метода setPressed() Модели). Но класс JButton не является только Контроллером. JButton — это класс, который и является Кнопкой, реализованной в соответствии с паттерном MVC.

Если первоначально под Моделью подразумевалась исключительно модель пользовательского интерфейса, то в настоящее время все чаще Модель ассоциируют с моделью приложения или с бизнес-логикой приложения. Основной принцип и в этом случае остается тот же — отделение данных от их визуального представления.

Паттерн MVC не представляет готового, законченного решения на все случаи жизни. Если бы все вопросы были решены, наверное, программное обеспечение отныне разрабатывалось без проблем и только самого высокого качества. Для того чтобы удостовериться, насколько различными могут быть реализации паттерна, рассмотрим принятые многими Web-программистами архитектуры *Model I* и *Model II*, которые основаны на паттерне MVC. Несмотря

на то, что Model I и Model II отличаются в своей реализации, и Model II считается боле перспективной для использования в Web-приложениях, мы не будем подробно останавливаться на различии этих моделей и подчеркнем то общее, что их связывает и имеет корни в паттерне MVC. На рис. 5.5 приведена схема, по которой работают Web-приложения Model I и Model II.

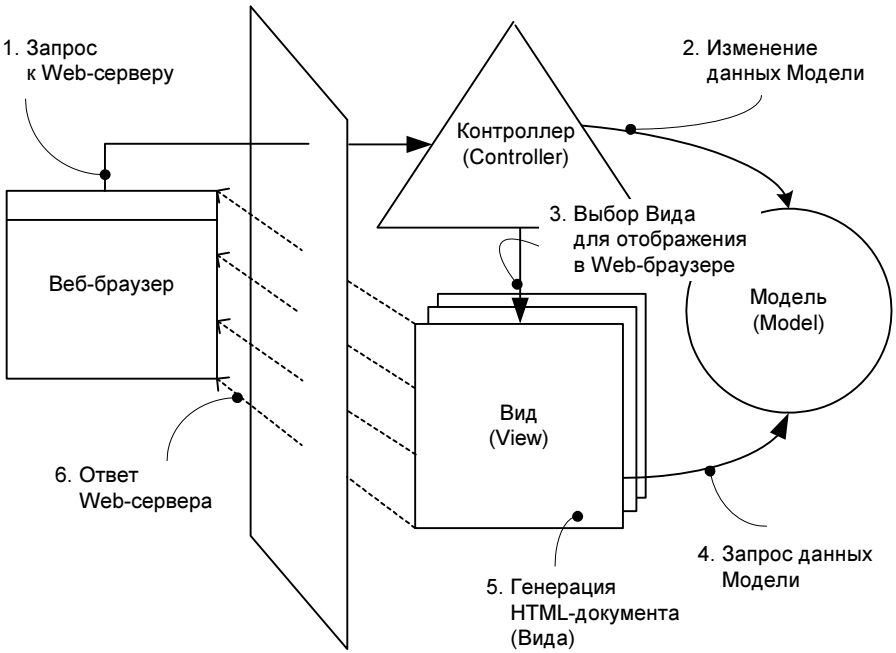


Рис. 5.5. Реализация паттерна MVC для Web-приложений Model I и Model II

Запрос от браузера к Web-серверу поступает на Контроллер, который преобразует запрос в бизнес-действие. Бизнес-действие передается Модели, которая представляет собой бизнес-логику Web-приложения и вызывает изменение состояния и данных Модели. В частности, такое бизнес-действие может быть отклонено, и состояние Модели не изменится. После изменения состояния и данных Модели, в соответствии с логикой работы приложения, выбирается Вид для отправки Web-браузеру. Выбранный Вид запрашивает данные Модели и генерирует HTML-документ, который отображается Web-браузером после получения ответа Web-сервера. Архитектура приложения Model II отличается от Model I более строгим разделением кода между уровнями приложений. В частности, в Model II все запросы обрабатываются единым Контроллером.

Как вы можете убедиться, реализация паттерна MVC в настольных и в Web-приложениях отличается. Отличаются и уровни абстракции, к которым применяется этот паттерн.

5.2. Использование паттерна MVC в Ajax-приложениях

Рассмотрим особенности Ajax-приложений с точки зрения применения паттерна MVC. Модель может быть представлена в Ajax-приложениях как традиционной для Web-приложений серверной Моделью, так и, по меньшей мере, еще двумя способами. Во-первых, в качестве Модели может выступать клиентская Модель, реализованная средствами JavaScript. Во-вторых, можно использовать реализованную в соответствии с паттерном MVC DOM-модель документа. На рис. 5.6 представлены три варианта хранения Модели, которые могут быть использованы при разработке Ajax-приложений. Рассмотрим конкретно каждый из этих вариантов хранения Моделей.

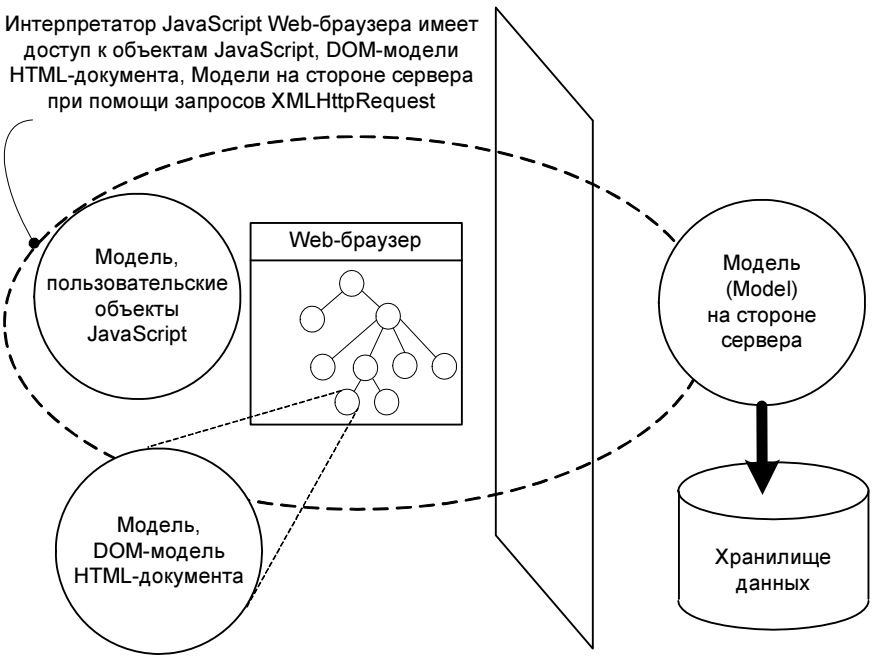


Рис. 5.6. Варианты хранения Модели

Варианты хранения Модели в Ajax-приложениях будем рассматривать на конкретном примере компонента пользовательского интерфейса Кнопка. Давайте создадим Кнопку, представленную в Web-браузере при помощи HTML-элемента `SPAN`, которая может иметь два состояния: нажатая и не нажатая. Визуальное представление состояния Кнопки будет реализовано при помощи двух стилевых классов:

```
span.button {
  padding: 4px;
  background-color: yellow;
  border-style: outset;
  border-width: 3px
}

span.buttonPressed {
  padding: 5px;
  background-color: yellow;
  border-style: inset;
  border-width: 2px
}
```

Для начала рассмотрим серверный вариант реализация Модели Кнопки (разумеется, пример этот является условным, и Модель Кнопки вы, скорее всего, будете реализовывать средствами JavaScript на стороне Web-браузера). Скрипт PHP 5 в файле button.php реализует Модель, Вид и Контроллер на стороне Web-сервера.

Здесь Модель реализована классом `ButtonModel`. Эта реализация достаточно традиционна и не вызывает вопросов. Объект Модели создается при первом обращении к Web-ресурсу и сохраняется в данных сессии:

```
session_start();
if (! isset($_SESSION['button'])) {
  $_SESSION['button'] = new ButtonModel("");
```

Класс `ButtonView` реализует Вид. В данном случае используется статический метод, которому в качестве параметра передается объект Модели:

```
ButtonView::renderModel($_SESSION['button']);
```

Метод `renderModel()` выводит в генерируемый динамически HTML-документ разметку в соответствии с текущим состоянием Модели, которое запрашивается методом `isPressed()`:

```
static function renderModel($model){
  echo '<span id="button" class=';
```

```
    if ($model->isPressed()) {
      echo "'buttonPressed'";
    } else {
      echo "'button'";
    }
  echo "onclick=\"javascript:document.location.href='button.php'\">"
      ."Кнопка</span>";
}
```

В качестве Контроллера в файле button.php выступает собственно код скрипта вне определения классов:

```
session_start();
if (! isset($_SESSION['button'])) {
  $_SESSION['button'] = new ButtonModel("");
} else {
  $_SESSION['button']->togglePressed();
}

ButtonView::renderModel($_SESSION['button']);
```

Приведенное решение показывает общую схему прохождения запроса. Конечно, в качестве Модели Web-приложений чаще выступает бизнес-логика, а Вид и Контроллер реализуют при помощи библиотек JavaScript и PHP 5, которые бывают достаточно сложны. Я же пытался подобрать пример, который мог бы в нескольких строчках кода показать суть происходящего. Конечно, это не тот код, который будет использоваться непосредственно при разработке приложений.

Теперь рассмотрим, как может быть представлена Модель средствами Web-браузера. Web-браузер может использовать для реализации Модели DOM-модель HTML-документа или пользовательские объекты JavaScript. Наиболее просто и удобно для простых Моделей использовать хранение Модели непосредственно в DOM-модели HTML-документа, которая сама построена с учетом архитектуры MVC. Любой HTML-элемент имеет ряд свойств, определенных спецификацией HTML, и методов, определенных в спецификации DOM Level 1, а также в других спецификациях. В спецификации HTML определены события, на которые должны реагировать HTML-элементы. Помимо этого, с HTML-элементом можно производить почти все действия, которые допускается производить пользовательским объектом языка JavaScript. В частности, можно присоединять к HTML-элементу произвольные свойства и методы:

```
<span id="button1">Кнопка</span>
<script type="text/javascript">
```

```
var button = document.getElementById("button1");
button.className = "button";
button.pressed = false;
button.onclick = togglePressed;
</script>
```

В приведенном примере к HTML-элементу `SPAN` было присоединено новое свойство `pressed`, которому присвоено логическое значение `false`. Подобный прием мы уже неоднократно использовали, только не акцентировали на этом особое внимание.

У вас может возникнуть вопрос: можно ли использовать определение неспецифицированных свойств непосредственно в разметке HTML-документа:

```
<span id="button1" pressed="false">Кнопка</span>
```

Я бы не рекомендовал делать так без особой необходимости, потому что это противоречит спецификации HTML и схемам определения HTML-документа. Но на самом деле такой вариант может использоваться и даже используется некоторыми популярными библиотеками. В таком варианте доступ к свойствам кроссбраузерно можно получить при помощи метода `getAttribute()`:

```
<span id="button1" pressed="false">Кнопка</span>
<script type="text/javascript">
var button = document.getElementById("button1");
alert(button.getAttribute("pressed"));
</script>
```

Значение атрибута, заданного непосредственно в HTML-разметке, может быть только строковым. Некоторые версии Web-браузеров игнорируют неспецифицированные атрибуты, которые заданы непосредственно в HTML-разметке, хотя наиболее распространенные версии Web-браузеров поддерживают такую функциональность. Другая часть Web-браузеров допускает использовать неспецифицированные атрибуты, заданные непосредственно в HTML-разметке, как обычные свойства:

```
<span id="button1" pressed-"false">Кнопка</span>
<script type="text/javascript">
var button = document.getElementById("button1");
alert(button.pressed);
</script>
```

Но такой доступ к неспецифицированным атрибутам лучше исключить.

Реализацию Модели непосредственно при помощи DOM-модели HTML-документа рационально применять в том случае, когда между объектом Модели и элементом DOM можно установить однозначное соответствие.

Например, Кнопку можно полностью ассоциировать с HTML-элементом SPAN. При этом доступ к данным модели из обработчиков событий можно получить непосредственно при помощи неявного параметра this (текущий объект). В файле button.html приведен код компонента Кнопка, реализованный при помощи хранения Модели в DOM документа.

Мы пользуемся уже хорошо известным нам приемом, и присоединяем поведение к существующему HTML-элементу:

```
function buttonAddBehivior(button) {
  if (typeof button == "string")
    button = document.getElementById(button);
  button.className = "button";
  button.pressed = false;
  button.onclick = togglePressed;
}
```

В параметре методу buttonAddBehivior() может быть передана или строка, содержащая идентификатор id HTML-элемента SPAN, или непосредственно ссылка на этот HTML-элемент. Непосредственное отношение к Модели имеет строка:

```
button.pressed = false;
```

В данном случае модель достаточно проста и может быть сохранена в единственном свойстве. Для более сложных случаев следует исключить доступ непосредственно к свойствам Модели и осуществлять его при помощи методов, модифицирующих и извлекающих данные Модели:

```
function buttonAddBehivior(button) {
  if (typeof button == "string")
    button = document.getElementById(button);
  button.className = "button";
  button.pressed = false;
  button.onclick = togglePressed;
  button.setPressed = setPressed;
  button.isPressed = isPressed;
}
function setPressed(pressed) {
  this.pressed = pressed;
}
function isPressed() {
  return this.pressed;
}
```

В качестве Контроллера в данном примере выступает система обработки событий Web-браузера. В качестве обработчика события `onclick` — функция `togglePressed()`:

```
function togglePressed() {
  this.setPressed(! this.isPressed());
  if (this.isPressed())
    this.className = "buttonPressed";
  else
    this.className = "button";
}
```

И тут, внимание, функция, заданная как обработчик события HTML-элемента, в неявном параметре `this` всегда содержит ссылку на текущий (на котором щелкнули) HTML-элемент. Немного забегая вперед, скажу, что именно это является небольшой проблемой при реализации Модели средствами объектов JavaScript, а не DOM-модели HTML-документа, о чем мы будем говорить далее в этой главе.

Вид реализован средствами CSS, хотя для более сложных случаев следовало бы вынести прорисовку Вида в отдельную функцию, например:

```
function togglePressed() {
  this.setPressed(! this.isPressed());
  renderButton(this, this)
}
function renderButton(model, view)
  if (model.isPressed())
    view.className = "buttonPressed";
  else
    view.className = "button";
}
```

Функции `renderButton()` в качестве фактических параметров передается ссылка на HTML-элемент, в первом параметре в качестве Модели, а во втором параметре в качестве Вида. Конечно, я сделал это для того, чтобы подчеркнуть, что в данном случае используются разные стороны HTML-элемента: как Модели и как Вида:

```
model.isPressed();
view.className;
```

Дело в том, что любой HTML-элемент в Web-браузере является сложным компонентом, который сам построен по принципу "Модель — Вид — Контроллер", и мы используем те или иные части встроенных возможностей

HTML-элемента. В одном случае это будет способность хранить данные в свойствах и обращаться с ними при помощи методов, подобно Модели. В другом случае это будут манипуляции свойствами стиля HTML-документа, которые будут приводить к изменению Вида HTML-элемента. В третьем случае это будет способность HTML-элемента реагировать на события от мыши, клавиатуры, подобно Контроллеру. Но во всех случаях нам приходится использовать ссылку на текущий объект `this`, поскольку явно выделить Модель, Вид и Контроллер из состава HTML-элемента мы не можем. Поэтому программный код может выглядеть так, что можно подумать, что Модель смешивается с Видом и Контроллером.

Если речь идет о более сложных Ajax-компонентах, например Lookup Combobox или редактируемых таблицах данных, то преимущества хранения Модели непосредственно в DOM HTML-документа теряются, поскольку нельзя установить однозначное соответствие между HTML-элементом и Моделью. Для этих случаев лучше представить Модель как пользовательский объект JavaScript. Об этих компонентах мы подробно поговорим в *главах 10* и *11*. Сейчас реализуем с таким подходом сквозной пример главы — Кнопку.

В файле button1.html представлена реализация паттерна MVC при помощи пользовательских объектов JavaScript.

Код конструктора объекта `Button()` существенно изменился, т. к. мы должны сохранить циклические ссылки объекта `Button` на HTML-элемент `SPAN` и наоборот:

```
function Button(span) {
  if (typeof span == "string")
    span = document.getElementById(span);
  span.model = this;
  this.span = span;
}
```

Наличие циклической ссылки является проблемой для "сборщика мусора" Web-браузера. Тем более, что HTML-элементы, которые удалены из DOM-модели HTML-документа, зачастую не удаляются из памяти, и не имея циклических ссылок. Поэтому циклическую ссылку при удалении компонента необходимо обязательно разорвать явно:

```
Button.prototype.destroy = function() {
  this.span.button = undefined;
  delete this.span.button;
  this.span.onclick = undefined;
  this.span = undefined;
```

```
  delete this.span;
}
```

Несмотря на многообещающее название, код функции `destroy()`, разумеется, необходимо будет вызывать явно. В нашем примере элемент освобождается после определенного количества щелчков мышью по Кнопке.

В качестве Контроллера выступает функция `toggPepressed()`, которая назначена в качестве обработчика события `onclick` HTML-элемента `SPAN`:

```
function togglePressed() {
  this.model.togglePressed();
  renderButton(this);
}
```

В качестве Вида выступает HTML-элемент `SPAN` и функция `renderButton()`:

```
function renderButton(span) {
  if (span.model.pressed)
    span.className = "buttonPressed";
  else
    span.className = "button";
}
```

Вы, наверное, обратили внимание, что код получился более сложный, чем при хранении Модели непосредственно в DOM HTML-документа. Но это справедливо только для самых простых Ajax-компонентов. Для более сложных компонентов реализация модели в пользовательском объекте JavaScript позволяет получить более простые конструкции. В качестве примера я приведу фрагмент программного кода Модели компонента Lookup Combobox, с которым мы будем детально знакомиться в *главе 9*:

```
bhv.Combobox.ComboboxData.prototype = {
getDisplayValue: function(rowIndex) {
if (rowIndex >= this.currentCount)
    return false;
return this.data[rowIndex][1];
},
getSearchValue: function(rowIndex) {
if (rowIndex >= this.currentCount)
    return false;
return this.data[rowIndex][2];
},
getKeyValue: function(rowIndex) {
if (rowIndex >= this.currentCount)
```

```
    return false;
return this.data[rowIndex][0];
},
getKeyIndex: function(rowKey) {
for (var i = 0; i < this.currentCount; i++)
    if (this.getKeyValue(i) == rowKey)
        return i
return -1;
},
getCurrentDisplayValue: function() {
return this.data[this.currentIndex][1];
}
,////////////////////////////////////////////////////////////
getCurrentKey: function() {
return this.data[this.currentIndex][0];
},
getCurrentSearchValue: function() {
return this.data[this.currentIndex][2];
}
}
```

Сложно представить, насколько громоздким мог бы оказаться код, в котором Модель компонента Lookup Combobox была бы реализована средствами DOM HTML-документа.

5.3. Создание компонента "Полоска меню" средствами HTML-разметки

Теперь давайте перейдем к созданию компонента "Полоска меню". Наш компонент будет содержать горизонтально расположенные пункты меню. Их можно реализовать HTML-элементами `DIV`, располагая их в теле контейнера, в качестве которого может выступать также HTML-элемент `DIV`. По щелчку кнопкой мыши на элементе `DIV` будет раскрываться подчиненное выпадающее меню или выполняться заданное действие. Очевидно, что можно установить однозначное соответствие между элементом `DIV` и пунктом меню. Поэтому можно хранить данные Модели непосредственно в DOM HTML-документа. Это сделает код более простым.

Компонент "Полоска меню" — это тот компонент, который используется практически на каждой Web-странице, и вопрос о том, как его реализовать, — один из наиболее распространенных на интернет-форумах по Web-

программированию. Поэтому мне хотелось сделать этот компонент достаточно простым для вашего использования. Как всегда, начнем с разработки стилистического оформления меню. Стилевое оформление компонента "Полоска меню, приведено в файле menu1.css.

Стилевой класс `div.menu` предназначен для HTML-элемента `DIV`, который является контейнером для всего компонента. В частности, установка свойства стиля `padding` позволяет отобразить вложенные элементы `DIV` в полном объеме. Стилевые классы `div.selected` и `div.other` предназначены для отображения пунктов меню элементами `DIV`. Визуальное отличие выбранного пункта меню реализовано за счет рамки `border-style: inset`. Появление и скрытие подчиненных выпадающих меню реализовано за счет контекстнозависимых стилевых классов:

```
div.selected  div.popup {
  position: absolute;
  visibility: visible;
  background-color: wheat;
  color: blue;
  padding: 0px
}

div.other  div.popup {
  position: absolute;
  top: 0px;
  left: 0px;
  visibility: hidden
}
```

При изменении стиля основного пункта меню с `div.other` на `div.selected` пункт меню отображается, а в противном случае — скрывается, за что отвечает свойство стиля `visibility`. Пункты подчиненного выпадающего меню реализованы при помощи HTML-элементов `DIV`. Это позволяет отображать пункты подчиненного выпадающего меню один под другим, как это и ожидает пользователь. Для них разработаны стилевые классы `div.itemOther` и `div.itemSelected`, которые визуально похожи на `div.other` и `div.selected` и отличаются параметрами границ `margin`, которые позволяют этим пунктам выглядеть соразмерно в панели Web-браузера.

Первый вариант компонента "Полоска меню" реализован наиболее простым способом при помощи HTML-разметки. Код компонента можно найти на прилагаемом диске в HTML-документе testMenuPopup1.html.

В данном случае максимально использованы возможности, предоставляемые стандартными средствами DOM-модели HTML-документа. На самом деле, можно было бы еще более полно использовать стандарт CSS2.1 и обойтись без JavaScript-кода для изменения стилевого оформления выделенных пунктов меню. Но не все Web-браузеры поддерживают некоторые более тонкие возможности. Так, например, средствами CSS можно реализовать изменение стилевого оформления элементов при наведении указателя мыши. Но данная функциональность поддерживается в полном объеме всеми Web-браузерами только для HTML-элементов `A` (гиперссылки). Определенная часть Web-программистов использует HTML-элементы `A` (гиперссылки) для выполнения несвойственных им функций, именно в расчете на изменение стиля при наведении указателя мыши. Мы такой способ не будем использовать. Для прочих HTML-элементов изменение стиля при наведении мыши кроссбраузерно приходится реализовывать при помощи JavaScript:

```
<span class="other"
      onmouseout="this.className='other'"
      onmouseover="this.className='selected';showPopup(this)">
```

Если элемент `DIV` содержит подчиненное выпадающее меню, необходимо не только изменить его стилевое оформление, отобразить подчиненное выпадающее меню на экране, но и правильно позиционировать подчиненное выпадающее меню, выровненное по нижней и левой границе пункта полоски меню, для чего вызывается функция `showPopup()`:

```
<div class="other"
     onmouseout="this.className='other'"
     onmouseover="this.className='selected';showPopup(this)">
  HTML
  <div class="popup">
    <div class="itemOther"
         onmouseover="this.className='itemSelected' "
         onmouseout="this.className='itemOther'"
         onclick="alertContent(this)">
      HEAD</div>
    <div class="itemOther"
         onmouseover="this.className='itemSelected' "
         onmouseout="this.className='itemOther'"
         onclick="alertContent(this)">
      BODY
    </div>
    <div class="itemOther"
```

```
        onmouseover="this.className='itemSelected' "
        onmouseout="this.className='itemOther'"
        onclick="alertContent(this)">
      SCRIPT
    </div>
  </div>
</div>
```

Функция `showPopup()` определяет координаты текущего пункта полоски меню и вычисляет координаты для подчиненного выпадающего меню:

```
function showPopup(element){
  var popup= firstElement(element.childNodes)
  if (!popup)
    return;
  popup.style.top = bhv.top(element) + element.offsetHeight + "px"
  popup.style.left = bhv.left(element) + "px"
}
```

Функции `bhv.top` и `bhv.left` определены в файле \bhvajax\bhv\util.js, который подключается элементом `SRIPT`, расположенным в HTML-элементе `HEAD`:

```
<script type="text/javascript" src="../bhv/util.js"></script>
```

В функциях `bhv.top` и `bhv.left` просматриваются HTML-элементы вверх по иерархии до корневого элемента и определяется накопительная сумма свойств `offsetTop` и `offsetLeft`:

```
bhv.top = function(element){
  var top = 0;
  try{
    top = element.offsetTop;
    while(element.offsetParent){
      element = element.offsetParent;
      top += element.offsetTop
    }
  } catch (ex){}

return top;
}

bhv.left = function(element){
  var left = 0;
```

```
  try{
    left = element.offsetLeft;
    while(element.offsetParent){
      element = element.offsetParent;
      left += element.offsetLeft
    }
  } catch (ex){}

return left;
}
```

Подобные функции содержат практически все библиотеки поддержки Ajax-приложений. В функциях `bhv.top` и `bhv.left` используется свойство HTML-элемента `offsetParent` (а не `parentNode`), относительно которого вычисляются величины `offseTop` и `offsetLeft`. Особенность данной реализации функции заключается в том, что обрабатывается исключительная ситуация при попытке передать функции некорректный аргумент. Для более простого поиска элемента, в котором расположено подчиненное выпадающее меню, используется функция `firstElement()`, возвращающая первый встретившийся в коллекции дочерних узлов `childNodes` узел типа "элемент":

```
function firstElement(nodes){
  for(var i = 0; i < nodes.length; i++)
    if (nodes[i].nodeType == 1)
      return nodes[i];
  return undefined;
}
```

Необходимость в такой функции возникает в связи с тем, что некоторые модели Web-браузеров формируют узлы, состоящие только из пробельных символов, а другие не формируют такие узлы.

5.4. Создание компонента "Полоска меню" средствами JavaScript

Для кого-то покажется проще определять полоску меню непосредственно в разметке HTML-документа. Но я отношусь к той части программистов, которые предпочитают автоматизировать процесс ручного кодирования. Мы применим уже неоднократно использовавшийся нами пример и определим данные для построения полоски меню в объекте JavaScript типа `Array`:

```
var menu = [
"Первое меню",
```

```
  [
  "Пункт меню 1", alertContent,
  "Пункт меню 2", alertContent,
  "Пункт меню 3", {
                    onClick: alertContent,
                    itemDisabled: checkDisabled
                  },
  "Пункт меню 4", alertContent
  ],
"Седьмое меню", alertContent
]
```

Каждому пункту полоски меню можно назначить функцию-обработчик выбора пункта меню:

```
["Седьмое меню", alertContent]
```

Можно назначить пункту полоски меню подчиненное выпадающее меню, а каждому пункту подчиненного выпадающего меню — функцию-обработчик выбора пункта меню:

```
[
"Первое меню",
  [
  "Пункт меню 1", alertContent,
  "Пункт меню 2", alertContent,
  ]
]
```

Для более сложных случаев можно назначить функцию-обработчик `onClick()`, которая делает пункт меню недоступным `itemDisabled()`, а также произвольные функции-методы, которые могут использоваться первыми двумя функциями:

```
[
"Первое меню",
  [
  "Пункт меню 3", {
                    onClick: alertContent,
                    itemDisabled: checkDisabled,
                  },
  ]
]
```

С учетом различных сочетаний, тестовый пример данных для построения полоски меню приведен в файле testMenuPopup2.html.

Мы создали описание полоски меню при помощи массива типа `Array`. Теперь остается средствами JavaScript (см. файл menu.js) создать разметку HTML-документа, которая будет повторять разметку из файла testMenuPopup1.html, только будет создаваться программно.

Основной функцией, которая формирует полоску меню, является `addMenuBehivior()`. Она перебирает элементы массива `Array` в цикле через два элемента. В четных (начиная с нуля) элементах хранится текст пунктов полоски меню, а в нечетных элементах (начиная с единицы) — либо ссылка на функцию-обработчик или на функции-методы, включая `onClick()`, либо массив с определением подчиненного выпадающего меню. Если в соответствующем пункте меню задана функция-обработчик, ссылка на него присваивается свойству `onclick`, а пункты, для которых должно активизироваться подчиненное выпадающее меню, создаются при вызове функции `addPopup()`, которая формирует подчиненное выпадающее меню:

```
function addMenuBehivior(element, o) {
  for (var I = 0; i < o.length; I += 2){
    var div = document.createElement("DIV");
    div.appendChild(document.createTextNode(o[i]));
    element.appendChild(div);
    div.className="other";
    div.onmouseout=div_onmouseout;
    div.onmouseover=div_onmouseover;
    if (typeof o[i+1] == "function")
        div.onclick = o[i+1];
    else if (typeof o[i+1] == "object" && o[i+1].length)
        addPopup(div, o[i+1]);
  }
}
```

В функции `addPopup()` происходит аналогичная обработка массива `Array`, который содержит определение подчиненного выпадающего меню. Единственное существенное отличие составляют строки, в которых анализируется, задана ли функция-обработчик выбора пункта подчиненного выпадающего меню непосредственно или в объекте в виде свойств:

```
if (typeof o[i+1] == "function")
  item.onclick = o[i+1];
else if (typeof o[i+1] == "object")
```

```
  for (var p in o[i+1])
    if (o[i+1][p] != Object.prototype[p]) {
      item[p] = o[i+1][p];
      item[p].onclick = item_onclick;
    }
```

Если обработчик задан в виде объекта, все свойства этого объекта переписываются в HTML-элемент `DIIV` текущего пункта меню. Сравнение с `Object.prototype` позволяет исключить унаследованные от объекта `Object` свойства:

```
if (o[i+1][p] != Object.prototype[p])
```

Свойству `onclick` присваивается ссылка на функцию `item_onclick()`:

```
function item_onclick() {
  if (! this.isDisabled())
    this.onClick();
}
```

Для пунктов, которые являются доступными, вызывается обработчик, заданный как `onClick()`. В JavaScript HTML-элементы имеют чувствительные к регистру имена свойств и методов. Поэтому `onclick()` и `onClick()` — это разные функции.

Для отображения недоступных пунктов меню в код функции `showPopup()` по сравнению с файлом testMenuPopup1.html внесены изменения:

```
for (var I = 0; i< popup.childNodes.length; i++)
  if (popup.childNodes[i].itemDisabled)
    if (popup.childNodes[i].itemDisabled())
      popup.childNodes[i].className = "itemDisabled";
    else
      popup.childNodes[i].className = "itemOther";
```

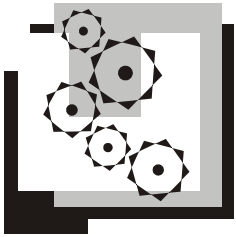
Соответственно, мы должны определить стилевой класс для недоступного пункта меню `itemDisabled`:

```
div.itemDisabled {
  color: #999;
  background-color: #ccc;
  padding: 2px ;
  border-width: 0px;
  margin: 0px;
}
```

Для этого стилевого класса использованы оттенки серого цвета, что интуитивно должно восприниматься пользователем как недоступность меню. Для того чтобы протестировать нашу работу с двумя различными стилевыми оформлениями, создадим файл menu2.css. Если в menu1.css эффект выделения достигался сменой стиля рамки, то в menu2.css выделение элемента будет достигаться сменой цвета фона `backgrounf-color`.

Наш компонент создан, и можно подвести итоги. В этой главе мы рассмотрели паттерн MVC и его применение в настольных (desktop) приложениях и реализацию в Web-приложениях как Model I и Model II. А также мы рассмотрели возможные способы реализации паттерна MVC в Ajax-приложениях. В Ajax-приложениях Модель может быть реализована на Web-сервере, а также в Web-браузере при помощи DOM-модели HTML-документов и объектов JavaScript.

В компоненте "Полоска меню" мы использовали реализацию Модели при помощи DOM-модели HTML-документа. Для более сложных компонентов, таких как Lookup Combobox и "Редактируемая таблица данных", мы будем использовать реализацию Модели при помощи объектов JavaScript.



Глава 6

Разработка Ajax-приложения "Редактор кода — отладчик PHP 5"

В этой главе мы установим PHP 5 на компьютер и сконфигурируем Web-сервер Apache для работы с PHP 5. Как всегда, особое внимание будет уделено правильной работе Web-приложений с кириллическими кодировками, в частности, с кодировкой Windows-1251. После установки PHP 5 мы разработаем приложение "Редактор кода — отладчик PHP 5", которое будет особенно полезно начинающим PHP-программистам. Разумеется, это приложение не идет в сравнение с профессиональными редакторами и отладчиками, но, благодаря своей легковесности и простоте использования, это приложение поможет быстро освоиться с программированием на PHP 5. Программный код приложения вы можете найти в папке \bhvajax\editor на прилагаемом к книге компакт-диске.

6.1. Установка PHP 5 на компьютер

Прежде чем начать установку PHP 5 на компьютер, вы должны получить установочные файлы PHP 5. Легче всего это сделать через Интернет, обратившись к Web-ресурсам:

http://www.php.net

http://www.php.net/downloads.php

На последнем из указанных Web-ресурсов вы найдете исходный код PHP 5 и несколько вариантов установочных пакетов последней поддерживаемой версии языка. На момент написания книги это была версия PHP 5.2.6. Из всех установочных пакетов проще всего воспользоваться PHP 5.2.6 zip package или PHP 5.2.6 installer. Первый из них содержит архив zip, который необходимо распаковать в папку на диске и вручную произвести настройку PHP 5 и Web-сервера Apache для совместной работы. Второй из пакетов содержит инсталляционную программу, которая устанавливает и конфигурирует

систему в процессе работы программы инсталляции. На момент вашего обращения к ресурсу **http://www.php.net** текущая версия PHP 5, скорее всего, будет другой. В составе пакетов имеется документация по установке, и вы должны строго следовать этой документации. Конфигурационные параметры Apache и PHP 5 время от времени меняются. Например, Apache может хранить все свои установки в едином файле httpd.conf или хранить конфигурацию в разных файлах. Так MIME-тип запросов может быть определен директивой в httpd.conf:

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

или определяться в файле mime.types:

```
application/x-httpd-php              php
application/x-httpd-php-source       phps
```

Если вы установили PHP 5 в папку C:\PHP, то в конфигурационном файле Apache httpd.conf должны присутствовать такие строки:

```
LoadModule PHP 5_module "C:/PHP/PHP 5apache2_2.dll"
PHPIniDir "C:/PHP/"
```

Путь `PHPIniDir` определяет папку, в которой находится конфигурационный файл PHP 5 php.ini. В файле php.ini вы должны определить такие установки:

```
extension_dir ="C:\PHP\ext"
default_mimetype = "text/html"
default_charset = "cp1251"
mbstring.language = Russian
mbstring.internal_encoding = cp1251
mbstring.http_input = auto
mbstring.http_output = cp1251
mbstring.encoding_translation = On
mbstring.detect_order = auto
```

Директива:

```
extension_dir ="C:\PHP\ext"
```

определяет, в какой папке будут располагаться дополнительные загружаемые модули PHP 5. В PHP 5 реализованы функции самого различного назначения. Часть из них включена в основной функционал PHP 5 и не требует загрузки дополнительных модулей. Другая часть, скажем, функции работы с конкретной системой управления баз данных, требует загрузки дополнительных модулей. Например, для получения доступа к MySQL 5 необходимо загрузить модуль `php_mysqli`:

```
extension=php_mysqli.dll
```

При загрузке модули могут потребовать наличие дополнительных библиотек, не включенных в состав установочного пакета PHP 5. В частности, это касается клиентских библиотек систем управления базами данных. Поэтому неиспользуемые модули нужно удалить из конфигурационного файла php.ini (или закомментировать при помощи знака `;` "точка с запятой"):

```
[PHP_DOMXML]
;extension=php_domxml.dll
```

Обратите внимание, библиотеку `php_domxml` не следует загружать в качестве дополнительного модуля. В PHP 5 реализована встроенная поддержка работы с DOM XML-документов. Некоторые новые функции PHP 5 после загрузки библиотеки `php_dom_xml` начинают работать в стиле PHP 4. Загружать библиотеку `php_dom_xml` следует только из соображений обратной совместимости кода, разработанного на PHP 4.

Что касается работы с MySQL, то встроенная поддержка работы с этим сервером в PHP 5 не поддерживается (как это было в PHP 4), и необходимо всегда явно загружать библиотеку `php_mysqli`:

```
extension=php_mysqli.dll
```

Параметры, которые определяют заголовки по умолчанию, задающие MIME-типа ответа Web-сервера и кодировку, также можно задать в php.ini:

```
default_mimetype = "text/html"
default_charset = "cp1251"
```

Мы договорились работать в продолжении всей книги с кодировкой Windows-1251. Особенность использования этой кодировки в Ajax-технологии связана с тем, что объект `XMLHttpRequest` отправляет все запросы к Web-серверу в кодировке UTF-8. В PHP 5 есть способ автоматического преобразования параметров запроса из многобайтовой кодировки, какой является UTF-8, в однобайтовую кодировку (например, Windows-1251). Для этого нужно использовать следующие конфигурационные параметры в файле php.ini:

```
mbstring.language = Russian
mbstring.internal_encoding = cp1251
mbstring.http_input = auto
mbstring.http_output = cp1251
mbstring.encoding_translation = On
mbstring.detect_order = auto
```

Скажу сразу, что такая перекодировка будет производиться только для параметров запросов типа `application/x-www-form-urlencoded`. Так что в некоторых случаях, и об этом мы поговорим в данной главе, придется явно перекоди-

ровать данные запроса. Перекодировка реализована при помощи загружаемой библиотеки mbstring, поэтому файл php.ini должен содержать строку, для загрузки этой библиотеки:

```
extension=php_mbstring.dll
```

Следующими важными параметрами, определяемыми в файле php.ini, являются: путь к папке, в которую будут загружаться файлы с Web-браузера, и путь к папке, в которой будут храниться данные сессии:

```
upload_tmp_dir="C:Temp\php\upload"
session.save_path="C:\Temp\php\session"
```

Разумеется, указанный вами путь может не совпадать с приведенным фрагментом, но, и это очень важно, папки должны существовать на компьютере. В противном случае загрузка файлов на сервер и возможность работать с сессиями будут недоступными.

Если вы загружаете PHP 5 в виде библиотеки при старте Web-сервера Apache:

```
LoadModule PHP 5_module "C:/PHP/PHP 5apache2_2.dll"
```

после редактирования файла php.ini конфигурации PHP 5 изменения вступят в силу только при следующем запуске Web-сервера Apache. Если в конфигурации содержатся ошибки, Web-сервер Apache может не стартовать. Наиболее частыми ошибками при конфигурировании PHP 5 являются ссылка на отсутствующие библиотеки, неверное задание папки, в которой находятся загружаемые библиотеки:

```
extension_dir ="C:\PHP\ext"
```

Также могут быть неверно заданы имена кодировок, временных зон, которые должны точно соответствовать документации PHP 5.

6.2. Особенности применения PHP 5 в Ajax-приложениях

Наша книга посвящена преимущественно клиентским технологиям, которые работают на стороне клиента Web-браузера. Серверные технологии, такие как PHP 5, не могут быть рассмотрены нами в полном объеме, поскольку это тема для отдельной книги. Я остановлюсь только на тех сторонах технологии PHP 5, на которые в литературе не акцентируется особое внимание, но которые будут востребованы вами уже в следующих главах.

Генерация динамического HTML-документа при помощи операторов `print` серверного языка, такого как Perl, может выглядеть громоздко. Использование

вместо этого встроенных в HTML-документы фрагментов программного кода оказалось идеей достаточно плодотворной, хотя и не лишенной недостатков. Недостатком явилась та легкость, с которой Вид (представление) HTML-документа можно смешивать с Моделью (данными, алгоритмами). Приведем простейший скрипт PHP:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Ajax: accordion</title>
<link rel="stylesheet" type="text/css" href="accordion.css">
</head>
<body>
<?php
  date_default_timezone_set("Europe/Moscow");
  echo date("Нч імин. sc.");
?>
</body>
</html>
```

Скрипт выводит текущее время и, одновременно, иллюстрирует тот момент, что в PHP 5 переход от HTML-документа к коду на языке программирования PHP 5 настолько прост, что если не предпринимать специальных мер, программный код очень быстро может стать неуправляемой смесью HTML-разметки и скриптов самого разнообразного содержания. Для того чтобы преодолеть эту проблему, были разработаны архитектуры Model I, Model II и реализующие их спецификации, библиотеки, которые построены в соответствии с паттерном MVC. Технология Ajax тоже может помочь в разделении Модели и Вида Web-приложений. Например, в *главе 10* мы будем определять редактируемые таблицы данных при помощи XML-документов:

```
<table_definition>
<sql>
 select kod,name,det, concat('kod=', kod) as where_clause from cennic16
</sql>
<update>
 update cennic16 set kod= $kod, name= '$name', det='$det'
</update>
<count>
 5
```

```
</count>
<order>
 order by det
</order>
<columns>
 <column>
  <header>
   Первая&lt;br&gt;колонка
  </header>
  <value>
   kod
  </value>
  <editor>
   new bhv.Combobox("combobox1", "comb1", 44685, 10, "cennic16", "kod",
                    "name", "det");
  </editor>
 </column>
 <column>
  <header>
   Вторая колонка
  </header>
  <displayValue>
   name
  </displayValue>
  <value>
   name
  </value>
 </column>
 <column>
  <header>
   Третья колонка
  </header>
  <value>
   det
  </value>
 </column>
</columns>
</table_definition>
```

Определив конфигурацию таблицы в файле test\table_definition.xml, таблицу можно будет создать оператором JavaScript `new`:

```
<script>
  new bhv.Table("table1", "test/table_definition.xml");
</script>
```

В приведенном фрагменте программного кода описание таблицы вынесено за пределы HTML-документа, т. е. использование Ajax может помочь разработчику в реализации паттерна MVC.

Для этого вам придется довольно активно использовать работу с DOM-моделью документа, которая в PHP 5 очень похожа на то, как мы работали с DOM-моделью на стороне Web-браузера средствами JavaScript, поскольку работа с DOM регламентируется общими для всех языков программирования спецификациями, такими как DOM Level 1:

```
$table_definition = new DOMDocument();
$table_definition->load($_REQUEST["definition"]);
$table_sql = $table_definition->getElementsByTagName('sql')
                              ->item(0)->firstChild->data;
$table_count = $table_definition->getElementsByTagName('count')
                                ->item(0)->firstChild->data;
$table_order = $table_definition->getElementsByTagName('order')
                                ->item(0)->firstChild->data;
foreach ($_REQUEST as $key => $value)
  $$key = $value;
if (isset($_REQUEST["command"])  && ('update' == $_REQUEST["command"])){
  $table_update = $table_definition->getElementsByTagName('update')
                                   ->item(0)->firstChild->data;
  eval("\$table_update = \"$table_update\";");
}
```

Вы можете убедиться, что при работе с DOM-моделью XML-документа в PHP 5 используются те же свойства и методы, что и в JavaScript. Боюсь только, что после легкости написания кода JavaScript, работа с PHP 5 может показаться намного более тяжеловесной.

В приведенном фрагменте мы использовали предопределенную переменную PHP 5 `$_REQUEST`. В этой переменной содержатся http-параметры запроса типа `application/x-www-form-urlencoded`. Это ассоциативный массив, который содержит те же данные, что `$_GET`, `$_POST` и `$_COOKIE` в одном массиве. В случае одноименных `GET`-, `POST`- и `COOKIE`-параметров, значение, которое хранится в ассоциативном массиве `$_REQUEST`, определяется настройками в php.ini. Использование ассоциативного массива `$_REQUEST` позволяет более

свободно обходиться с параметрами запросов, но приводит к снижению защищенности приложения, так как значение cookie или скрытых полей злоумышленник может ввести с клавиатуры в строке запроса Web-браузера.

Для одноименных параметров в PHP 5 иногда используют "хитрость" и задают имя параметра в виде массива без индекса:

```
<html>
  <head>
  </head>
  <body>
    <form>
      <input type="checkbox" name="chk[]">
      <input type="checkbox" name="chk[]">
      <input type="checkbox" name="chk[]">
      <input type="checkbox" name="chk[]">
    </form>
  </body>
</html>
```

При разборе параметра запроса на сервере скрипт PHP 5 в соответствии с синтаксисом будет формировать массив. Так что подобный код эквивалентный следующему:

```
<html>
  <head>
  </head>
  <body>
    <form>
      <input type="checkbox" name="chk[0]">
      <input type="checkbox" name="chk[1]">
      <input type="checkbox" name="chk[2]">
      <input type="checkbox" name="chk[3]">
    </form>
  </body>
</html>
```

Мы не будем использовать такой стиль, но в связи с тем, что подобный программный код — не редкость, эта возможность была рассмотрена.

Вполне вероятно, что вы будете использовать XML-документы в качестве тела запроса. Тогда ваш код JavaScript должен выглядеть так:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
```

```
<html>
<head>
  <script src="../bhv/util.js"></script>
</head>
<body>
<script>
bhv.sendRequest(
  "POST",
  "echo.php",
  "<?xml version='1.0' encoding='Windows-1251'?>\n\n"
  + "<test>Русский текст</test>",
  null,
  echo,
  null,
  [],
  "application/xml");

function echo() {
  alert(this.responseXML.documentElement.firstChild.data)
}
</script>
</body>
</html>
```

Мы установили MIME-тип запроса `application/xml` (использование типа `text/xml` в настоящее время считается устаревшим). Этот запрос не формирует значения `$_GET`-, `$_POST`- и `$_REQUEST`-переменных. Для запросов с типом, отличным от `application/x-www-form-urlencoded` и `multipart/form-data`, формируется значение переменной `$HTTP_RAW_POST_DATA`, которое содержит тело запроса в необработанном виде. Преобразования кодировки из UTF-8 в Windows-1251 для таких запросов необходимо производить явно:

```
<?php
header("Content-Type: application/xml");
#echo $HTTP_RAW_POST_DATA;
echo iconv("UTF-8", "Windows-1251",$HTTP_RAW_POST_DATA);
?>
```

Напоминаю, что не зависимо от кодировки HTML-документа, запрос `XMLHttpRequest` всегда отправляется в кодировке UTF-8. Для запросов типа `application/x-www-form-urlencoded` многобайтовая кодировка UTF-8 перекодируется в однобайтовую кодировку в соответствии с установками в php.ini.

В этом кратком обзоре некоторых возможностей PHP 5 я постарался дать информацию, которая в справочниках обычно содержится разрозненно и на которую, может быть, не обращено особое внимание. Конечно, такой обзор не может заменить изучение документации по языку PHP 5 и специальной литературы, среди которой особо хочу порекомендовать книгу: Котеров Д. В., Костарев А. Ф. PHP 5. — СПб.: БХВ-Петербург, 2005.

6.3. Разработка приложения "Редактор кода — отладчик PHP 5"

Гораздо успешней изучение языка программирования происходит, если вы можете тут же протестировать написанный код и получить подробные сообщения об ошибках. Для этого я и решил, вместе с вами, разработать небольшое приложение: "Редактор кода — отладчик PHP 5". Приложение реализовано в Web-браузере с использованием технологии Ajax. В верхней части панели Web-браузера расположены кнопки, щелкнув по которым мышью можно загрузить текст скрипта из файла, выполнить его с заданными http-параметрами и сохранить текст скрипта в файл. Кроме кнопок в верхней панели расположены два текстовых поля, в которые вы можете ввести адрес загружаемого или сохраняемого файла и http-параметры. Адрес файла задается относительно папки \bhvajax и не должен начинаться с косой черты (слэша). Наряду с абсолютными и относительными URL-адресами и путями, часто используется такое понятие, как *каталог приложения*. Это понятие является логическим и с точки зрения http-протокола не существует. Удобно, если некоторые адреса задаются относительно каталога приложения. В этом случае на одном Web-сервере можно разместить неограниченное количество приложений — каждое в своем каталоге приложения. При этом возникает вопрос: как задавать адреса Web-ресурсов приложения? Конечно, для большинства случаев достаточно использовать относительную адресацию. Но рано или поздно возникает ситуация, когда относительный адрес использовать неудобно или даже невозможно. Поэтому найден следующий выход: задавать адрес относительно каталога приложения и при помощи несложного кода расширять этот адрес до абсолютного пути:

```
<script>
var APPLICATION_FOLDER = "/bhvajax/";
function getAbsolutePath(path) {
  return APPLICATION_FOLDER + path;
}
</script>
```

Я думаю, идея достаточно понятна. Может возникнуть другой вопрос: как определять APPLICATION_ROOT? Это можно сделать один раз для всего приложения, изменив скрипт. Но еще проще — определить адрес APPLICATION_ROOT по адресу нашей загружаемой библиотеки bhv\util.js. Мы договорились, что эта библиотека всегда будет загружаться первой в наших HTML-документах. В HTML-документе ссылку на эту библиотеку лучше всего задавать относительным адресом. Проанализировав этот адрес, можно определить путь к каталогу приложений:

```
bhv.APPLICATION_FOLDER = null;

bhv.getApplicationFolder = function(){
  if (bhv.APPLICATION_FODER)
    return bhv.APPLICATION_FOLDER;
  var scripts = document.getElementsByTagName("SCRIPT");
  var indexOfRoot = -1;
  for (var i = 0; i < scripts.length; i++) {
    indexOfRoot = String(scripts[i].src).
      replace(/\\/g,'/').lastIndexOf('bhv/util.js');
    if (indexOfRoot >= 0){
        bhv.APPLICATION_FOLDER =
          new String(scripts[i].src).substring(0, indexOfRoot)
        return bhv.APPLICATION_FOLDER;
    }
  }
}
```

Адреса загружаемых в разрабатываемый нами Редактор файлов будут всегда задаваться относительно каталога приложения. В нашем случае это будет каталог \bhvajax. В листинге 6.1 приведен код HTML-документа, который будет служить Web-интерфейсом Редактора кода-отладчика.

Листинг 6.1. Редактор кода-отладчика /bhvajax/editor/editor.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Простой PHP - редактор/отладчик в Web-браузере</title>
<link rel=stylesheet type="text/css" href=editor.css>
<script type="text/javascript" src="../bhv/util.js"></script>
```

```
</head>
<body>
Файл для редактирования
/<input type="text" style="width:30%" id="filename">
<input type="button" value="Загрузить" onclick="loadFile()">
<input type="button" value="Выполнить с параметрами:"
                    onclick="testFile()">
<input type="text" id="parameters">
<input type="button" value="Сохранить" onclick="saveFile()">
<textarea id="sourcecode" class="programEditPane"></textarea>
<div id="programoutput"></div>
<script type="text/javascript">
function loadFile(){
  var filename = document.getElementById("filename");
  var sourcecode = document.getElementById("sourcecode");
  bhv.sendRequest(
    "get",
    "editor_load.php",
    "filename="+filename.value,
    true,
    function(){sourcecode.value = this.responseText;})
}

function saveFile(){
  var filename = document.getElementById("filename");
  var sourcecode = document.getElementById("sourcecode");
  bhv.sendRequest(
    "post",
    "editor_save.php",
    "filename="+filename.value+"&sourcecode="
    +encodeURIComponent(
        sourcecode.value.replace(/(\r\n)|(\n)/g,"\r\n")),
    false,
    function(){alert(this.responseText);})
}

function testFile(){
  var filename = document.getElementById("filename");
  var sourcecode = document.getElementById("sourcecode");
```

```
  var programoutput = document.getElementById("programoutput");
  var parameters = document.getElementById("parameters");
  bhv.sendRequest(
    "post",
    "editor_test.php",
    "filename="+encodeURIComponent(""+filename.value)
        +"&sourcecode="+encodeURIComponent(""+sourcecode.value)
        +"&"+parameters.value,
    true,
    function(){programoutput.innerHTML=this.responseText;})
}
</script>
</body>
</html>
```

Как вы обнаружите, код приложения достаточно прост и заключатся в вызове функций `loadFile()`, `testFile()` и `saveFile()`. Эти функции вызывают соответствующие скрипты editor_load.php, editor_test.php и editor_save.php, при помощи разработанной нами в *главе 2* функции `bhv.sendRequest()`.

При загрузке скрипта PHP 5 текст скрипта присваивается HTML-элементу `TEXTAREA`:

```
function(){sourcecode.value = this.responseText;}
```

При тестировании результат присваивается свойству `innerHTML` элемента `DIV`:

```
function(){programoutput.innerHTML = this.responseText;}
```

При сохранении файла выводится сообщение об успехе или ошибке:

```
function(){alert(this.responseText);}
```

Из интересных моментов можно рассмотреть замену символов "перевод строки" на пару символов "перевод строки" и "возврат каретки". Дело в том, что элемент `TEXTAREA` в некоторых Web-браузерах все символы "возврат каретки" убирает и оставляет только "перевод строки". Для восстановления возвратов каретки я воспользовался регулярным выражением:

```
sourcecode.value.replace(/\r\n)|(\n)/g,"\r\n");
```

В принципе, вы можете не делать такое преобразование, если ваш текстовый редактор правильно отображает текст без символов "возврат каретки".

Стилевое оформление задано каскадными таблицами стилей:

```
<link rel=stylesheet type="text/css" href=editor.css>
```

В этом случае стилевое оформление не несет никакой функциональной нагрузки и поэтому подробно не рассматривается.

Гораздо интереснее рассмотреть код скриптов PHP 5, которые и берут на себя весь функционал нашего приложения. Проще всего устроена загрузка текста скрипта в файле editor_load.php.

Прежде всего, устанавливается заголовок ответа:

```
header("Content-Type: text/html; charset=Windows-1251");
```

Устанавливать заголовок необходимо до того, как будет выведен текст.

Обработка ошибок ведется в немного старомодном стиле при помощи функции-обработчика `eh()`:

```
set_error_handler("eh");
```

Для того чтобы в случае ошибки вывод не смешивался с сообщениями об ошибке, используется буферизация вывода:

```
ob_start();
ob_clean();
ob_end_flush();
```

Первая из приведенных функций направляет весь вывод в буфер. Вторая функция очищает буфер, а третья — переписывает текст из буфера в выходной поток Web-сервера, чтобы этот текст был отправлен Web-браузеру.

Код скрипта расположен в файле \bhvajax\editor\editor_load.php. То есть, по нашей терминологии, каталогом приложения \bhvajax будет родительская папка для папки \bhvajax\editor. Поэтому в скрипте устанавливается в качестве текущего каталог приложения:

```
chdir('../');
```

Имя загружаемого файла относительно этого каталога получаем из http-параметра:

```
$filename = $_GET['filename'];
```

Далее воспользуемся функцией, которая одновременно читает файл, выводит его в поток вывода и, если не случилась ошибка, отправляет содержимое файла Web-браузеру:

```
readfile($filename);
ob_end_flush();
```

После загрузки текст файла можно корректировать, тестировать и сохранять.

Теперь рассмотрим скрипт editor_save.php, который сохраняет отредактированный текст в файл. Многое из кода скрипта editor_save.php уже знакомо

вам по скрипту editor_load.php. Текст загружаемого файла берется из параметров запроса:

```
#$sourcecode = iconv('UTF-8', 'Windows-1251', $_POST['sourcecode']);
$sourcecode = $_POST['sourcecode'];
```

Если бы мы не установили автоматическую перекодировку многобайтовых символов в однобайтовые, то нам пришлось бы явно сделать перекодировку при помощи функции `iconv()` (сейчас эта строка кода отключена знаком комментария `#`).

Если папка, в которой необходимо сохранить файл, не существует, она будет создана:

```
if (! file_exists(dirname($filename)))
  mkdir(dirname($filename), 0777, TRUE);
```

Конечно, этому может помешать ограничение безопасности операционной системы. Если файл существует, то для большей безопасности старая версия переименовывается — к ней присоединяется число от нуля и выше, пока не будет сгенерировано новое имя файла, не имеющееся в данной папке:

```
if (file_exists($filename)){
  $i = 0;
  while (file_exists($filename . $i))
    $i++;
  copy($filename, $filename . $i);
}
```

Далее, идет открытие, запись и закрытие файла — операции, от которых многие программисты уже успели отвыкнуть:

```
$file_handler = fopen($filename, "w+");
fwrite($file_handler, $sourcecode);
fclose($file_handler);
echo "Файл сохранен";
ob_end_flush();
```

Если в процессе сохранения не произошло ошибок, файл считается сохраненным, и Web-браузеру отправляется текст сообщения: "Файл сохранен".

Напоминаю, что за обработкой ошибок следит функция `eh()`:

```
ob_start();
function eh($errorcode, $message, $file, $line){
  ob_clean();
  header("BHV-errorcode: $errorcode");
```

```
  header("BHV-errormessage: $message");
  header("BHV-errorfile: $file");
  header("BHV-errorline: $line");
  echo "Error: #$errorcode - $message in $file line $line \n";
  die('Файл не сохранен');
}
set_error_handler("eh");
```

Начиная с этого момента, любая ошибка приводит к передаче управления функции `eh()` и отправке Web-браузеру текста сообщения:

```
die('Файл не сохранен');
```

Я думаю, что этот код не должен вызвать вопросов, и перехожу к наиболее интересному коду скрипта editor_test.php. В целом, скрипт editor_test.php является вариацией на тему предыдущих двух скриптов. Поэтому рассмотрим только отличительные строки кода. Для информирования пользователя об ошибках создаются глобальные переменные:

```
$editor_test_is_ok = true;
$editor_test_errorline = -1;
$editor_test_errormessage = '';
```

Они становятся доступными из тела функции-обработчика ошибок `eh()` благодаря использованию ключевого слова `global`:

```
function eh($errorcode, $message, $file, $line){
  global $editor_test_is_ok,
  $editor_test_errorline,
  $editor_test_errormessage;
  if ($editor_test_is_ok){
    $editor_test_errorline = $line;
    $editor_test_errormessage = $message;
  }
  $editor_test_is_ok = false;
  ob_clean();
  header("BHV-errorcode: $errorcode");
  header("BHV-errormessage: $message");
  header("BHV-errorfile: $file");
  header("BHV-errorline: $line");
  echo "Error: #$errorcode - $message in $file line $line";
}
```

Кроме установки функции-обработчика ошибок, устанавливается и уровень обработки ошибок, при возникновении которых будут выдаваться сообщения об ошибках:

```
set_error_handler("eh");
error_reporting(E_ALL+E_STRICT);
```

Такой уровень позволит нам получать достаточно подробные сообщения об ошибках.

Тестируемый скрипт переписывается в файл, для того чтобы его можно было загрузить и выполнить оператором `include()`:

```
$filename = $_REQUEST['filename'];
$tmp_filename = "edit_debug.php";
#$sourcecode = iconv('UTF-8', 'Windows-1251', $_REQUEST['sourcecode']);
$sourcecode = $_REQUEST['sourcecode'];


$file_handler = fopen($tmp_filename, "w+");
fwrite($file_handler, $sourcecode);
fclose($file_handler);
```

Далее, этот файл загружается и выполняется оператором `include()`:

```
ini_set("display_errors", "TRUE");
include($real_tmp_path);
ini_set("display_errors", "FALSE");
```

Функция `ini_set()` устанавливает параметр из конфигурации php.ini в нужное нам значение, чтобы сообщения об ошибках направлялись в вывод. Если в процессе выполнения `include()` произошла ошибка, специальному флагу обработчиком ошибок будет присвоено логическое значение `false`:

```
$editor_test_is_ok = false;
```

Приведенный код обрабатывает ошибки двумя способами. Если это грубая ошибка синтаксиса, то будет выведено предупреждающее сообщение, поскольку выполнение `include()` будет прервано PHP 5:

```
echo "Error: #$errorcode - $message in $file line $line";
```

Если же ошибка не критическая, то будет выведен листинг файла с подсветкой синтаксиса (рис. 6.1). Для этого используется функция PHP 5 `highlight_string()`:

```
$highlighted_string = highlight_string($sourcecode, true);
```

После этого, довольно неприятный участок кода, который я не буду разбирать подробно, находит строку с ошибкой, выводит перед этой строкой

предупреждающее сообщение в красной рамке и, наконец, возвращает подсвеченный текст скрипта Web-браузеру:

```
echo($highlighted_string);
```

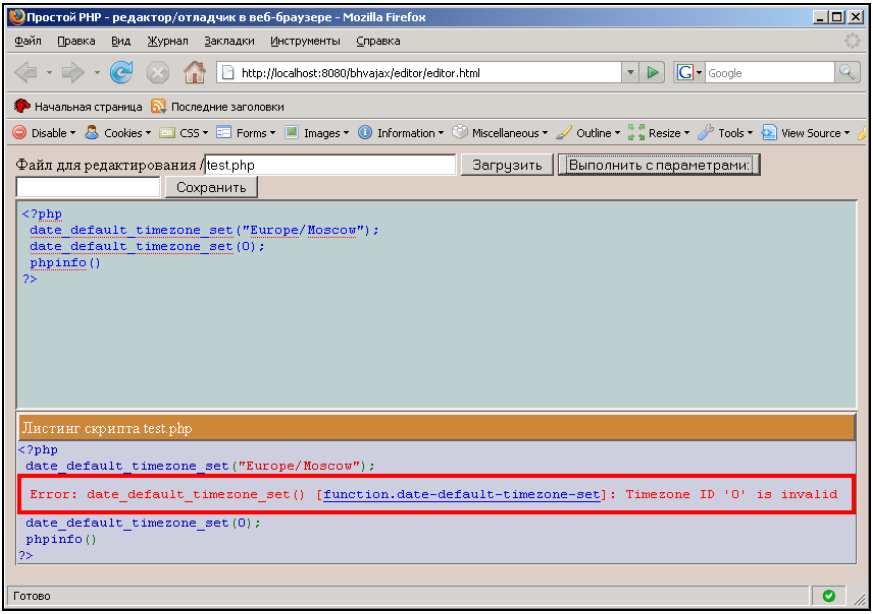


Рис. 6.1. Сообщение об ошибке и листинг в окне "Редактора кода — отладчика PHP 5"

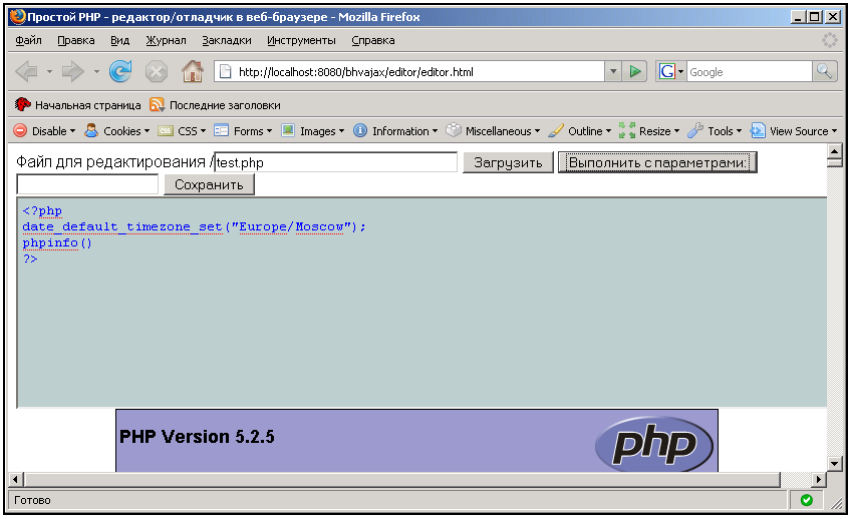
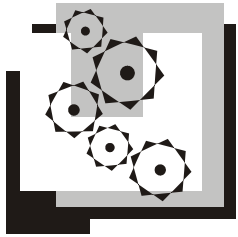


Рис. 6.2. Результат выполнения скрипта в окне "Редактора кода — отладчика PHP 5"

Если код не содержит ошибок и не произошло ошибок времени выполнения, то Web-браузеру будет возвращен результат выполнения тестируемого скрипта (рис. 6.2). Это результат действия оператора `include()`, который не прерывался ошибками.

Наше приложение "Редактор кода — отладчик PHP 5" разработано. Для его работы скрипты должны обладать значительными полномочиями по изменению и удалению файлов. Поэтому работать он будет только на компьютере, который сконфигурирован для тестирования и обучения программированию. На реальных Web-хостингах защита, скорее всего, не позволит выполнять такие небезопасные операции, как изменение и удаление файлов на Web-сервере.

Я уверен, что программистам, которые только начинают работать с PHP 5, такой "Редактор кода — отладчик PHP 5" поможет быстро и наглядно познакомиться с возможностями PHP 5, хотя для профессиональной разработки приложений вы, скорее всего, будете использовать более серьезные продукты с наличием автоподстановки текста и всплывающих подсказок. Впрочем, может быть, вы захотите весь такой функционал реализовать самостоятельно, с помощью технологии Ajax, тем более что в PHP 5 имеются средства для отражения (reflection).



Глава 7

Разработка Ajax-приложения "Консоль базы данных MySQL 5"

При создании Ajax-компонентов для Web-приложений в следующих главах нам потребуется работать с базами данных. Из всех серверов баз данных наш выбор пал на очень распространенный в Интернете сервер баз данных MySQL 5. Впрочем, как и ранее, я могу сказать, что в технологии Ajax наиболее интересная часть связана с программированием на стороне клиента Web-браузера средствами JavaScript. Поэтому вы без больших проблем можете адаптировать код Ajax-компонентов и приложений, рассматриваемых в книге, для использования с любым сервером баз данных, как и с любым Web-сервером. Если же вы будете работать с сервером баз данных MySQL 5, то сможете получить дополнительные выгоды от использования программного кода на прилагаемом к книге диске.

В этой главе мы установим сервер баз данных MySQL 5 на компьютер и создадим небольшое приложение *"Консоль базы данных MySQL 5"*. Его разработка позволит нам решить целый ряд полезных задач:

- мы разберем функции языка PHP 5, при помощи которых можно работать с сервером баз данных MySQL 5;
- мы создадим простейшую Консоль для работы с сервером баз данных MySQL 5, которая позволит быстро познакомиться операторами языка SQL тем читателям, для кого эта тема нова;
- и, наконец, мы заполним учебную базу данных, которую будем использовать для работы с Ajax-компонентами в последующих главах, тестовыми данными.

Текст приложения "Консоль базы данных MySQL 5" вы можете найти в папке \bhvajax\mysql на прилагаемом к книге компакт-диске.

7.1. Установка сервера баз данных MySQL на компьютер

Для установки сервера баз данных MySQL 5 вам необходимо иметь инсталляционный пакет, который можно загрузить с интернет-ресурса **http://www.mysql.com**. Мы будем использовать сервер баз данных MySQL 5.0, который на момент написания книги был последней доступной версией. Процесс установки сервера баз данных MySQL 5 предельно прост для пользователя. Конечно, в то время, когда вы будете держать в руках эту книгу, может выйти новый релиз сервера баз данных MySQL, и процесс инсталляции будет отличаться в деталях. Но можно быть уверенным, что этот процесс останется таким же удобным для пользователя.

Чтобы процесс инсталляции сервера баз данных MySQL 5.0 стал для вас еще более понятным, я проделаю вместе с вами все шаги инсталляции и приведу соответствующие диалоговые окна. Вы можете не во всем повторять мою конфигурацию. Это не будет иметь существенных последствий для работы примеров книги.

> ***Очень важное предупреждение***
>
> Рассматриваемая конфигурация сервера баз данных MySQL 5 рассчитана на работу в качестве среды для изучения программирования и не может быть рекомендована для реально работающих серверов баз данных.

В процессе инсталляции для продвижения к следующему шагу необходимо нажимать кнопку Next. В любой момент инсталляцию можно прервать, нажав кнопку Cancel. Рекомендую выбрать тип инсталляции Custom (рис. 7.1), которая позволит, по крайней мере, ознакомиться со списком компонентов, которые вы устанавливаете на компьютер. При выборе устанавливаемых компонентов рекомендую установить все доступные компоненты.

После выбора устанавливаемых компонентов, программа инсталляции предложит вам сконфигурировать экземпляр сервера баз данных (**Instance**) — выберите эту возможность. При конфигурировании сервера рекомендую выбрать пункт **Detailed Configuration** (рис. 7.2), который позволит сделать более подробный выбор установок. В этом случае вы сможете выбрать тип сервера (рис. 7.3). Для нашего учебного сервера я рекомендую выбрать тип сервера **Developer Machine**, который позволит более экономно использовать память. Один из самых важных для нас пунктов — выбор кодировки. Мы условились в примерах книги работать с кодировкой Windows-1251, которая для MySQL задается как cp1251 (рис. 7.4). На рис. 7.5 предлагается установить флажки **Install As Windows Service** и **Launch the MySQL Server automatically**, которые автоматически запускают сервер баз данных MySQL 5 как *службу* при загрузке компьютера. Наконец, программа инсталляции предложит установить

пароль пользователя `root` (рис. 7.6). Пользователь `root` является суперпользователем сервера баз данных MySQL 5, поэтому имя пользователя `root` и введенный вами пароль необходимо хорошо запомнить, в противном случае вы не сможете получить доступ к серверу баз данных и продолжить работу с базами данных.

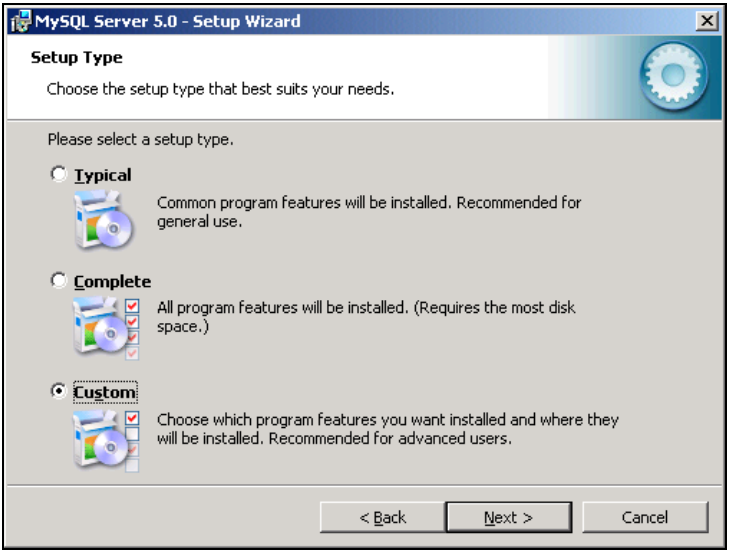


Рис. 7.1. Инсталляция MySQL 5. Выбор типа инсталляции сервера

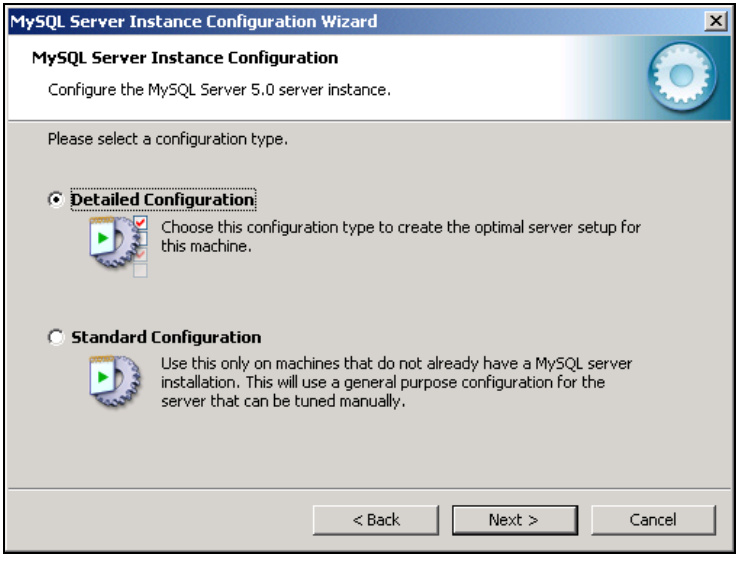


Рис. 7.2. Инсталляция MySQL 5. Выбор подробной конфигурации сервера

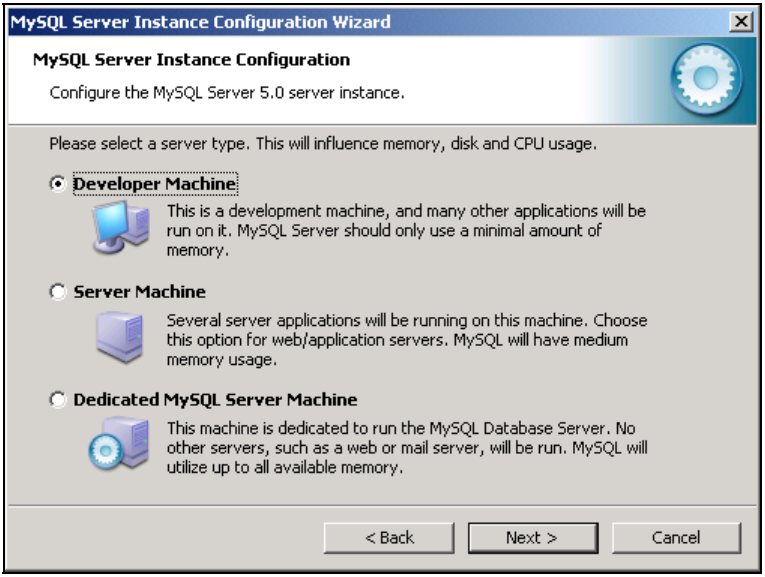


Рис. 7.3. Инсталляция MySQL 5. Выбор типа сервера

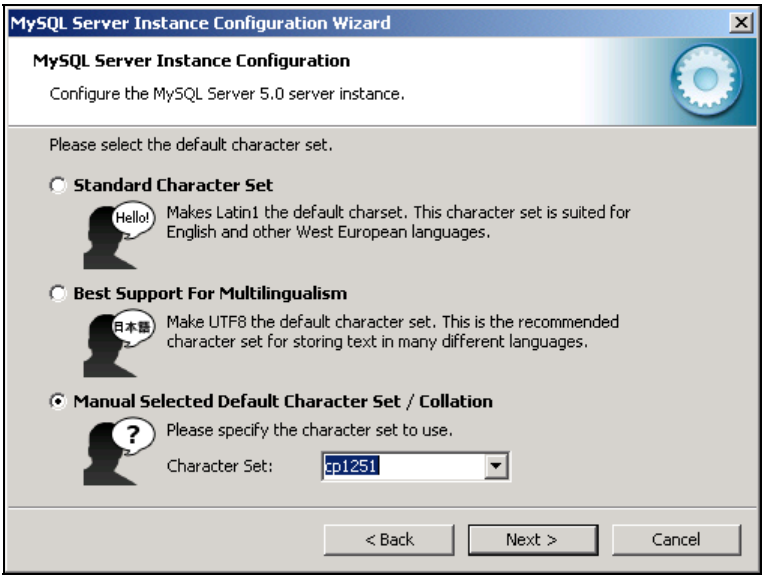


Рис. 7.4. Инсталляция MySQL 5. Выбор символьной кодировки сервера

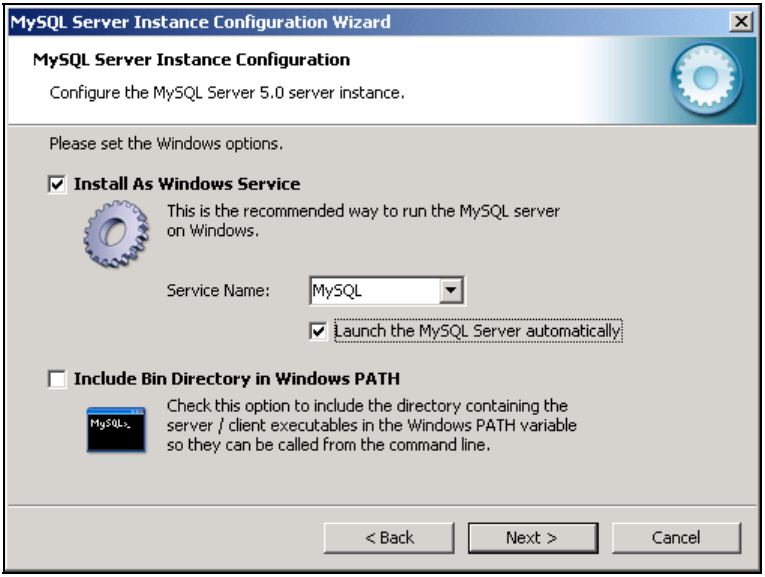


Рис. 7.5. Инсталляция MySQL 5. Выбор типа запуска сервера

Рис. 7.6. Инсталляция MySQL 5. Задание пароля пользователя `root`

Если вы отказались от конфигурирования сервера баз данных MySQL 5 при инсталляции, вы всегда сможете сделать это, запустив Мастер конфигурирования из меню **Пуск | Программы | MySQL | MySQL Server 5.0 | MySQL Server Instance Config Wizard**. Этот Мастер конфигурирования вносит изменения в файл my.ini. В последующем вы можете работать непосредственно с этим файлом для внесения более тонких изменений в параметры конфигурации. Впрочем, это необходимо только для реально работающих под нагрузкой серверов баз данных MySQL 5, а не для нашего учебного сервера.

Как видите, установка сервера MySQL 5.0 — достаточно простой процесс, и я проделал работу вместе с вами только для того, чтобы акцентировать внимание на некоторых деталях установки, и чтобы вы не забыли выбрать кодировку cp1251 (Windows-1251) в качестве кодировки по умолчанию. Ведь именно с этой кодировкой мы условились работать в примерах книги.

7.2. Краткий обзор реляционных баз данных

Настоящий раздел не может заменить документацию и специальную литературу по *базам данных* и серверу баз данных MySQL 5. С другой стороны, я не могу оставить читателя без краткого описания терминологии, используемой в реляционных базах данных и подмножества языка SQL, которое было бы достаточным для работы с примерами книги. Я постараюсь дать только самые необходимые сведения настолько полно, насколько это позволяет формат книги, которая не посвящена специально базам данным.

В настоящее время наибольшее распространение получили реляционные базы данных. Практически все системы управления реляционными базами данных обеспечивают работу с языком SQL, при помощи которого можно определять и модифицировать структуру данных, добавлять, изменять и удалять данные, а также делать самые разнообразные выборки данных.

Наибольшее распространение получили серверы реляционных баз данных, которые основаны на клиент-серверной архитектуре и к которым относится MySQL 5. Эти серверы обеспечивают устойчивую работу с базами данных одновременно большого количества клиентов (это могут быть десятки, сотни, тысячи и миллионы клиентов — все зависит от используемого оборудования и программного обеспечения). Кроме этого, реляционную модель данных реализуют так называемые настольные базы данных, такие как dBASE, Foxpro, Clarion, Paradox, Access. Практически все ведущие настольные базы данных в настоящее время поддерживают возможность работы в качестве клиентов серверов баз данных при помощи технологий ODBC, BDE, ADO и др.

Реляционные базы данных построены на строгой теории реляционных баз данных, которая, в свою очередь, базируется на теории множеств и теории

отношений. Реляционная база данных представляет собой набор таблиц, между которыми заданы связи. Строки таблиц называют записями, а элементы, из которых состоит запись — полями. В теории реляционных баз данных таблицы называют отношениями, записи — кортежами, а поля — атрибутами.

Таблица реляционной базы данных не может содержать повторяющихся записей (строк). Это требование как бы унаследовано из теории множеств. Минимальный набор полей, который позволяет отличить запись от любой другой записи, называется ключом. Все значения ключа в пределах одной таблицы должны быть уникальными. Каждая таблица должна иметь, по крайней мере, один ключ, что прямо следует из того, что таблица не может содержать повторяющихся записей. Ключи таблицы могут состоять из одного поля — такие ключи называют атомарными или простыми ключами. Ключи могут состоять из нескольких полей — составные ключи. Таблица базы данных может иметь один ключ или несколько ключей. Один из ключей назначают в качестве первичного ключа, а остальные называют потенциальными (в теории реляционных баз данных) или альтернативными (в конкретных реализациях некоторых баз данных).

Часто используют суррогатные первичные ключи — это ключи, состоящие из поля (или полей), которые не несут информации из предметной области, а служат заменой (суррогатом, паллиативом) для естественного (натурального) первичного ключа. В качестве суррогатных ключей чаще всего используются счетчики (генераторы, последовательности) `AUTO_INCREMENT` или глобально-уникальные идентификаторы (GUID). Существуют полярные мнения о допустимости использования суррогатных ключей. Сторонники их использования рекомендуют всегда создавать суррогатный ключ для каждой таблицы, за исключением самых тривиальных случаев, когда имеется достаточно ярко выраженный естественный (натуральный) ключ. Другая часть разработчиков баз данных находит эти ключи избыточными, особенно с появлением технологии каскадного обновления записей в связанных таблицах. Среди сторонников суррогатных ключей идет дискуссия по поводу допустимости использования глобально-уникальных идентификаторов GUID в качестве суррогатного ключа. Еще одним яблоком раздора является вопрос, могут ли суррогатные ключи быть составными или допускаются только простые (атомарные) суррогатные ключи. Мы принципиально не включаемся в эту дискуссию, тем более что любому практически работающему программисту в каждой конкретной ситуации ясно, какую из возможностей следует использовать. Для того чтобы подчеркнуть тот факт, что первичный ключ не является суррогатным — его называют естественным (натуральным) первичным ключом.

Замечание

Наличие суррогатного первичного ключа у таблицы не снимает вопрос о необходимости иметь естественный (натуральный) ключ, который в этом случае будет потенциальным ключом. Суррогатный ключ необходимо воспринимать как технический прием, который позволяет получить более простой программный код и существенно ускорить доступ к данным (особенно, соединение данных, выбранных из нескольких таблиц). Однако наличие суррогатного ключа только формально обеспечивает таблице соответствие принципам правильного построения реляционных баз данных. На самом деле, таблица, в которой существует только суррогатный ключ, с точки зрения логики построения данных, остается по-прежнему таблицей без первичного ключа и, возможно, даже с повторяющимися строками (если исключить из рассмотрения поле суррогатного ключа), что может свидетельствовать о недостаточно проработанной структуре базы данных.

Правильно разработанная структура базы данных должна удовлетворять специальным правилам, которые базируются на теории отношений и называются *нормальными формами*. Процесс приведения структуры к соответствию нормальным формам называют *нормализацией*. Строгое определение нормальных форм вы можете прочитать в литературе по базам данных, я же приведу не строгую трактовку этих правил.

Мы уже договорились, что таблица не может содержать повторяющихся строк и должна иметь первичный ключ. Кроме того, все поля таблицы должны быть атомарными, т. е. неделимыми с точки зрения конкретной задачи, которая решается при помощи базы данных. Наиболее часто встречающийся пример нарушения атомарности — хранение фамилии, имени и отчества в одном поле таблицы, а не в трех полях. Тем не менее, я хочу удержать вас от резких замечаний в сторону разработчиков структуры базы данных, если они вам предложат такое неатомарное поле. Дело в том, что поле базы данных является атомарным или неатомарным не само по себе, а только в связи с решением конкретной задачи. Условие атомарности говорит только о том, что никакие запросы к базе данных не будут требовать вычленения части поля, в нашем случае — отдельно фамилии, отдельно имени и отдельно отчества. Очевидно, что для обеспечения общепринятой сортировки сначала по фамилии, а затем по имени и отчеству при выводе списков, нам пришлось бы выделять отдельно фамилию, отдельно имя и отдельно отчество. Поэтому в подавляющем большинстве случаев фамилию, имя и отчество все же необходимо хранить в различных полях таблицы.

Когда вы будете отрабатывать поля своих таблиц на атомарность, всегда помните о том, что атомарность должна рассматриваться с точки зрения конкретной решаемой задачи. В качестве упражнения можете раскрыть свой паспорт (если он у вас есть) и проработать структуру базы данных для хранения паспортных данных, обращая особое внимание на атомарность полей

(атрибутов). Вам будет над чем подумать. Например, у паспорта есть серия и номер. Это два атомарных поля или одно атомарное поле? Кем выдан? Вот где есть простор для размышлений!

Следующее требование — в таблице не допускаются повторяющиеся столбцы (поля, атрибуты). Например, если по правилам пользования библиотекой читатель может получить на руки не более трех книг, есть соблазн учитывать выданные на руки книги в таблице, имеющей структуру:

```
Выданные_книги (Читатель ⇒ Книга1, Книга2, Книга3)
```

Очевидно, что все три книги являются значением, взятым из одного и того же множества книг (в теории реляционных баз данных множество значений столбца или поля называют доменом). Такую структуру таблицы можно признать структурой с повторяющимися столбцами, которую лучше представить так:

```
Выданные_книги (Читатель, Книга)
```

В отличие от первого варианта, каждый читатель будет представлен не одной строкой, а от нуля до трех строк (если вы помните, по условию задачи библиотека выдает не более трех книг на руки одному читателю).

Исключение повторяющихся столбцов может показаться совсем простым делом, пока речь идет о книжных примерах. Но не всегда так просто ответить на вопрос, являются ли столбцы таблицы повторяющимися, взятыми из одного домена (множества) или они все же принадлежат разным доменам (множествам). Давайте рассмотрим самый простой ценник на товар, который может иметь оптовую и розничную цену:

```
Ценник (Товар ⇒ Цена оптовая, Цена розничная)
```

А теперь попробуйте дать ответ на вопрос, принадлежат ли две цены к одному домену, и не следует ли хранить данные таблице с такой структурой:

```
Ценник (Товар, Тип цены ⇒ Цена)
```

Ответить на такой вопрос один раз и навсегда невозможно, поскольку, как и в случае с атомарностью, все зависит от конкретной решаемой задачи.

Для дальнейшего улучшения структуры данных используется понятие *функциональной зависимости*. Говорят, что поле (атрибут) Y таблицы (отношения) функционально зависит от поля (атрибута) X или набора полей (атрибутов) (X_1, X_2, ..., X_N), если в любой момент времени каждому определенному набору значений атрибутов (X_1, X_2, ..., X_N) соответствует ровно одно значение атрибута Y. Это условие можно символически записать одним из приведенных способов:

$(X_1, X_2, ..., X_N) \Rightarrow Y$

$Y = R(X_1, X_2, ..., X_N)$

$(X_1, X_2, ..., X_N)\ R\ Y$

Все не ключевые поля (атрибуты) таблицы (отношения) функционально зависят от ключа (набора ключевых полей), что прямо следует из определения ключа. Дальнейшим развитием понятия *функциональной зависимости* является понятие *полной функциональной зависимости*. Говорят, что поле (атрибут) Y таблицы (отношения) *функционально полно* зависит от набора полей (атрибутов) (X_1, X_2, ..., X_N), если поле (атрибут) Y функционально зависит от набора полей (X_1, X_2, ..., X_N) и не зависит ни от какого нетривиального подмножества этих полей. Если поле (атрибут) таблицы (отношения) функционально полно зависит от подмножества ключа или от неключевого поля (атрибута), такое поле (атрибут) можно вынести в отдельную таблицу (отношение). В качестве примера рассмотрим таблицу `Рабочие`, которая содержит сведения о рабочем, включая профессию, тарифный разряд, тарифную сетку и тарифную ставку:

```
Рабочие(Табельный номер ⇒ Фамилия, Профессия,
        Тарифный разряд, Тарифная сетка, Тарифная ставка)
```

Ключом таблицы (отношения) `Рабочие` является поле (атрибут) `Табельный номер`. На основании формального анализа нельзя сделать вывод о наличии функциональных зависимостей между не ключевыми атрибутами, но, изучив предметную область, можно сделать вывод, что `Тарифная ставка` зависит от `Тарифного разряда` и `Тарифной сетки`. А `Тарифная сетка` зависит от `Профессии`. Следовательно, отношение (таблицу) лучше разбить на три отношения (таблицы):

```
Профессии(Профессия ⇒ Тарифная сетка)

Тарифные ставки(Тарифный разряд, Тарифная сетка ⇒ Тарифная ставка)

Рабочие(Табельный номер ⇒ Фамилия, Профессия, Тарифный разряд)
```

Теперь, для того чтобы узнать `Тарифную ставку` для рабочего по его `Табельному номеру`, нам необходимо будет запросить `Тарифную сетку` из таблицы `Профессии`, и только после этого `Тарифную ставку` из таблицы `Тарифные ставки`.

Вы можете справедливо заметить, что выборка данных стала сложнее, зато упрощаются процедуры, связанные с добавлением, изменением и удалением данных. Например, при изменении тарифных ставок на предприятии достаточно поменять `Тарифные ставки` в таблице `Тарифные ставки`, которая содержит не более 20 строк, а не изменять данные для 20 тыс. строк в таблице `Рабочие`.

Процесс выделения зависимых полей (атрибутов) в отдельные таблицы (отношения) называется *нормализацией*. Цель нормализации — облегчить добавление, изменение и удаление данных. Функциональная зависимость — это только одно их проявлений зависимости, возникающей между данными.

Кроме этого, могут быть другие зависимости. Например, данные, которые могут быть определены при помощи выполнения вычислений над несколькими строками связанной таблицы (чаще всего суммирования):

```
Строка накладной(Номер накладной, Товар ⇒ Цена, Количество)
Накладная(Номер накладной ⇒ Сумма = Сумма(Цена x Количество)для всех
  Строк накладной)
```

Специально для работы с реляционными базами данных был разработан язык SQL. Этот язык разрабатывался максимально приближенным к естественному языку, и предполагалось, что запросы при помощи языка SQL будут составлять конечные пользователи баз данных, а не программисты. Оказалось, что работать с языком SQL конечным пользователям все же не так удобно, и приложения для работы с базами данных по-прежнему разрабатывают программисты. Но теперь они имеют в своих руках достаточно мощное и очень удобное средство — язык запросов SQL. Один запрос SQL может заменить не один десяток строк программного кода, хотя среди программистов давно наблюдается тенденция не использовать сложные запросы SQL, а заменять их программной обработкой данных, например, средствами языка программирования PHP 5, что негативно сказывается на производительности приложения.

Операторы языка SQL определены стандартами, из которых наиболее широко поддерживается стандарт SQL-92. Вот наиболее часто употребляемые операторы языка SQL:

```
CREATE TABLE имя_таблицы (список_полей);
INSERT INTO имя_таблицы (список_полей) VALUES (список_значений);
UPDATE имя_таблицы SET имя_поля=значение_поля WHERE условие_отбора;
DELETE FROM имя_таблицы WHERE условие_отбора;
SELECT список_полей FROM имя_таблицы WHERE условие_отбора;
```

Перечисленные операторы создают таблицу базы данных, вставляют, изменяют и удаляют записи базы данных, а так же делают выборку из базы данных. Эти операторы имеют особенности реализации для каждого сервера баз данных, но основной синтаксис поддерживается всеми без исключения серверами баз данных, поддерживающими стандарт SQL-92.

Условие отбора `WHERE` позволяет производить действия не со всеми записями таблицы, а только с определенной частью этих записей, для которых выражение в условии отбора принимает логическое значение истины (`TRUE`).

Оператором `SELECT` можно сделать выборку из нескольких таблиц базы данных. В этом случае после слова `FROM` список таблиц указывается через запятую:

```
SELECT список_полей FROM таблица1, таблица2, таблица3
```

Если не задать специально условие отбора или соединения, то вернется декартово произведение строк из всех перечисленных таблиц, т. е. каждая строка *таблицы1* будет соединена во всех возможных сочетаниях с каждой из строк *таблицы2* и *таблицы3*. Поэтому при выборке из двух и более таблиц одним оператором SELECT необходимо задавать условие отбора или условие соединения. Условие отбора задается так:

```
SELECT * FROM Рабочие, Профессии, Тарифные_ставки WHERE
  Рабочие.Профессия = Профессии.Прообфессия AND
  Рабочие.Тарифный_разряд = Тарифные_ставки.Тарифный_разряд AND
  Профессии.Тарифная_сетка = Тарифные_ставки.Тарифная_сетка
```

Мы использовали для примера таблицы, которые были рассмотрены в этом разделе, и условный (не MySQL) сервер баз данных. Знак * (звездочка) означает, что выбираются все поля из таблиц. Аналогичный запрос можно записать более красиво, воспользовавшись не условием отбора, а условием соединения:

```
SELECT * FROM Рабочие INNER JOIN Профессии ON
    Рабочие.Профессия = Профессии.Прообфессия
  INNER JOIN Тарифные_ставки ON
    Рабочие.Тарифный_разряд = Тарифные_ставки.Тарифный_разряд AND
    Профессии.Тарифная_сетка = Тарифные_ставки.Тарифная_сетка
```

Такая запись лучше по двум причинам. Во-первых, она свидетельствует о том, что между таблицами существуют связи, а не просто выполняется отбор записей по некоторому условию. И, во-вторых, потому что такой запрос может работать значительно быстрее, поскольку не формируется декартово произведение всех записей из трех таблиц. Современные серверы баз данных в состоянии оптимизировать запросы, так что время выполнения на реальном сервере баз данных может существенно не отличаться для запросов, заданных условиями отбора, в сравнении с запросами, заданными условием соединения.

И, наконец, обсудим специальное значение NULL, которое говорит о том, что у поля нет определенного значения. По умолчанию все поля таблицы базы данных могут принимать значение NULL. Для того чтобы запретить значения NULL, при определении таблицы в операторе CREATE TABLE необходимо задать для этого поля условие NOT NULL. Значения NULL усложняют работу с таблицами и могут приводить к неожиданным ошибкам. Например, любая арифметическая операция с полем, которое имеет значение NULL, будет так же иметь значение NULL. Считается, что два значения NULL не равны друг другу, поэтому записи со значениями NULL не попадут в выборку, если в условие отбора или соединения попадает поле NULL.

Как правило, поле или набор полей, которые принимают значение `NULL`, могут быть вынесены в отдельную таблицу. Поэтому приходится выбирать между созданием новой таблицы, связанной отношением "один-к-одному" с основной таблицей, или мириться с наличием значений `NULL` в полях таблицы. Например, отдел кадров может заносить в базу данных домашний телефон работника. Но у работника может не быть домашнего телефона, или он может не пожелать его указать в анкете. Для хранения номеров домашних телефонов может быть использована основная таблица:

```
Работники(ИД, Табельный_номер, Фамилия, ..., Домашний_телефон)
```

В этом случае поле `Домашний_телефон` может содержать значение `NULL`. Другой вариант — поле `Домашний_телефон` выносится в отдельную таблицу:

```
Работники(ИД, Табельный_номер, Фамилия, ...)
Домашние_телефоны(Работник_ИД, Домашний_телефон)
```

Выбор не так уж и прост. Скорее всего, будет выбрана схема данных, допускающая `NULL`-значения. Потому что вынесение всех `NULL`-значений в отдельные таблицы породило бы массу мелких таблиц, работать с которыми было бы еще сложнее, чем со значениями `NULL`. Просто подводные камни использования значений `NULL` необходимо учитывать при разработке структуры баз данных и программного обеспечения, работающего с базами данных.

Мы неформально познакомились с терминологией реляционных баз данных. В следующем разделе содержится краткое описание работы с сервером баз данных MySQL 5.

7.3. Основы работы с сервером баз данных MySQL

Подавляющее большинство серверов реляционных баз данных реализуют работу с данными при помощи языка SQL. Имеется несколько стандартов SQL, среди которых наиболее широко поддерживаемым является стандарт SQL-92. Сервер баз данных MySQL 5 поддерживает данный стандарт практически в полном объеме и предлагает ряд расширений, которые помогают разрабатывать более быстрые и надежные приложения.

Одной из существенных особенностей сервера баз данных MySQL (в частности, MySQL 5) является наличие двух моделей поддержки целостности данных — транзакционной и нетранзакционной ("атомарные операции"). Наличие двух моделей поддержки целостности данных породило ряд недоразумений вокруг сервера баз данных MySQL. Многие ошибочно полагают, что MySQL не поддерживает транзакций. Это не так. MySQL поддерживает и транзакционную

модель, и нетранзакционную модель ("атомарные операции"). Второе недоразумение связано с недостаточно четким представлением разработчиками прикладных программ особенностей работы нетранзакционной модели, что приводит к неэффективным решениям. В общем, документация по MySQL, которая написана в неформальной манере, с большим количеством примеров, должна помочь всем желающим разобраться с этим вопросом.

Первое, что необходимо сделать после установки сервера, — создать базу данных, в которой мы будем работать. База данных создается оператором:

```
CREATE DATABASE имя_базы_данных character set cp1251
```

У вас может возникнуть вопрос: как выполнить этот оператор? Для выполнения этого оператора и других операторов SQL можно воспользоваться консолью mysql (mysql.exe), которая поставляется вместе с сервером баз данных MySQL. Кроме того, вы можете найти и установить клиентское приложение с графическим оконным интерфейсом для работы с сервером MySQL (некоторые клиентские приложения распространяются бесплатно и доступны для скачивания из Интернета). Наконец, для выполнения операторов SQL и работы с сервером баз данных MySQL 5 мы разработаем приложение "Консоль базы данных MySQL 5" далее в этой главе.

Один из вариантов, который я рассматривал при написании данной главы, — начать с разработки приложения "Консоль базы данных", а только затем рассмотреть основы работы с сервером баз данных MySQL 5. В этом случае вы могли бы использовать уже сейчас наше приложение для работы с примерами. Но потом я понял, что программный код этого приложения без знания основ работы с сервером MySQL 5 может вызвать затруднения. Вы можете пока не искать средства для работы с сервером баз данных MySQL 5, а прочитать этот раздел в ознакомительном режиме, а затем, после создания приложения "Консоль базы данных MySQL 5", выполнить на практике все примеры раздела.

Мы будем работать с базой данных, которую назовем `bhvdb`. С учетом этого, оператор создания базы данных будет выглядеть так:

```
CREATE DATABASE bhvdb character set cp1251
```

Кодировка задается, как мы и условились в начале книги, Windows-1251 (cp1251).

Чтобы выполнить этот оператор, необходимо подключиться к системной базе данных сервера MySQL 5, которая имеет имя `mysql`, с правами суперпользователя `root`.

Для работы с нашей новой базой данных `bhvdb` нам надо создать нового пользователя, чтобы мы случайно не нарушили системные базы данных,

располагая правами суперпользователя `root`. Новый пользователь создается оператором:

```
CREATE USER 'имя' IDENTIFIED BY 'пароль'
```

Например:

```
CREATE USER 'bhvajax'@'localhost' IDENTIFIED BY 'sg2D5kvEqR8'
```

Теперь созданному пользователю необходимо предоставить права для работы с нашей новой базой данных:

```
GRANT ALL PRIVILEGES ON bhvdb.* TO 'bhvajax'@'localhost'
  WITH GRANT OPTION
```

Оператор `GRANT` выполняет назначение прав пользователя на некоторый объект. В данном случае это объект `bhvdb.*`, который означает все объекты базы данных `bhvdb`. На объект даются все права `ALL PRIVILEGES`. Права назначаются пользователю `bhvajax`, но только для доступа с локального компьютера:

```
'bhvajax'@'localhost'
```

Наконец, `WITH GRANT OPTION` означает, что пользователь для этой базы данных сам может выполнять оператор `GRANT` и назначать права другим пользователям. Мы обязательно воспользуемся этой возможностью. (`ALL PRIVILEGES` не включают `GRANT OPTION`, поэтому необходимо явно указывать эту возможность.)

Теперь пользователь `bhvajax` при доступе с локального компьютера (`localhost`) будет обладать всеми правами над объектами базы данных `bhvdb`. Особенность сервера баз данных MySQL заключается в том, что пользователи соотносятся не с базой данных, а с логином при доступе к серверу базы данных. Поэтому `ALL`-привилегии для базы данных не позволяют создавать нового пользователя. Для того чтобы пользователь `bhvajax` мог создавать новых пользователей, необходимо дать ему соответствующие права:

```
GRANT create user ON *.* TO 'bhvajax'@'localhost'
```

Обратите внимание, что пользователи создаются для всех баз данных, о чем говорит символьная маска `*.*`. В то же время, создав нового пользователя, пользователь `bhvajax` сможет назначить для него права только относительно объектов из базы данных `bhvdb`.

Итак, мы создали пользователя `bhvajax`, который для одной отдельно взятой базы данных `bhvdb` обладает привилегиями, предоставленными суперпользователю `root` для всех баз данных сервера. Точно так же, как и работать, подключившись с правами суперпользователя `root`, работать с правами `bhvajax` будет довольно опасно, поскольку этот пользователь обладает очень широкими полномочиями. Поэтому желательно создать пользователя с меньшим

набором прав, например, только для выполнения операторов SELECT по конкретной таблице, и даже по конкретным столбцам таблицы. Для этого можно воспользоваться правами только что созданного пользователя bhvajax и создать нового пользователя, например bhv. Подключитесь в качестве пользователя bhvajax к базе данных bhvdb и выполните следующий оператор SQL:

```
GRANT ALL ON bhvdb.* TO 'bhv'@'localhost' IDENTIFIED BY 'd57LU293gxz'
```

В этом примере создание пользователя совмещено с назначением этому пользователю привилегий ALL. Этот пользователь имеет все еще большие привилегии, сравнимые с привилегиями пользователя bhvajax и даже root. Но этот пользователь уже не может создавать новых пользователей и назначать привилегии для объектов базы данных bhvdb другим пользователям.

Для работы с базой данных из приложений, скорее всего, вы предпочтете создавать пользователей с довольно ограниченными правами. Например:

```
GRANT SELECT (kod, name) on bhvdb.products
    TO 'select_products'@'localhost' IDENTIFIED BY 'd2re9df54dX'
```

Такой оператор создает пользователя, который может только читать оператором SELECT два поля kod и name из таблицы products.

Научившись создавать новых пользователей, можно перейти к созданию таблиц. Новая таблица создается оператором CREATE TABLE, в котором задаются имя таблицы, имена и тип полей таблицы, первичный ключ, внешние ключи и некоторые другие параметры. Сервер MySQL 5 поддерживает впечатляющее количество типов данных, часть из которых вы вряд ли будете использовать в своих первых приложениях. С подмножеством типов данных, которые будут использоваться в примерах книги, вы можете познакомиться в табл. 7.1.

Таблица 7.1. *Типы данных MySQL 5, используемые в примерах книги*

Тип	Диапазон значений или точность	Примечания
BIGINT	–9 223 372 036 854 775 808.. 9 223 372 036 854 775 807	UNSIGNED BIGINT 0..18 446 744 073 709 551 615
NUMERIC(N, D)	N до 65 значащих цифр; D до 30 знаков после запятой	N по умолчанию 10; D по умолчанию 0
DATE	До '9999-12-31'	'YYYY-MM-DD'
DATETIME	До '9999-12-31 23:59:59'	'YYYY-MM-DD HH:MM:SS'

Таблица 7.1 *(окончание)*

Тип	Диапазон значений или точность	Примечания
`VARCHAR(length)`	До 65532 символа	
`CHAR(length)`	До 255 символов	
`BLOB`	`L`+2 байтов, где `L` < 2^{16}	Бинарные объекты

Примечание. С прочими типами данных вы можете познакомиться по документации: `BIT`, `TINYINT`, `SMALLINT`, `MEDIUMINT`, `INT`, `INTEGER`, `REAL`, `DOUBLE`, `FLOAT`, `DECIMAL`, `TIME`, `TIMESTAMP`, `YEAR`, `BINARY`, `VARBINARY`, `TINYBLOB`, `MEDIUMBLOB`, `LONGBLOB`, `TINYTEXT`, `TEXT`, `MEDIUMTEXT`, `LONGTEXT`, `ENUM`, `SET`.

Поддерживаются геометрические типы данных (для геоинформационных систем). Геометрические типы данных реализованы с использованием абстрактных типов данных и наследования. Некоторые типы геометрических данных, такие как Geometry и др., являются абстрактными (non-instatniable). Это означает, что нельзя создать экземпляр объекта (instance) типа Geometry. В противоположность этому можно создавать экземпляры объектов конкретных (instatiable) типов, например Point:

- `Geometry` (non-instantiable);
- `Point` (instantiable);
- `Curve` (non-instantiable);
- `LineString` (instantiable);
- `Line` (instantiable);
- `LinearRing` (instantiable);
- `Surface` (non-instantiable);
- `Polygon` (instantiable);
- `GeometryCollection` (instantiable);
- `MultiPoint` (instantiable);
- `MultiCurve` (non-instantiable);
- `MultiLineString` (instantiable);
- `MultiSurface` (non-instantiable);
- `MultiPolygon` (instantiable).

Теперь можно приступить к практическому применению операторов, создающих таблицы:

```
CREATE TABLE table0
  (key0 BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  field0 VARCHAR(65532),
  field1 DECIMAL(65,30))
```

Мы создали таблицу `table0` и определили для нее первичный ключ по полю `key0`. Это простой или атомарный первичный ключ. Если ключ составной, его необходимо определить следующим образом:

```
CREATE TABLE table1
  (key0 BIGINT NOT NULL,
  key1 BIGINT NOT NULL,
  field0 VARCHAR(65532),
  field1 DECIMAL(65,30),
  PRIMARY KEY (key0, key1))
```

Теперь создадим таблицы, которые будут использоваться в последующих разделах для тестирования Ajax-компонентов. Вспомним, что мы уже создали базу данных `bhvdb` и двух пользователей — `bhvajax` и `bhv`. Для создания таблиц нам достаточно полномочий пользователя `bhv`.

Первая наша тестовая таблица будет содержать список товаров или изделий предприятия — справочник *Номенклатура*. Обычно в таких справочниках предусматривается достаточно сложная структура, содержащая поля, по которой товары и изделия классифицируются, задаются соответствующие бухгалтерские счета и субсчета, сведения об изготовителе. В некоторых упрощенных системах этот справочник может содержать цену (*Номенклатура-ценник*). Все эти сложные и неоднозначные вопросы мы вынесем за скобки и рассмотрим ровно три поля, которые будем использовать для выбора номенклатуры из списка: *Код*, *Наименование*, *Краткое наименование для поиска*. *Код* будем генерировать полем `AUTO_INCREMENT`. В реальных системах код номенклатурной позиции может быть суррогатным ключом, произвольно формируемым автоматизированной системой, а может иметь естественное значение и формироваться в соответствии со стандартами предприятия, отрасли или государства. *Наименование* — это поле, которое соответствует официально утвержденному наименованию номенклатурной позиции. Наименование может быть довольно сложным для ввода с клавиатуры и содержать мелкие подробности — кавычки, запятые, точки, тире, что затрудняет поиск номенклатурной позиции в списке при выборе. Для организации удобного ввода пользователя служит поле *Краткое наименование для поиска*.

Это наименование, которое удобно вводить с клавиатуры. Для этого из него исключены мелкие подробности, которые невозможно запомнить пользователю. Например, для изделия `Изд.-100,01-80.88/4"` из справочника *Номенклатура* может быть задано *Краткое наименование для поиска*, равное `изд1000180884`, или еще проще: `1000180884`. В некоторых случаях такой подход позволяет существенно повысить скорость и точность ввода информации пользователем.

Наша таблица будет создаваться следующим оператором `CREATE TABLE`:

```
CREATE TABLE products (
  kod bigint(20) NOT NULL auto_increment,
  name varchar(100) default NULL,
  search char(100) default NULL,
  PRIMARY KEY  (`kod`)
);
```

Используя эту таблицу, мы будем тестировать наш компонент *Lookup Combobox* в *главе 9*. В *главе 10* мы создадим компонент для отображения и редактирования таблицы данных с использованием, в частности, и компонента Lookup Combobox. Для тестирования нам понадобится еще одна таблица, в которой хранятся данные, связанные с только что созданной нами таблицей `products`. Пусть это будут заказы клиентов:

```
CREATE TABLE orders_details (
  kod_order bigint(20) NOT NULL,
  kod_product bigint(20) NOT NULL,
  count decimal(10,0) default NULL,
  price decimal(20,4) default NULL,
  PRIMARY KEY  (kod_order, kod_product)
);
```

В этой таблице содержится детальная информация по заказу клиента. Заказ в ней представлен номером заказа — `kod_order`. Этот номер может быть естественным (натуральным), т. е. соответствовать номеру заказа клиента, который присутствует в документах, либо быть суррогатным и формироваться при помощи `AUTO_INCREMENT`. Соответственно, должна быть создана таблица, в которой отражается общая информация о заказе:

```
orders(kod, date, customer ...)
```

Мы эту таблицу назвали, но реально создавать не будем, поскольку создание этой таблицы потянет за собой необходимость создать таблицу заказчиков, менеджеров, ответственных за выполнение заказа, поступления средств на расчетный счет и т. п. И мы сможем остановиться, только воссоздав всю

структуру базы данных. Недаром наши базы данных называют реляционными. Для работы с примерами книги нам будет вполне достаточно двух созданных таблиц для тестирования Ajax-компонентов.

Для добавления данных в таблицу служит оператор `INSERT`:

```
INSERT INTO products (name, search) VALUES ('Товар №112', '112');
INSERT INTO orders_details (kod_order, kod_product, count, price)
  VALUES (12, 1, 10, 29.99)
```

Вы видите, что в таблице `products` мы не присваиваем значение полю `kod`, которое определено как `AUTO_INCREMENT`. Но вы можете и явно задавать значение этого поля (некоторые серверы баз данных это не допускают), в таком случае автонумерация продолжится со следующего доступного номера. В таблице `orders_details` мы явно задаем все поля. В таких случаях перечислять список полей не обязательно, и можно выполнить оператор SQL в сокращенном виде:

```
INSERT INTO orders_details VALUES (12, 1, 10, 29.99)
```

Я такой возможностью стараюсь не пользоваться, т. к. структура таблицы (количество и порядок следования полей) может изменяться, и это приведет к неожиданной ошибке, которую сложно будет обнаружить.

Данные из таблиц можно выбрать оператором `SELECT`:

```
SELECT * FROM products;
SELECT * FROM orders_details;
SELECT o.kod_order, p.name, o.count, o.price, o.count*o.price as total
  FROM orders_details o INNER JOIN products p
  ON o.kod_product = p.kod
```

Для работы с примерами книги я подготовил тестовые данные. Вы можете воспользоваться ими, взяв в папке \bhvajax\mysql\test_data.sql, либо заполнить таблицы самостоятельно своими данными.

Перейдем к созданию приложения "Консоль базы данных MySQL 5", при помощи которой можно будет выполнить все рассмотренные операторы SQL, если вы еще не сделали это при помощи других средств. Если это так, после создания этого приложения еще раз перечитайте материал данного раздела и выполните все примеры.

7.4. Создание программного кода приложения "Консоль базы данных MySQL 5"

Наше приложение будет состоять из двух основных файлов: console.html и ping_database.php. С общим видом приложения в окне Web-браузера вы можете

познакомиться на рис. 7.7. В поля HTML-документа console.html необходимо ввести имя хоста, на котором работает сервер баз данных MySQL 5, имя базы данных, имя пользователя и пароль. При первом обращении после инсталляции сервера баз данных в поле имени базы данных необходимо указать `mysql` (системная база данных), пользователя `root` (суперпользователь) и пароль, который был введен при установке сервера баз данных MySQL 5. Если вы при установке разрешили доступ к серверу пользователя `root` только с локального хоста, то можете в качестве хоста задать только `localhost`. При этом, разумеется, PHP 5 будет обращаться к локальному хосту, на котором работает Web-сервер Apache, а не к тому, на котором запущен Web-браузер.

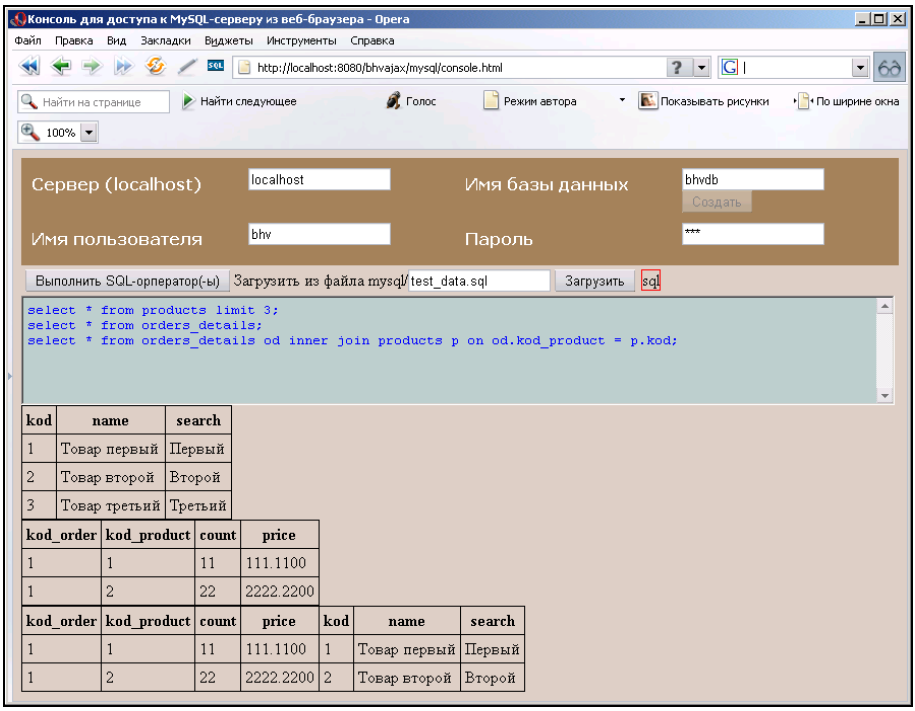


Рис. 7.7. Консоль базы данных MySQL в окне Web-браузера

По нажатию клавиш при вводе или корректировке пароля будет вызываться запрос `XMLHttpRequest`, который обращается к скрипту PHP 5 ping_database.php. В зависимости от введенных в HTML-элементы `INPUT` данных, будет дано разрешение на выполнение операторов SQL, создание новой базы данных и будут выполнены операторы SQL, передаваемые в качестве http-параметров запроса.

В этом разделе мы познакомимся с функциями PHP 5, которые могут быть использованы для работы с сервером баз данных MySQL 5. Но сначала рассмотрим HTML-документ и программный код JavaScript приложения "Консоль базы данных MySQL 5".

7.4.1. Программный код HTML и JavaScript приложения "Консоль базы данных MySQL 5"

Программный код документа HTML и JavaScript уже не должен вызывать у вас никаких вопросов. В файле console.html представлен код приложения.

Программный код JavaScript размещен непосредственно в HTML-документе. Я не выделил программный код JavaScript в отдельный js-файл, т. к. его функциональность тесно связана с конкретным приложением и с конкретным HTML-документом, поэтому нет необходимости повторно его использовать. Давайте рассмотрим основные моменты реализации Консоли.

Как всегда, в `HEAD` HTML-документа мы подключаем основную библиотеку bhv\util.js, в которой определена функция `bhv.sendRequest()`:

```
<script type="text/javascript" src="../bhv/util.js"></script>
```

Разметка HTML-элементов `INPUT`, в которые пользователь будет вводить имя хоста, базы данных, пользователя и пароль, совершенно идентична, поэтому рассмотрим ее на примере одного из них. В этот HTML-элемент `INPUT` пользователь будет вводить имя хоста, на котором работает сервер баз данных MySQL:

```
<input type="text" size="20" id="host" value="localhost"
  tabindex="1" onkeyup="pingDatabase(3000)">
```

В качестве значения по умолчанию задается значение `localhost`. При отпускании клавиши вызывается функция:

```
pingDatabase(3000)
```

В первом параметре функции `pingDatabase()` передается задержка выполнения Ajax-запроса к Web-серверу в миллисекундах (т. е. задано три секунды). Почему в этом случае необходимо вводить задержку отправки Ajax-запроса, я сейчас постараюсь объяснить.

В классическом Web-приложении пользователь сначала заполняет все поля ввода `INPUT`, а затем при помощи кнопки `SUBMIT` отправляет введенные данные на сервер. Такой процесс не может быть признан удобным для пользователя, поскольку полей может быть много, при вводе пользователь может допускать ошибки. В общем, все, кто регистрировался на сайтах в Интернете, может восстановить в своей памяти массу неудачных решений ввода инфор-

мации. Это привело даже к распространению ошибочного мнения, что Web-интерфейс не может быть удобным для ввода информации. На самом деле это не так. И я неоднократно убеждался, что Web-интерфейс может быть не только сравним с обычным оконным интерфейсом, но даже превосходить оконный интерфейс по удобству с точки зрения пользователя. Давайте подумаем, как можно улучшить стандартное решение и не заставлять пользователя нажимать лишние кнопки, которые часто имеют невыразительные названия. Один из вариантов — отправлять Ajax-запросы к серверу, обрабатывая нажатия клавиш в HTML-элементе `INPUT`, и проверять данные на правильность в фоновом режиме, предлагать варианты для выбора из списка по первым введенным буквам. В общем, это то, с чего начиналась технология Ajax.

В нашем приложении "Консоль базы данных MySQL 5", для подключения к базе данных необходимо ввести в HTML-элементы `INPUT` имя хоста, базы данных, пользователя и пароль. Можно было бы предусмотреть отдельную кнопку **Подключиться**, при нажатии которой введенные данные отправлялись бы на сервер, после чего пользователю может быть разрешено выполнение запросов к базе данных или выдано сообщение об ошибке подключения. Но я предлагаю реализовать обработку ввода данных по-другому. Пусть запрос на сервер отправляется без дополнительных усилий со стороны пользователя, после того, как пользователь введет всю необходимую информацию для подключения. Для этого мы будем использовать обработчики нажатия клавиш, а если быть совсем точным — событие `onkeyup`. Именно это событие следует использовать потому, что к моменту его вызова последний введенный символ уже доступен из атрибута `value` HTML-элемента `INPUT`.

Давайте разберем положительные моменты такого решения. Пользователь должен просто ввести правильные имена хоста, базы данных, пользователя и пароль. Для проверки введенных данных на сервере и подключения не следует нажимать никаких дополнительных элементов ввода. Пользователь не может направить запрос к серверу, пока не введет правильные данные для соединения. На самом деле, ввод операторов SQL у пользователя может отнять много времени и энергии. И если в результате он получит сообщение о неправильно введенном пароле вместо ожидаемых результатов, да еще и в довершение пустой элемент `INPUT`, в который он последние полчаса вводил текст запроса — пользователь, как минимум, перестанет пользоваться таким средством доступа к базе данных.

Но есть у предлагаемого решения и отрицательные стороны. Рассмотрим их более подробно, потому что Ajax-запрос очень часто вызывается для обработки нажатия клавиш при вводе в HTML-элементы `INPUT`. Имеется в виду, что при вводе произвольного слова, например `localhost`, нам необходимо

вызвать функцию только после ввода всего слова. В противном случае на сервер будут выданы запросы для каждой буквы:

```
l
lo
loc
loca
```

и т. д.

Сервер загружается дополнительными запросами, увеличивается сетевой трафик. Но не только это является отрицательным моментом. На самом деле, в Интернете запросы могут обгонять друг друга. То есть отправляться будут запросы в порядке нажатия клавиш, а вот порядок вызова функций для обработки ответов Web-сервера может быть произвольным, например:

```
lo
l
loca
loc
```

и т. д.

Поэтому последний вызов обработчика ответа Web-сервера обработает не все слово полностью, а только некоторую его часть. Для преодоления такого недостатка можно, конечно, вообще исключить вызов обработчика для события `onkeyup` и создать кнопку для запуска запроса к Web-серверу или делать такой запуск при нажатии клавиши <Enter>. Но такой вариант сильно повлиял бы на удобство пользователя при работе с приложением. Поэтому для уменьшения сетевого трафика и обеспечения более вероятного соответствия порядка отправленных и полученных запросов, используют задержку отправки запроса на некоторый не очень большой интервал времени — 1—5 секунд. При этом очередное нажатие клавиши отменяет выполнение предыдущего запроса, который еще не был отправлен и находится в состоянии ожидания (если время задержки отправки еще не истекло). В *главе 9* мы специально для этого реализуем утилитную функцию, а сейчас ограничимся самым простым решением:

```
window.clearTimeout(pingHandle);

pingHandle = window.setTimeout(function() {
  bhv.sendRequest("post", "ping_database.php",
    "host=" + host.value + "&username=" + username.value
      +"&password="+encodeURIComponent(password.value)
```

```
      +"&database="+database.value+action,
    true, innerPing)}, delay);
```

В глобальной переменной `pingHandle` мы запоминаем идентификатор запроса, который возвращает функция `window.setTimeout()`, и в случае повторного выполнения функции `pingDatabase()` отменяем предыдущий вызов при помощи вызова функции `window.clearTimeout(pingHandle)`. В данном случае использование глобальной переменной не приведет к недоразумениям, поскольку вызывается всего одна функция `pingDatabase()`. Мы решим эту задачу без использования глобальных переменных и для произвольного количества запросов в *главе 9*.

В дополнение к установлению задержки мы проверяем условие:

```
if (! host.value || ! database.value || ! username.value)
  return;
```

То есть если хотя бы одно из полей ввода, кроме пароля, останется пустым — запрос не отправляется.

Функцию `pingDatabase()` мы будем вызывать и для проверки подключения к серверу базы данных, и для выполнения операторов SQL. Для большей гибкости, в функции `pingDatabase()` необходимо предусмотреть второй параметр, который говорит о типе выполняемой операции:

```
pingDatabase(0, "execute");
pingDatabase(0, "create");
```

Для операций `execute` и `create` задержка выполнения равна нулю, поскольку функция `pingDatabase()` с этими параметрами будет вызываться не обработчиками `onkeyup` HTML-элементов `INPUT type=text` (текстовых полей ввода), а кнопками:

```
<input type="button" id="execute" value="Выполнить SQL-орператор(-ы)"
  onclick="pingDatabase(0, 'execute')"
  tabindex="9999" disabled="disabled">
<input type="button" id="create" value="Создать" disabled="disabled"
  tabindex="9999" onclick="pingDatabase(0,'create')">
```

Мы выделяем отдельно оператор создания базы данных, хотя это такой же оператор SQL. Я выделил этот оператор в отдельную команду, чтобы создание новой базы данных гарантированно производилось командой:

```
CREATE DATABASE if not exists имя_базы character set cp1251
```

В этом случае, как мы и договаривались, в нашей книге будет последовательно соблюдаться работа с кодировкой Windows-1251.

Теперь мы можем рассмотреть собственно код функции `pingDatabase()` (фрагмент, в котором определена внутренняя функция `innerPing()`, пропущен и будет рассмотрен далее в тексте раздела):

```
function pingDatabase(delay, action) {

  var host = document.getElementById("host");
  var username = document.getElementById("username");
  var password = document.getElementById("password");
  var database = document.getElementById("database");
  var create = document.getElementById("create");
  var execute = document.getElementById("execute");
  var command = document.getElementById("command");
  var output = document.getElementById("output");
  var error = document.getElementById("error");
  var panel = document.getElementById("panel");

  if (! host.value || ! database.value || ! username.value)
    return;

  output.innerHTML = "";
  error.innerHTML = "";
  error.style.display = "none";

  if (action == 'create')
    action = "&create=yes";
  else if (action == 'execute'){
    action = "&execute=" + encodeURIComponent(command.value);
    var element = document.createElement("span");
    element.innerHTML = "sql";
    element.title = command.value;
    element.style.border="red solid 1px";
    element.style.margin="2px";
    element.onclick = function(){command.value = this.title;};
    panel.appendChild(element);
    panel.appendChild(document.createTextNode(" "));
  }else
    action = "";

  function innerPing(){ // begin inner function
```

```
//  Код внутренней функции пропущен
...
  }// end inner function

  window.clearTimeout(pingHandle);

  pingHandle = window.setTimeout(function() {
  bhv.sendRequest("post", "ping_database.php",
    "host=" + host.value + "&username=" + username.value
      +"&password="+encodeURIComponent(password.value)
      +"&database="+database.value+action,
    true, innerPing)},delay);
}
```

В первых строчках функции мы ищем HTML-элементы, с которыми нам придется работать, и запоминаем их в локальных переменных:

```
  var host = document.getElementById("host");
...
```

Это позволит удобно обращаться к HTML-элементам, а также во внутренней функции `innerPing()`, которая определена в теле функции `pingDatabase()`, обращаться к этим переменным, используя замыкания JavaScript. Обратите внимание на одну подробность. Имена локальных переменных и атрибуты `ID` HTML-элементов `INPUT` совпадают. Если бы мы попытались использовать одноименные с `ID` HTML-элементов глобальные переменные, это могло бы привести в некоторых Web-браузерах к неожиданной и сложно выявляемой ошибке. Дело в том, что некоторые Web-браузеры для облегчения доступа из JavaScript к HTML-элементам создают глобальные переменные, одноименные с атрибутом ID HTML-элемента. В настоящее время эту возможность используют редко. Для доступа к HTML-элементам по идентификатору ID используют определенную спецификацией DOM Level 1 функцию `document.getElementById()`. Для совместимости с написанным кодом, по-прежнему Web-браузером создаются и одноименные глобальные переменные. Если вы переопределите такую глобальную переменную, код JavaScript начинает работать с ошибками. Но мы, подчеркиваю, определили одноименные с атрибутами `ID` переменные не на глобальном уровне, а на локальном уровне (в теле функции `pingDatabase()`) и это не приводит к ошибке времени выполнения.

Далее, мы проверяем, заполнены ли необходимые реквизиты, и в случае отсутствия хотя бы одного из них прерываем выполнение функции:

```
if (! host.value || ! database.value || ! username.value)
  return;
```

В следующих строчках программного кода мы очищаем HTML-элементы, в которых будет отображаться результат выполнения операторов SQL:

```
output.innerHTML = "";
error.innerHTML = "";
error.style.display = "none";
```

Затем, мы анализируем второй параметр `action` функции `pingDatabase()`, который может быть пустым, если функция вызвана с одним параметром, а также иметь строковое значение `"create"` или `"execute"`:

```
if (action == 'create')
  action = "&create=yes";
else if (action == 'execute'){
  action = "&execute=" + encodeURIComponent(command.value);
  var element = document.createElement("span");
  element.innerHTML = "sql";
  element.title = command.value;
  element.style.border="red solid 1px";
  element.style.margin="2px";
  element.onclick = function(){command.value = this.title;};
  panel.appendChild(element);
  panel.appendChild(document.createTextNode(" "));
}else
  action = "";
```

Если параметр `action` равен `"create"`, мы формируем фрагмент строки запроса:

```
action = "&create=yes";
```

Если формальный параметр `action` равен `"execute"`, строка запроса будет содержать параметр `execute`, который в качестве значения будет иметь перекодированную по правилам MIME типа `application/x-www-form-urlencoded` строку операторов SQL:

```
action = "&execute=" + encodeURIComponent(command.value);
```

После этого обеспечивается минимальное удобство для пользователя Консоли. Текст операторов SQL будет сохранен в атрибуте `title` вновь создаваемого HTML-элемента `SPAN`. Этот элемент будет добавлен на панель Консоли и выглядеть как текст **sql**, взятый в красную рамку. При наведении мыши на этот HTML-элемент `SPAN` будет показан текст операторов SQL во всплывающем окне (это стандартное поведение для элементов, имеющих атрибут

title). При щелчке кнопкой мыши по элементу SPAN в HTML-элемент TEXTAREA будет перенесен текст операторов SQL:

```
var element = document.createElement("span");
element.innerHTML = "sql";
element.title = command.value;
element.style.border="red solid 1px";
element.style.margin="2px";
element.onclick = function(){command.value = this.title;};
panel.appendChild(element);
panel.appendChild(document.createTextNode(" "));
```

Далее, в теле функции pingDatabase() определяется внутренняя функция innerPing(), которая будет передана в функцию sendRequest() в качестве обработчика ответа Web-сервера. Код этой функции мы рассмотрим несколько позже. И, наконец, вызывается функция bhv.sendRequest():

```
window.clearTimeout(pingHandle);


pingHandle = window.setTimeout(function() {
bhv.sendRequest("post", "ping_database.php",
  "host=" + host.value + "&username=" + username.value
    +"&password="+encodeURIComponent(password.value)
    +"&database="+database.value+action,
  true, innerPing)}, delay);
```

Функция sendRequest() вызывается не сразу, а при помощи функции window.setTimeout(). Вызов функции window.setTimeout() имеет некоторые особенности, связанные с контекстом выполнения данной функции. Дело в том, что на момент вызова функции window.setTimeout(), функция pingDatabase() уже отработает и локальный контекст функции pingDatabase() будет недоступен по имени. Поэтому мы создаем новую анонимную функцию, внутри которой вызываем функцию sendRequest() и внутри которой будут доступны локальные переменные host, database, username, password, формальный параметр action и внутренняя функция innerPing(). Этот же набор локальных переменных и формальных параметров будет доступен и в теле функции innerPing(), когда эта функция будет вызвана в качестве обработчика ответа Web-сервера.

Прежде чем перейти к рассмотрению функции innerPing(), давайте подумаем, в каком виде мы будем получать данные с Web-сервера? Основных вариантов два: получать данные в виде XML-документа или в виде JSON-объекта. Я предлагаю использовать XML-документ. Мы не рассматривали

в книге язык определения схемы XML-документа, но в простейших случаях достаточно просто рассмотреть конкретный экземпляр XML-документа с тестовыми данными для того, чтобы понять структуру XML-документа. Образец XML-документа приведен в листинге 7.1.

Листинг 7.1. XML-документ для обмена данными в приложении Консоль MySQL

```
<?xml version="1.0" encoding="Windows-1251"?>

<response>

  <databaselist>
    <database>information_schema</database>
    <database>bhvajax</database>
    <database>bhvdb</database>
  </databaselist>

  <nocreate/>
  <execute/>

  <results>

    <result>
      <names>
        <name>kod</name>
        <name>name</name>
        <name>search</name>
      </names>
      <rows>
        <row>
          <field>1</field>
          <field>Товар первый</field>
          <field>Первый</field>
        </row>
        <row>
           <field>2</field>
           <field>Товар второй</field>
           <field>Второй</field>
         </row>
```

```
        <row>
          <field>3</field>
          <field>Товар третий</field>
          <field>Третий</field>
        </row>
        <row>
          <field>4</field>
          <field>Товар четвертый</field>
          <field>Четвертый</field>
        </row>
      </rows>
    </result>

    <result>
      <names>
        <name>kod_order</name>
        <name>kod_product</name>
        <name>count</name>
        <name>price</name>
      </names>
      <rows>
        <row>
          <field>1</field>
          <field>1</field>
          <field>11</field>
          <field>111.1100</field>
        </row>
        <row>
          <field>1</field>
          <field>2</field>
          <field>22</field>
          <field>2222.2200</field>
        </row>
      </rows>
    </result>

  </results>

</response>
```

Документ начинается с объявления XML-документа, в котором задана версия 1.0 и кодировка Windows-1251. Замечу, что это не избавляет от необходимости явно задавать в http-заголовках MIME-тип ответа:

```
header('Content-Type: application/xml; charset="windows-1251"');
```

Разумеется, задать явно `Content-Type` можно не только в программном коде PHP 5, но и в конфигурационных параметрах Web-сервера, каталога.

Имя документа — `response`, что следует из открывающего и закрывающего тегов XML-документа `<response>` и `</response>`. Далее следует элемент `databaselist`, содержащий элементы `database`, в которых находятся имена баз данных, доступных на сервере текущему пользователю. Я предусмотрел в XML-документе наличие элемента `databaselist`, т. к. планировал на основании анализа списка разрешать или запрещать пользователю выполнять операторы SQL к конкретной базе данных или создание конкретной базы данных. Как оказалось при дальнейшей разработке, этот список не был востребован, т. к. проверку удобнее осуществить на сервере. Но я оставил схему XML-документа как есть, чтобы не было никакой подтасовки. Кроме того, на самом деле этот список можно использовать для реализации выбора баз данных из списка. Вы можете в качестве домашнего задания реализовать такую функциональность после изучения материала главы.

Для того чтобы определить, какие операции разрешены пользователю, служат четыре одиночных тега, два из которых вы можете найти в листинге 7.1:

```
<nocreate/>
<execute/>
```

Наличие этих тегов в XML-документе означает, что пользователю разрешено выполнять операторы SQL для базы данных и запрещено создавать базу данных с заданным именем, поскольку такая база уже существует. Еще возможные варианты сочетаний элементов:

```
<nocreate/>
<noexecute/>
```

Такое сочетание означает, что пользователь не может выполнять никаких действий, поскольку или он не является допустимым пользователем, или неправильно введен пароль.

Следующее сочетание означает, что пользователь ввел правильное имя пользователя и пароль и может попытаться создать новую базу данных, имя которой указано в запросе:

```
<create/>
<noexecute/>
```

И, наконец, сочетание, которое не может появиться в XML-документе:

```
<create/>
<execute/>
```

Поясню, в чем тут дело. Если пользователю разрешено выполнить операторы SQL для базы данных, указанной в параметре `database`, значит, эта база данных существует, поэтому пытаться создавать базу данных с заданным именем не имеет смысла.

Наконец, собственно ответ сервера MySQL содержится в теле элемента `results` между тегами `<results>` и `</results>`. Как свидетельствует само название элемента, наборов данных с результатами может быть несколько. В PHP 5 реализована новая библиотека mysqli, которая позволяет в одном запросе к серверу выполнять несколько операторов SQL, и получать в качестве ответа несколько наборов данных с результатами. Такая возможность, вероятно, не будет часто использоваться в прикладных программах, но для создания "Консоли базы данных MySQL 5" она просто необходима. В связи с возможностью получить несколько наборов данных с результатами, элемент `results` может содержать ни одного, один или более одного элемента `result`, в которых будут находиться собственно данные, полученные с сервера баз данных MySQL.

Каждый элемент `result` содержит два элемента — `names` и `rows`, в которых хранятся наименования полей данных и строки, содержащие данные.

Элемент `names` содержит заголовки наименования полей:

```
<names>
  <name>kod</name>
  <name>name</name>
  <name>search</name>
</names>
```

В этом порядке поля будут следовать в теле элемента `row`:

```
<rows>
  <row>
    <field>1</field>
    <field>Товар первый</field>
    <field>Первый</field>
  </row>
  <row>
    <field>2</field>
    <field>Товар второй</field>
```

```
    <field>Второй</field>
  </row>
  <row>
    <field>3</field>
    <field>Товар третий</field>
    <field>Третий</field>
  </row>
  <row>
    <field>4</field>
    <field>Товар четвертый</field>
    <field>Четвертый</field>
  </row>
</rows>
```

Может быть, вам такая структура покажется не совсем правильной, и вы предпочтете работать с другой структурой XML-документа:

```
<row>
  <kod>1</kod>
  <name>Товар первый</name>
  <search>Первый</search>
</row>
```

Или вам больше по душе такая схема:

```
<row>
  <field name="kod">1</field>
  <field name="name">Товар первый</field>
  <field name="search">Первый</field>
</row>
```

Или даже такая:

```
<row>
  <field>
    <name>kod</name><value>1</value>
  </field>
  <field>
    <name>name</name><value>Товар первый</value>
  </field>
  <field>
    <name>search</name><value>Первый</value>
  </field>
</row>
```

В данном случае я исходил из того, что мне придется только выводить результирующие данные в виде HTML-таблицы, поэтому я выбрал самый простой вариант реализации XML-документа.

Теперь, разобравшись со схемой XML-документа, мы можем обсудить работу функции обработки ответа Web-сервера, содержащего это XML-документ, т. е. функции `innerPing()`. Эта функция является внутренней для функции `pingDatabase()`. С полным кодом этой функции вы можете ознакомиться в файле console.html, поэтому отдельно листинг функции `innerPing()` я приводить не буду и ограничусь фрагментами кода.

Ответ Web-сервера в виде XML-документа содержится в свойстве `responseXML` объекта `XMLHttpRequest`. Это свойство в теле функции `innerPing()` доступно как `this.responseXML`, поскольку в теле функции `bhv.sendRequest()` функция `innerPing()` вызывается в контексте объекта запроса `XMLHttpRequest`. (Реализация функции `bhv.sendRequest()` подробно описана в *главе 2*.) Для того чтобы работать с XML-документом (объектом DOM) было удобно, ссылку на него сохраним в локальной переменной `dom`:

```
var dom = this.responseXML;
```

Это позволит не только сократить программный код, но и уменьшить "на одну точку" доступ к свойствам объекта, что ускорит выполнение скрипта. Сравните:

```
this.responseXML.getElementsByTagName("execute");
dom.getElementsByTagName("execute");
```

Для правильного восприятия дальнейшего кода следует иметь в виду, что в теле внутренней функции `innerPing()` доступны, благодаря действию замыкания JavaScript, локальные переменные охватывающей функции `pingDatabase()`. Перечислим их:

```
var username = document.getElementById("username");
var password = document.getElementById("password");
var database = document.getElementById("database");
var create = document.getElementById("create");
var execute = document.getElementById("execute");
var command = document.getElementById("command");
var output = document.getElementById("output");
var error = document.getElementById("error");
var panel = document.getElementById("panel");
```

Эти переменные хранят ссылки на HTML-элементы документа console.html.

Сначала проверяем возможность создания новой базы данных:

```
if (dom.getElementsByTagName("create").length > 0) {
  create.disabled = false;
  execute.disabled = true;
}else
  create.disabled = true;
```

Как мы уже говорили, сочетание в XML-документе одновременно элементов:

```
<create/>
<execute/>
```

исключается. Поэтому при установке доступным HTML-элемента `<INPUT id="create">` можно сразу сделать недоступным элемент `<INPUT id="execute">`.

Аналогично проверяем возможность выполнения операторов SQL:

```
if (dom.getElementsByTagName("execute").length > 0) {
  execute.disabled = false;
  create.disabled = true;
}else
  execute.disabled = true;
```

Далее, ищем в XML-документе сообщения об ошибках, и если находим — выводим в специально предназначенный для этого HTML-элемент `DIV` сообщения об ошибках:

```
var errors = dom.getElementsByTagName("error");
error.innerHTML= "";
for (var i = 0; i < errors.length; i++) {
  error.innerHTML += "Ошибка: "
   + dom.getElementsByTagName("error")[0].firstChild.data + "<br />";
  error.style.display = "block";
}
```

В нашем XML-документе отсутствуют сообщения об ошибках. Поэтому приведем пример, как это сообщение будет выглядеть. Сообщение об ошибках может содержаться в любом месте документа в теле элемента `error`, например:

```
<error>
  Ошибка: Неверное имя пользователя
</error>
```

В общем, подготовительная работа проведена, и теперь можно обработать результат выполнения операторов SQL, если таковой будет присутствовать в XML-документе:

```
var result = dom.getElementsByTagName("result");
var allTables = ""
```

```
for (var r = 0; r < result.length; r++) {
  var tableHead = "<thead><tr>";
  var tableBody = "<tbody>";
  var names = result[r].getElementsByTagName("names")[0].childNodes;
  for (var i = 0; i < names.length; i++)
    tableHead += "<th>" + names[i].firstChild.data + "</th>";
  tableHead += "</tr></thead>";
  var rows = result[r].getElementsByTagName("rows")[0].childNodes;
  for (var i = 0; i < rows.length; i++) {
    var fields = rows[i].getElementsByTagName("field");
    tableBody += "<tr>";
    for (var j = 0; j < fields.length; j++) {
      if (fields[j].firstChild)
        tableBody += "<td>" +  fields[j].firstChild.data  + "</td>";
      else
        tableBody += "<td>" +  "<i>Empty</i>"  + "</td>";
    }// for j
    tableBody += "</tr>";
  }// for i
  tableBody += "</tbody>";
  var allTables = allTables + "<table  cellspacing='0'>"
        + tableHead + tableBody+ "</table>";
}// for r
output.innerHTML = allTables;
```

Этот фрагмент кода является простым перебором в цикле всех элементов `result`:

```
var result = dom.getElementsByTagName("result");
var allTables = ""
for (var r = 0; r < result.length; r++) {
...
```

В теле внешнего цикла перебираются сначала все имена полей результата:

```
var names = result[r].getElementsByTagName("names")[0].childNodes;
for (var i = 0; i < names.length; i++)
...
```

А затем все строки результирующего набора данных:

```
var rows = result[r].getElementsByTagName("rows")[0].childNodes;
for (var i = 0; i < rows.length; i++) {
...
```

И, наконец, для каждой строки результирующего набора данных перебираются поля результатов, которые и содержат данные:

```
var fields = rows[i].getElementsByTagName("field");
tableBody += "<tr>";
for (var j = 0; j < fields.length; j++) {
  if (fields[j].firstChild)
    tableBody += "<td>" +  fields[j].firstChild.data  + "</td>";
  else
    tableBody += "<td>" +  "<i>Empty</i>"  + "</td>";
}// for j
```

Как вы помните по *главе 4*, текстовые данные из тела XML-элемента необходимо извлекать достаточно громоздкой конструкцией:

```
fields[j].firstChild.data
```

Для полей данных, имеющих значение `NULL` (т. е. фактически, не имеющих значений), элемент будет иметь пустое свойство `firstChild` (это будет обеспечиваться серверным кодом). Для таких полей выведем в результирующую HTML-таблицу значение:

```
<i>Empty</i>
```

которое будет отображено наклонным шрифтом, что позволит легко отличить их от непустых элементов, содержащих строковое значение, равное `"Empty"`.

HTML-таблица в виде строки JavaScript формируется в локальной переменной `allTables`, и после полного формирования присваивается свойству `innerHTML` HTML-элемента `DIV`:

```
output.innerHTML = allTables;
```

На этом обработка запроса заканчивается, и наша Консоль переходит в состояние ожидания нового ввода пользователя.

В заключение рассмотрим один очень простой вопрос, связанный с загрузкой текста скрипта в "Консоль базы данных MySQL 5". В папке \bhvajax\mysql на прилагаемом к книге компакт-диске вы можете найти файл test_data.sql, который содержит операторы SQL, создающие в базе данных таблицы для тестирования наших Ajax-компонентов и заполняющие эти таблицы тестовыми данными. Я думал, как сделать процесс создания тестовых данных более наглядным для читателя, и понял, что лучше всего, если текст операторов SQL будет загружен в Консоль и выполнен примерно так же, как и операторы SQL, введенные с клавиатуры пользователем. Этот вариант не является допустимым для загрузки данных промышленных системах. Для промышленной

загрузки больших объемов данных серверы предоставляют особые операторы, которые оптимизированы специально для массовой загрузки новых данных. Но это является отдельной технологией, которая не связана с разрабатываемой нами Консолью. На этом я заостряю ваше внимание, чтобы вы не пытались загрузить через Консоль большие объемы данных. Это у вас не получится, потому что время выполнения скрипта PHP 5 на сервере ограничено, и большие объемы данных не могут быть обработаны таким скриптом, если специально не установить время выполнения скрипта на порядок выше обычного. Кроме того, Web-сервер ограничивает объем передаваемых на сервер данных в теле http-запроса. Но наших тестовых данных эти ограничения не касаются, потому что объем наших данных будет минимальным.

Вот текст с определением HTML-элементов, которые будут задействованы для загрузки файла, содержащего операторы SQL, в Консоль:

```
Загрузить из файла mysql/
<input type=text id="file" value="test_data.sql" size="20">
<input type="button" value="Загрузить" onclick="loadFile()">
```

Имя файла будет браться из HTML-элемента `INPUT id="file"`, для которого предусмотрено значение по умолчанию `"test_data.sql"`. Вы можете создавать и свои небольшие файлы с операторами SQL и загружать их в Консоль, вводя имя файла в HTML-элемент `INPUT`. Все пути задаются относительно текущего каталога mysql (bhvajax\mysql). При нажатии кнопки, расположенной рядом с полем ввода, вызывается функция `loadFile()`. Эта функция выполняет Ajax-запрос к файлу, заданному в поле ввода, и присваивает полученный текст файла HTML-элементу `command`:

```
function loadFile(delay, action) {
  var command = document.getElementById("command");
  var file = document.getElementById("file");
  bhv.sendRequest("post", file.value, null, true,
      function(){command.value = this.responseText;} );
}
```

Я уверен, что вы не раз воспользуетесь возможностью этой функции для выполнения операторов SQL.

7.4.2. Программный код PHP 5 приложения "Консоль базы данных MySQL 5"

В PHP 5 для работы с базами данных MySQL можно воспользоваться модулем mysql, который реализует функциональность, знакомую всем по PHP 4. Можно воспользоваться новым модулем mysqli, с которым мы и будем рабо-

тать в данной книге. Кроме того, в PHP 5 реализован единый для различных серверов баз данных модуль PDO, который позволяет единообразно работать с различными серверами баз данных, изменяя только строку подключения к серверу.

Я некоторое время выбирал между этими тремя возможностями работать с сервером баз данных MySQL в PHP 5 и не мог выбрать лучший вариант для использования в книге, чтобы соблюсти интересы уважаемых читателей.

Использование PDO показалось мне наиболее заманчивой идеей, поскольку такой программный код можно легко адаптировать для работы с серверами баз данных достаточно широкого круга. Я так и разработал несколько компонентов, пользуясь параллельно базами данных PostgreSQL и MySQL. Но неожиданно для меня оказалось, что сборщик системы Linux, на которой я тестировал свои примеры, не скомпилировал соответствующие модули. Это не является большой проблемой для специалиста, но может стать непреодолимой преградой для изучения книги начинающими программистами. Таким образом, от использования модуля PDO я отказался. Далее, меня смущал тот факт, что в период написания книги некоторые провайдеры попрежнему поддерживали только MySQL 4 и PHP 4. Учитывая это, я колебался в выборе и готов был уже использовать модуль mysql. Но потом понял, что мои читатели, скорее всего, предпочтут использовать самые современные технологии. И с учетом того, что Web-сервер Apache, сервер баз данных MySQL 5 и язык программирования PHP 5 любой, даже неподготовленный читатель, может без проблем установить на свой домашний компьютер — решил использовать для работы с сервером баз данных MySQL 5 в примерах книги модуль mysqli.

Одним из новшеств, которое поддерживает модуль mysqli, является возможность выполнять несколько операторов SQL в одном запросе и получать несколько результирующих наборов данных. Для обычных приложений эта возможность не будет, скорее всего, использоваться вами часто. Но для приложения "Консоль базы данных MySQL 5" такая возможность позволяет расширить функциональность приложения.

При работе с базой данных MySQL 5 в PHP 5 работают с такими объектами, как соединение с базой данных, запрос к базе данных, подготовленный запрос к базе данных, набор данных, строка набора данных. Запрос к базе данных и формирование соответствующего XML-документа с результатами являются достаточно простыми операциями, которые необходимо методично выполнить. Код обработчика Ajax-запроса к Web-серверу приведен в файле ping_database.php.

Первое, что необходимо сделать — установить правильные заголовки `Content-Type`:

```
header('Content-Type: application/xml; charset="windows-1251"');
```

Устанавливать заголовки необходимо до вывода первого символа документа. Устанавливать `charset` из PHP 5 при помощи функции `header()` не всегда может быть признано хорошей практикой. Например, ваше приложение может в перспективе работать с другими национальными кодировками или с UTF-8. В этом случае желательно управлять кодировками при помощи конфигурационных параметров Web-сервера или каталога Web-сервера. Но я не могу быть уверен, что все мои читатели будут иметь доступ к конфигурации Web-сервера, и поэтому задаю кодировку явно в заголовке `Content-Type`.

После установки заголовка я определяю функцию-обработчик ошибок:

```
function eh($errorcode, $message, $file, $line)
```

И устанавливаю эту функцию в качестве основного обработчика ошибок:

```
set_error_handler("eh");
error_reporting(E_ALL+E_STRICT);
```

В случае ошибки эта функция выведет в результирующий XML-документ элемент `error`:

```
echo "<error>Error: #$errorcode - $message in $file line $line</error>";
```

Далее мы присваиваем переменным значения параметров запроса:

```
$host = $_REQUEST['host'];
$database = $_REQUEST['database'];
$username = $_REQUEST['username'];
$password = $_REQUEST['password'];
```

В зависимости от настроек сервера баз данных MySQL подключиться к серверу можно, указав только лишь хост, хост и имя пользователя, или хост, имя пользователя и пароль:

```
if (empty($username))
    $mysqli = new mysqli($host);
elseif (empty($password))
    $mysqli = new mysqli($host, $username);
else
    $mysqli = new mysqli($host, $username, $password);
```

Если быть до конца точным, наиболее полный вызов конструктора может содержать еще и имя базы данных, номер прослушиваемого сервером порта и сокет:

```
new mysqli($host, $username, $password, $dbname, $port, $socket);
```

Использовать такую сигнатуру конструктора необходимо в том случае, если значения по умолчанию не устраивают, например, если задан нестандартный порт.

Модуль mysqli поддерживает два стиля записи — процедурно-ориентированный и объектно-ориентированный. Мы выбрали объектно-ориентированный стиль записи:

```
$mysqli = new mysqli($host, $username, $password);
```

Этот оператор эквивалентен вызову в процедурно-ориентированном стиле функции `mysqli_connect()`:

```
$mysqli = mysqli_connect($host, $username, $password);
```

Иногда процедурно-ориентированный стиль может быть более наглядным, как в приведенном примере. Из того, что вызван конструктор, не всякий сделает вывод, что происходит соединение (connect) с сервером баз данных MySQL.

Если соединение с сервером не удалось, мы выводим в результирующий XML-документ элементы `error`, `nocreate` и `noexecute`, закрываем XML-документ `response` и прекращаем работу скрипта:

```
if (! $mysqli){
    echo "<error>Error: no create mysqli</error>"
      . <nocreate/><noexecute/></response>";
    die();
}
```

Далее следует крайне важный оператор SQL, который позволяет правильно установить кодировку Windows-1251 (cp1251) для работы с сервером MySQL 5:

```
$result = $mysqli->query("set character set cp1251");
```

Метод `query()` позволяет выполнить строку запроса, которая состоит из ровно одного оператора языка SQL. Вызов метода возвращает значение `TRUE` при успешном выполнении запроса или `FALSE` в случае ошибки. Для запросов, содержащих операторы SQL: `SELECT`, `SHOW`, `DESCRIBE` или `EXPLAIN`, метод `query()` вернет набор данных, полученный в результате выполнения оператора.

Если запрос выполнился с ошибкой, мы формируем сообщение об ошибке и завершаем работу скрипта:

```
if (! $result) {
  echo "<error>Error: no create result</error>"
    . "<nocreate/><noexecute/></response>";
  die();
```

```
}

if ($mysqli->errno) {
  echo '<errorcode>' . $mysqli->errno . '</errorcode>';
  echo '<error>' . $mysqli->error . '</error>';
  echo "<nocreate/><noexecute/></response>";
  die();
}
```

Такую проверку мы будем проводить после каждого выполненного запроса SQL и далее в тексте раздела особо не будем останавливаться на этом моменте.

После установки кодировки Windows-1251 (cp1251) мы переходим к формированию элемента XML-документа `databaselist`:

```
$result = $mysqli->query("show databases");
$databaselist = array();
echo '<databaselist>';
while ($row = $result->fetch_array()) {
  $databaselist[] = strtoupper($row[0]);
  echo "<database>${row[0]}</database>";
}
echo '</databaselist>';
```

В данном случае выполняется оператор SQL `SHOW DATABASES`, поэтому вызов метода `query()` вернет результирующий набор данных. Строки результирующего набора данных можно перебрать последовательно в цикле, вызвав метод `fetch_array()`. Каждый новый вызов этого метода помещает в переменную `$row` новую строку результирующего набора данных. После выбора последней строки набора данных, вызов метода `fetch_array()` вернет значение `NULL`.

Вызов метода `fetch_array()` без параметров возвращает строку результирующего набора данных, одновременно, как ассоциативный массив и как простой (индексный) массив. То есть элементы можно выбирать как по имени поля, так и по индексу поля. В данном случае выбор элемента производится по его целочисленному индексу:

```
$databaselist[] = strtoupper($row[0]);
echo "<database>${row[0]}</database>";
```

Далее мы проверяем, не был ли вызван запрос с параметром `create=yes`, что означает запрос на создание новой базы данных:

```
if (array_key_exists('create', $_REQUEST)) {
  $mysqli->query(
```

```
    "create database if not exists $database character set cp1251");
  if ($mysqli->errno){
    echo '<errorcode>' . $mysqli->errno . '</errorcode>';
    echo '<error>' . $mysqli->error . '</error>';
  }else {
    echo "<execute/><nocreate/></response>";
    @$mysqli->close();
    die();
  }
}
```

Единственной причиной, по которой мы выделили запрос на создание новой базы данных, является необходимость установить для вновь создаваемой базы данных кодировку по умолчанию Windows-1251 (cp1251). С таким же успехом вы можете ввести этот запрос в окно Консоли и выполнить, как и обычный оператор SQL.

Если в запросе задана строка операторов SQL для обработки сервером баз данных MySQL, выполняется достаточно объемный фрагмент программного кода, который перебирает в цикле наборы данных, сроки наборов данных и поля строк наборов данных. Все это выглядит громоздко, но не является сложным с точки зрения логики программирования.

Вначале мы определяем, что пришел запрос на выполнение операторов SQL:

```
if (array_key_exists('execute', $_REQUEST))
```

Этот запрос может содержать не один, а сразу несколько операторов SQL, разделенных точкой с запятой. Такой запрос называют множественным запросом. Для его выполнения мы используем не метод `query()`, а метод `multi_query()`, который позволяет за один вызов функции выполнить несколько операторов SQL и получить в качестве ответа несколько результирующих наборов данных:

```
$sql = $_REQUEST['execute'];
$result = $mysqli->multi_query($sql);
```

Метод `multi_query()` никогда не возвращает результирующий набор данных и может принимать значение `TRUE`, если первый оператор SQL выполнен с успехом, или `FALSE`, если первый оператор SQL был выполнен с ошибкой.

Для получения данных из запроса `multi_query()` следует вызвать метод `store_result()` или `use_result()`:

```
$mysqli = new mysqli($host, $username, $password);
$sql = $_REQUEST['execute'];
```

```
$result = $mysqli->multi_query($sql);
$resultset = $mysqli->store_result();
```

Разница методов `store_result()` и `use_result()` заключается в том, что метод `store_result()` сразу помещает весь набор данных в память компьютера, а метод `use_result()` выбирает строки запроса с сервера по одной, загружая их с сервера.

После получения набора данных методом `store_result()` или `use_result()`, с ним можно работать теми же методами, что и с запросом, полученным методом `query()`. В частности, мы воспользуемся методом `fetch_row()`, который помещает строку результирующего набора данных в индексный (не ассоциативный) массив:

```
while ($row = $resultset->fetch_row()) {
  echo '<row>';
  foreach ($row as $field){
    echo "<field>"
    . htmlspecialchars($field, ENT_NOQUOTES)
    . "</field>";
  } // endfor
  echo '</row>';
} // endwhile
```

Собственно, этот участок кода и формирует часть XML-документа, содержащую данные из таблиц базы данных (элементы `row` и `fields`). Обратите внимание на использование функции:

```
htmlspecialchars($field)
```

Поля таблицы базы данных могут содержать недопустимые для текстовых данных XML-документа символы. Функция позволяет перекодировать недопустимые символы в символьные мнемоники. Небольшим отличием от XML является то, что функция `htmlspecialchars()` перекодирует одиночную кавычку (апостроф) не в мнемонику `&apos;`, а по числовому коду символа `&#039;`. Такая перекодировка выполняется, если задан второй параметр функции `ENT_QUOTES`. В данном случае второй параметр не задан, и поэтому преобразование для одинарной кавычки (апострофа) не выполняется. Но нам оно и не нужно. Напоминаю, что в *главе 4* мы подробно разобрали этот вопрос и пришли к выводу, что в текстовых узлах (node) XML-документа обязательно следует перекодировать только символ `&` как `&` и символ `<` как `<`. Для атрибутов, кроме того, может понадобиться перекодировать двойную или одинарную кавычку (апостроф). То есть необходимую перекодировку вызов функции `htmlspecialchars()` обеспечивает.

Мы выбрали поля в порядке их следования, и пока ничего не сказали об именах полей. Имена полей выбираются при помощи вызова метода `fetch_fields()`:

```
$fields = $resultset->fetch_fields();
echo "<result>\n<names>";
foreach ($fields as $field){
  echo "<name>$field->name</name>";
}
echo '</names>';
```

При выводе в результирующий XML-документ имен полей, функция `htmlspecialchars()` не применяется, т. к. имена полей не будут содержать недопустимых для XML-документа символов.

Не все запросы возвращают наборы данных. Для запросов, которые не возвращают набор данных, метод `store_result()` или `use_result()` вернет логическое значение `FALSE`. Для таких запросов мы определим и выведем в результирующий XML-документ количество обработанных строк таблицы базы данных:

```
if ($resultset) {
...
}else {
  echo '<result><names>'
    . <name>Count of processing rows</name></names>';
  echo '<rows><row><field>#'
    . $mysqli->affected_rows . '</field></row></rows></result>';
}// if resulteset
```

Теперь, когда мы обсудили работу с одним (самым первым) набором данных при выполнении запроса `multi_query()`, у вас должен возникнуть вопрос, как можно последовательно перебрать все оставшиеся наборы данных, полученных при выполнении множественного запроса (`multi_query()`). Для этого служит метод `next_result()`:

```
$mysqli = new mysqli($host, $username, $password);
$sql = $_REQUEST['execute'];
$result = $mysqli->multi_query($sql);
do {
  $resultset = $mysqli->store_result();
...
} while ($mysqli->next_result()); // end do while
```

С первого раза такая конструкция может показаться не очень понятной. Поэтому разберем подробнее приведенный фрагмент кода. Вызов метода `multy_query()` не помещает ни одного набора данных в результирующий набор данных. Результирующий набор данных будет возвращен методом `store_result()` или `use_result()`. Для перехода к следующему набору данных следует выполнить метод `next_result()`. Этот метод, так же как и метод `multi_query()`, возвращает `TRUE`, если очередной оператор SQL был выполнен без ошибок, или `FALSE`, если очередной оператор SQL был выполнен с ошибкой. Для получения очередного результирующего набора данных после вызова метода `next_result()` необходимо вызвать все тот же метод `store_result()` или `use_result()`.

Для обработки запросов в таком стиле как раз подходит цикл `do...while`, в котором после выполнения тела цикла вызывается метод `next_result()`, и в случае успешного выполнения метода начинается следующая итерация цикла. (Некоторые программисты почему-то не любят использовать цикл `do...while`.)

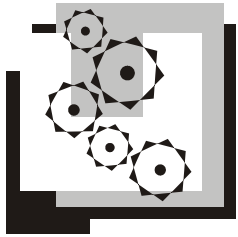
В завершающих строчках скрипта ping_database.php осуществляется проверка на допустимость создания новой базы данных и на выполнение запросов к базе данных:

```
if(in_array(strtoupper($database), $databaselist)){
    echo '<nocreate/>';
    if ($mysqli->select_db($database))
        echo '<execute/>';
}else{
    echo '<create/>';
    echo '<noexecute/>';
}
```

Все, что остается сделать, — закрыть соединение с сервером баз данных и закрыть документ `response`:

```
$mysqli->close();
echo '</response>';
```

Код компонента "Консоль базы данных MySQL 5" разработан. Теперь вы можете вернуться к *разд. 7.3* и проделать всю работу по созданию тестовой базы данных в "Консоли баз данных MySQL 5". Я уверен, что тем из вас, кто только приступает к изучению баз данных и сервера баз данных MySQL 5, работа с Консолью поможет быстро изучить операторы языка SQL и создавать самые сложные SQL-запросы.



Глава 8

Применение Ajax для регистрации пользователей Web-приложения

Работа с учетными записями пользователей реализована практически на любом Web-сайте, в любом Web-приложении. Цели, в соответствии с которыми создается система управления учетными записями пользователей, могут быть самыми разнообразными. Это может быть простой сбор статистики посещений, когда пользователь даже не знает наверняка, что создается его учетная запись. Такая учетная запись может либо создаваться для каждого сеанса пользователя, либо приложение может пытаться с определенной долей вероятности отследить повторные посещения пользователей по установленным cookie или адресу, с которого был сделан запрос. Пользователь может явно управлять своей учетной записью — регистрироваться, входить в систему, выходить из системы, настраивать пользовательский интерфейс Web-приложения с учетом своих предпочтений. Почти все Web-приложения предполагают разграничение прав доступа пользователей и реализуют один интерфейс для простых пользователей, другой для зарегистрированных пользователей, третий для администраторов системы. Наконец, наиболее тщательно проектируются системы управления пользователями в тех случаях, когда пользователи могут совершать ответственные операции в Web-приложениях, связанных с движением денежных средств или с обменом ценной информацией.

В целом, Интернет и Web-приложения могут считаться зоной достаточно открытой для атак со стороны злоумышленников. На сегодняшний день, без использования последних версий антивирусного программного обеспечения, доступ к Интернету стал практически невозможным. Тем более важно представлять себе, какая часть Web-приложения должна быть защищена, как часть может быть защищена и какие средства для этого могут использоваться. Этими вопросами мы и займемся в данной главе.

Все примеры, иллюстрирующие материал главы, вы можете найти в папке \bhvajax\profile на прилагаемом к книге компакт-диске.

8.1. Реализация базовой аутентификации и авторизации Web-сервером Apache

Базовая (basic) аутентификация и авторизация — это наиболее простая и наименее защищенная реализация аутентификации и авторизации, которая определена стандартами и поддерживается всеми популярными Web-браузерами и Web-серверами. Аутентификация и авторизация — это связанные между собой процедуры. Под *аутентификацией пользователя* понимают подтверждение пользователем, который обратился к Web-серверу, своей личности. Для аутентификации пользователя могут использоваться пароли, цифровые сертификаты, биометрические показатели пользователя. *Авторизация доступа* — это принятие сервером аутентификации пользователя и обеспечение возможности для прикладного программного обеспечения узнать имя пользователя, группу, к которой принадлежит пользователь.

Для того чтобы запросить у Web-браузера базовую аутентификацию, Web-сервер отправляет Web-браузеру заголовок:

```
WWW-Authenticate: Basic realm="realm name"
```

Параметр `realm` задает имя области, для которой действует подлинность аутентификации пользователя.

После получения запроса на аутентификацию Web-браузер выдает пользователю диалоговое окно, в котором предлагает ввести имя пользователя и пароль (рис. 8.1).

После ввода имени пользователя и пароля Web-браузер в качестве ответа на запрос аутентификации отправляет Web-серверу заголовок авторизации, например, такой:

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

Этот заголовок отправляется Web-браузером на Web-сервер в открытом (незашифрованном) виде и может быть легко прочитан злоумышленниками. В приведенном примере `Authorization` — это имя заголовка, которое отделено двоеточием от значения заголовка:

```
Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

Слово `Basic` означает, что применяется авторизация базового типа, а строка:

```
QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

это преобразованные по правилам `base64` имя пользователя и пароль, разделенные двоеточием, которые в исходном виде имели значение:

```
Aladdin:open sesame
```

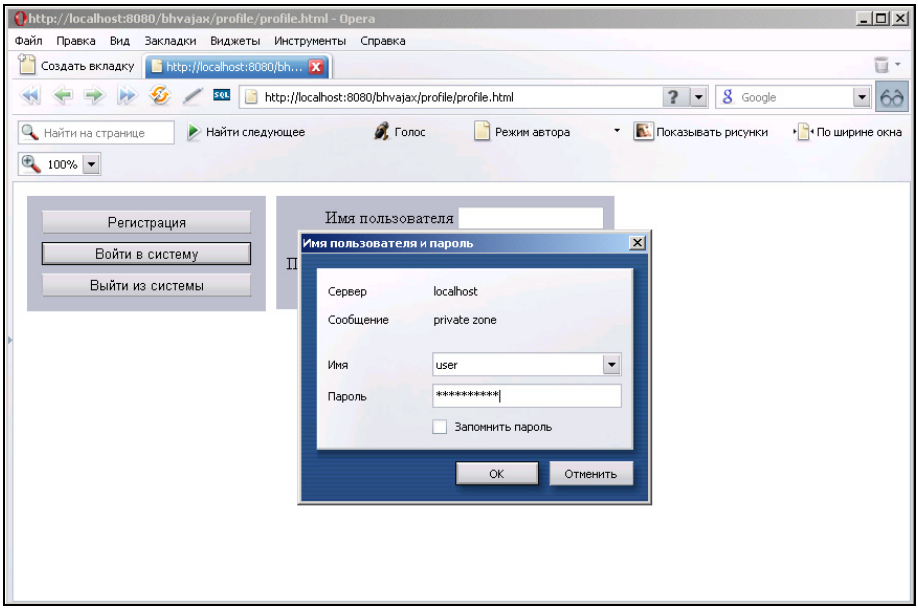


Рис. 8.1. Диалоговое окно Web-браузера для ввода имени пользователя и пароля

Несмотря на не очень читаемый вид, такая строка может быть преобразована из кодировки `base64` к обычному виду скриптом PHP 5:

```
list($name, $password) =
  explode(':', base64_decode('QWxhZGRpbjpvcGVuIHNlc2FtZQ=='));
```

> ***Замечание***
>
> Кодировка base64 включает 64 символа: буквы латинского алфавита, цифры, знаки + (плюс) и / (слэш, косая черта). В качестве дополнительного (65-го) символа используется знак = (равно), при помощи которого перекодированные строки дополняются справа из расчета, чтобы длина строки в кодировке base64 была кратна 4 для того, чтобы строки были выровнены по границе байта. Использование знака + является проблемным для Web-приложений, т. к. при помощи знака + в запросах типа `application/x-www-form-urlencoded` передается пробел. Поэтому строки в кодировке base64 приходится дополнительно перекодировать при помощи функции `encodeURIComponent()` перед отправкой на сервер.

В случае если пользователь ввел неправильное имя или пароль, Web-сервер может повторно затребовать аутентификацию или отказать в авторизации, отправив заголовок:

```
header('HTTP/1.0 401 Unauthorized');
```

Такой же заголовок будет отправлен Web-браузеру, если пользователь откажется от аутентификации и нажмет кнопку **Отменить** (Cancel) (см. рис. 8.1).

Самым удобным и надежным вариантом является управление аутентификацией и авторизацией пользователей при помощи встроенных средств Web-сервера. В стандартную поставку Web-сервера Apache входят модули, которые реализуют базовую аутентификацию и авторизацию, используя в качестве провайдеров этой услуги плоские файлы, базы данных формата DBM (это не серверы баз данных SQL), серверы LDAP. В Apache 2.2 включен модуль mod_authn_dbd, который позволяет задавать аутентификацию и авторизацию пользователей с использованием в качестве провайдера этой услуги сервера баз данных SQL, в частности сервера баз данных MySQL. К сожалению, для использования этого модуля на момент написания книги необходимо было дополнительно получить исходный код драйвера к интересующему вас серверу баз данных SQL и перекомпилировать исходный код Apache. Поэтому мы не будем рассматривать работу с модулем mod_authn_dbd в нашей книге.

Наиболее простой для использования с Web-сервером Apache является реализация базовой аутентификации и авторизации, основанная на плоских файлах, в которых хранятся имена пользователей и пароли. Для создания таких файлов с Web-сервером Apache поставляется утилита `htpasswd` (htpasswd.exe). Создавать новых пользователей и изменять пароли для существующих пользователей можно, запуская эту утилиту из командной строки. Вариантов запуска утилиты может быть несколько. Вы можете задавать имя пользователя и пароль в командной строке или вводить их в ответ на запрос с клавиатуры. В последнем случае пароли не будут отображаться на мониторе, и это создает дополнительные гарантии безопасности. Для учебного приложения обеспечивать такую секретность не имеет смысла, поэтому вы можете вводить имя и пароль в командной строке. При первом обращении к утилите для одновременного создания нового файла паролей и нового пользователя необходимо ввести такую команду:

```
htpasswd -bc имя_файла имя_пользователя пароль
```

Флаг `-b` означает, что пароль будет вводиться в командной строке, а флаг `c` означает, что будет создан новый файл с паролями. Соответственно, если файл с таким именем уже существует, все введенные до этого данные будут уничтожены. Поэтому с флагом `-bc` утилиту следует запускать ровно один раз при создании файла с паролями и первого пользователя. В дальнейшем следует запускать утилиту только с флагом `-b`:

```
htpasswd -b имя_файла имя_пользователя пароль
```

Пароль пользователя хранится, как правило, в зашифрованном виде. Реализация утилиты в некоторых операционных системах позволяет сохранять

пароли в виде открытого (незашифрованного) текста. Иногда хранение паролей в виде открытого (незашифрованного) текста выбирается по умолчанию. Способ шифрования паролей можно задавать явно в командной строке, например для сохранения пароля в виде дайджеста md5:

```
htpasswd -bm имя_файла имя_пользователя пароль
```

Очевидно, что, получив файл, даже с зашифрованными паролями, злоумышленник может методом простого перебора подобрать пароль для интересующей его учетной записи, особенно если пароль короткий. Поэтому файл с паролями должен храниться в папке (каталоге), который недоступен через Web-сервер из Интернета, т. е. не в тех папках, где вы храните общедоступные HTML-документы, скрипты, таблицы стилей и изображения. Кроме того, желательно при заведении паролей проверять, чтобы пароль имел достаточную длину исходя из требуемой степени защиты данных. Второй распространенной ошибкой при выборе пользователем пароля является выбор для паролей слов из словаря. Словарный запас человека ограничен, поэтому перебрать все слова представляется не такой уж сложной задачей. С другой стороны, если система требует ввода слишком сложного пароля — пользователь может отказаться регистрироваться в такой системе.

Файл с паролями обязательно должен храниться в папке, недоступной через Web-сервер из Интернета. Для нашего учебного приложения создадим папку users в папке Web-сервера Apache и файл с паролями C:\Apache\users\password.pwd. Теперь зададим для одного из каталогов Web-сервера необходимость базовой авторизации. Для этого в файл httpd.conf добавим следующий текст:

```
<Directory "C:/bhvajax/profile/private">
  AuthType Basic
  AuthName "realm name"
  AuthBasicProvider file
  AuthUserFile C:/Apache/users/password.pwd
  Require valid-user
</Directory>
```

Мы задали аутентификацию и авторизацию типа `Basic` (базовую) для каталога C:\bhvajax\profile\private, используя файловый провайдер. Имена и пароли пользователей хранятся в файле C:\Apache\users\password.pwd. Опция `Require valid-user` означает, что доступ к каталогу будет предоставлен всем пользователям, указавшим правильный пароль, который хранится в фале с паролями C:\Apache\users\password.pwd. Если необходимо предоставить доступ не всем пользователям, а только части — задается опция `Require` в таком виде:

```
Require user user01 user02
```

Такое подробное управление доступом пользователей может очень быстро стать неудобным. Поэтому есть возможность объединять пользователей в группы и предоставлять доступ для группы пользователей:

```
<Directory "C:/bhvajax/profile/private">
  AuthType Basic
  AuthName "realm name"
  AuthBasicProvider file
  AuthUserFile C:/Apache/users/password.pwd
  AuthGroupFile C:/Apache/users/group.grp
  Require group group01
</Directory>
```

Файл с определением состава групп пользователей создается в виде обычного текстового файла в таком формате:

```
group01: user01 user03 user12 user17
group02: user01 user07 user14 user21
```

После внесения изменений в файл httpd.conf и создания файлов с паролями пользователей, составом групп пользователей, необходимо перезапустить Web-сервер Apache. Затем при попытке получить доступ из Web-браузера к любому из ресурсов защищенного каталога Web-сервер направит в браузер заголовки:

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

В результате Web-браузер выдаст пользователю диалоговое окно для ввода имени и пароля (см. рис. 8.1). После ввода правильного имени пользователя и пароля защищенный ресурс будет повторно запрошен на Web-сервере. Причем запрос будет отправлен с теми же значениями параметров `POST` и `GET`, с теми же значениями cookie, так что повторный запрос будет "прозрачен" для разработчиков Web-приложения.

Следует особо отметить, что существуют определенные проблемы с обеспечением выхода пользователя при базовой аутентификации и авторизации. Надежный выход пользователя будет произведен только после закрытия им приложения Web-браузера. Это связано с тем, что введенные пользователем имя и пароль запоминаются Web-браузером на все время сеанса для того, чтобы при повторном обращении к защищенной области не было необходимости вновь вводить их с клавиатуры.

У разработчика Ajax-приложения может возникнуть вопрос: как будет вести себя Web-браузер при попытке отправить запрос к защищенному при помощи базовой аутентификации и авторизации ресурсу при помощи объекта

`XMLHttpRequest`? Давайте проверим это на практике. В файле test_login.html приведен HTML-документ, который при помощи Ajax-запроса обращается к защищенному Web-ресурсу private\test.html.

Как и в случае обращения к защищенному Web-ресурсу из адресной строки, Web-браузер выдает диалоговое окно для ввода имени пользователя и пароля (см. рис. 8.1).

Мы уже говорили что аутентификация, т. е. подтверждение пользователем своей личности — это одна сторона процесса. Другая сторона — авторизация доступа — заключается в предоставлении разработчику прикладного программного обеспечения сведений о текущем пользователе, который работает с Web-приложением.

После ввода имени пользователя и пароля для скриптов PHP 5 доступны две серверные переменные, которые содержат имя пользователя и пароль: `$_SERVER['PHP_AUTH_USER']` и `$_SERVER['PHP_AUTH_PW']`. Значения этих переменных можно использовать аналогично идентификатору сессии для сохранения состояния приложения между запросами в базе данных MySQL:

```
$info = $mysqli->real_escape_string(serialize($user->info));
$mysqli->query("update users set info='$info'"
  . " where name='{$mysqli->real_escape_string($this->name)}'");
```

При следующем обращении пользователя эту информацию можно извлечь из соответствующей таблицы базы данных:

```
$result = $mysqli->query("select * from users where name='$name'");
if ($result && $row = $result->fetch_array() ) {
  $info = $row['info'];
  $user->info = unserialize($info);
}
```

В приведенном фрагменте кода используются функции PHP 5 `serialize()` и `unserialize()`, которые позволяют сохранять объекты PHP 5 (с некоторыми ограничениями) в строку и восстанавливать их из строки. Для простых объектов, например ассоциативных массивов, все члены которых имеют значения скалярных типов — строкового, числового, логического — вполне достаточно использовать встроенные возможности PHP 5 для сериализации объектов. Для более сложных случаев PHP 5 предоставляет возможность программисту самостоятельно управлять сериализацией объекта.

Использование имени пользователя вместо идентификатора сессии для хранения данных сессии в таблице базы данных имеет свои достоинства, поскольку избавляет от необходимости во всех запросах явно задавать параметр сессии в случае, если пользователь отключил поддержку cookie

в Web-браузере. Но имеется один и очень существенный недостаток. Вполне вероятно, что под одним и тем же именем в систему одновременно могут войти два пользователя, если приложение не очень ответственное и пользователь по каким-то соображениям сообщил имя и пароль второму лицу, что, разумеется, делать не рекомендуется, а некоторыми Web-приложениями даже запрещается. В этом случае правильная работа системы будет нарушена.

Две переменные `$_SERVER['PHP_AUTH_USER']` и `$_SERVER['PHP_AUTH_PW']` будут созданы только в случае, если PHP 5 установлен как ISAPI-модуль Apache. В случае если PHP 5 установлен как CGI-приложение (или FastCGI), эти переменные не будут определены. Часто для создания этих переменных при работе PHP 5 как CGI-приложения Apache используют возможности модуля mod_rewrite Web-сервера Apache. Для этого в параметры каталога в файле httpd.conf необходимо добавить следующий текст (правило):

```
RewriteEngine On
RewriteRule .* - [E=MY_AUTH:%{HTTP:Authorization},L]
```

Это правило гласит, что для любого (`.*`) запроса Web-сервера будет определена серверная переменная окружения `MY_AUTH` со значением, заданным в заголовке запроса Web-браузера `HTTP:Authorization`. Далее, в скрипте PHP 5 необходимо произвести разбор значения этой переменной:

```
if(!isset($_SERVER['PHP_AUTH_USER'])
  && isset($_SERVER['MY_AUTH'])
  && preg_match('/Basic\s+(.*)$/i', $_SERVER['MY_AUTH'],
        $matches)){
  list($name, $password) = explode(':', base64_decode($matches[1]));
  $_SERVER['PHP_AUTH_USER'] = $name;
  $_SERVER['PHP_AUTH_PW'] = $password;
}
```

Базовая аутентификация и авторизация, основанные на использовании файлового провайдера, являются наименее эффективным решением с точки зрения производительности, т. к. при каждом обращении к защищенному ресурсу нового пользователя будет прочитан весь файл, содержащий пароли пользователей. Для систем, которые обслуживают тысячи и, возможно, миллионы пользователей, использование файлового провайдера не подойдет из соображений производительности. Кроме того, управление файлами при помощи утилиты командной строки `htpasswd` тоже нельзя назвать удобным решением для администрирования большого количества учетных записей.

Более эффективным является хранение паролей и сведений о составе групп пользователей в базах данных формата DBM. Строго говоря, базы данных

формата DBM — это целый класс баз данных, который предусматривает доступ к данным, похожий на доступ к членам ассоциативного массива. Провайдеров баз данных формата DBM несколько. При этом некоторые провайдеры предусматривают хранение данных в плоских файлах и по скорости работы практически не отличаются от работы с плоскими файлами. Хранение информации о пользователях в базах данных DBM имеет смысл осуществлять при серверном программировании на языке Perl, поскольку доступ к базам данных DBM имеет хорошую поддержку для языка Perl. Системная утилита Apache dbmmanage.pl, которая предназначена для управления учетными записями пользователей в базах данных DBM, также написана на языке Perl. Соответственно, имеется возможность управлять учетными записями пользователей из Web-интерфейса, используя возможности серверного программирования на Perl.

PHP 5 также имеет средства для работы с базами данных формата DBM. Работая над книгой, я пытался управлять базами данных DBM с учетными записями пользователей при помощи скриптов на PHP 5, но тут меня ожидало разочарование. Оказалось, что из обширного списка провайдеров баз данных DBM, который поддерживают PHP 5 и Apache 2.2, не нашлось и одного, который поддерживался одновременно и PHP 5, и Web-сервером Apache 2.2. Поэтому пришлось оставить эту идею.

Наконец, наиболее удачным с точки зрения производительности и удобства управления учетными записями из Web-интерфейса следует признать использование в качестве провайдеров базовой аутентификации и авторизации серверов баз данных SQL. Такая возможность в настоящее время реализована в модуле Web-сервера Apache 2.2 mod_authn_dbd. Но, к сожалению, для работы этого модуля недостаточно тех средств, которые можно получить в виде готовых инсталляционных пакетов с Web-ресурса **httpd.apache.org**. Для работы модуля mod_authn_dbd потребуется самостоятельная компиляция дополнительных библиотек, а в некоторых случаях и компиляция Web-сервера Apache из исходного кода.

8.2. Обеспечение безопасности при базовой аутентификации и авторизации

Базовая аутентификация и авторизация не содержат, как и протокол HTTP в целом, надежных средств обеспечения безопасности, на что указано даже в спецификациях прокола HTTP и базовой аутентификации и авторизации. На этом следует заострить особое внимание, поскольку вывод диалогового окна с предложением ввести имя пользователя и пароль может создать

у пользователя (да и у начинающего Web-разработчика) ложное представление о том, что все проблемы с безопасностью этим решены. К сожалению, это не так. Имя пользователя и пароль, как и весь трафик протокола HTTP, передается по сети как открытый (незашифрованный) текст и может быть прочитан (и даже изменен) злоумышленниками. Незначительным усовершенствованием базовой аутентификации и авторизации является дайджест-аутентификация. При дайджест-аутентификации по сети передается не открытый пароль, а так называемый дайджест, т. е. контрольный отпечаток, который вычисляется Web-браузером на основании введенного пользователем пароля. Пароль, передаваемый в виде дайджеста, сложнее расшифровать, но обмен данными между Web-сервером и Web-браузером, за исключением дайджеста пароля, по-прежнему остается незащищенным, поскольку весь текст пересылается в открытом (незашифрованном) виде.

Несравнимо более высокий уровень защиты предоставляет использование SSL (Secure Socket Layers) и протокола HTTPS. До недавнего времени были существенные преграды для использования SSL и HTTPS, связанные с экспортными ограничениями и авторскими правами. Если вы решите использовать протокол HTTPS в своих Web-приложениях, вам необходимо тщательно проработать вопрос о том, можете ли вы использовать этот протокол, чтобы не иметь проблем с законом. Но это не мешает нам в общих чертах рассмотреть особенности технологии SSL и протокола HTTPS с точки зрения обеспечения ими безопасности Web-приложения.

Название SSL недаром содержит слово *Layers*, т. е. слой, прослойка. На самом деле это небольшое дополнение к протоколу HTTP, которое заключается в том, что сообщения Web-браузера и Web-сервера после сжатия `gzip` и перед отправкой на Web-сервер или в Web-браузер шифруется одним из алгоритмов с симметричным ключом. То, что текст шифруется после сжатия `gzip`, принципиально важно. Дело в том, что зашифрованный текст является статистически более однородным, чем открытый текст, и сжатие такого текста не имеет смысла, потому что не будет приводить к уменьшению размера сообщения.

Симметричный ключ в протоколе HTTPS формируется для каждого сеанса связи Web-браузера и Web-сервера. Обмен симметричными ключами происходит тоже в зашифрованном виде, но с использованием несимметричных ключей, иначе говоря, открытого и закрытого ключей. Такой двухступенчатый процесс связан с тем, что открытым ключом (одним из пары несимметричных ключей) можно открыто обмениваться. Кроме того, использование несимметричных ключей обеспечивает проверку доверия к сторонам, вступающим в обмен данными. Эти ключи могут быть подписаны признанными центрами сертификации и быть ограничены в сроке действия. Так что при попытке вступить в общение по протоколу HTTPS с сомнительным Web-

сервером (т. е. с Web-сервером, имеющим сертификат самоподписанный, подписанный организацией, не имеющей доверия или просроченный) Web-браузер выдаст предупреждающее сообщение о том, что сертификат Web-сервера не пользуется доверием.

Полностью строить общение Web-браузера с Web-сервером при помощи шифрования с помощью несимметричных ключей (т. е. открытого и закрытого ключей) было бы неэффективным решением, поскольку алгоритмы шифрования и дешифрования с симметричными ключами работают существенно быстрее несимметричных. При помощи несимметричного шифрования производится обмен симметричным ключом, который имеет срок действия до окончания сеанса связи, и далее сообщения Web-браузера и Web-сервера шифруются одним из алгоритмов шифрования с симметричным ключом.

Существует предвзятое мнение о том, что работа по протоколу HTTPS существенно замедляет работу приложения. Разумеется, симметричное шифрование — это еще один дополнительный шаг при отправке и приеме сообщений Web-сервером и Web-браузером, но не такой уж решающий. Например, загрузка изображений может отнять гораздо больше ресурсов и у Web-сервера, и у Web-браузера, и у сети, чем шифрование сообщений в рамках протокола HTTPS. Большую роль играет эффективность реализации шифрования. Как показывают отчеты о производительности одного из ведущих разработчиков серверного программного обеспечения, шифрование обмена по протоколу HTTPS посредством алгоритма DES56 практически не отличается по занимаемым серверным ресурсам от передачи незашифрованного текста по протоколу HTTP. Передача данных по сети по протоколу HTTPS и вовсе не имеет никаких отличий от протокола HTTP. Поэтому работать с помощью защищенного протокола HTTPS можно рекомендовать для любого приложения, которое нуждается даже в минимальном уровне защиты.

Как правило, по защищенному протоколу имеет смысл передавать только текст (но не изображения, имеющие декоративное назначение). В этом случае повышение нагрузки на сервер будет минимальным. Для того чтобы ресурсы, например изображения, передавались по незащищенному протоколу HTTP, если основной документ загружен по защищенному протоколу HTTPS, придется в тегах IMG прописывать абсолютные адреса, включая имя протокола:

```
<IMG src="http://localhost:8080/bhvajax/text/img/image1.gif">
```

Это не очень удобно, т. к. теряются преимущества от использования относительной адресации. Другим вариантом является использование модуля mod_rewrite Web-сервера Apache:

```
RewriteCond %{SERVER_PORT} !=8080
RewriteRule ^(.*\.gif.*) http://localhost:8080/bhvajax/$1 [R,L]
```

Это правило определяет, что если запрашивается файл изображения с расширением gif через порт, отличный от 8080, на котором в нашем учебном приложении работает Web-сервер Apache по протоколу HTTP, адрес переписывается и ответ перенаправляется на порт 8080. К сожалению (а может быть и к счастью), такой вариант работает не во всех Web-браузерах и не может быть рекомендован для работы в общем случае.

Теперь коснемся отрицательных сторон работы Web-приложений по протоколу HTTPS. Разумеется, этот протокол более ресурсоемкий, чем протокол HTTP, но, как мы уже разобрали, при хорошей реализации протокола HTTPS — лишь незначительно. Дополнительным шагом по сравнению с протоколом HTTP является первоначальный обмен симметричными ключами в начале сеанса.

Сдерживающим моментом являются экспортные и авторские ограничения на использование как SSL и HTTPS в целом, так и отдельных составляющих этого протокола, таких как алгоритмы шифрования. В некоторых случаях ограничения могут накладываться на длину используемого ключа. На получение квалифицированной юридической экспертизы возможности использования защищенного протокола вам, возможно, придется затратить средств не меньше, чем на приобретение техники и программного обеспечения.

Необходимым условием успешного использования SSL и протокола HTTPS является наличие сертификата, который подписан признанным центром сертификации. Если сертификат не будет подписан признанным центром сертификации, вместо положительного эффекта от использования защищенного протокола можно получить отрицательный для бизнеса эффект. Конечно, менее защищенным Web-приложение не станет, но сообщение в диалоговом окне Web-браузера, предупреждающее о том, что сертификат Web-сервера не подписан признанным центром сертификации, скорее всего, не внушит доверия не только к Web-приложению, но и к фирме. И это является парадоксом, поскольку в данном случае пользователь будет больше доверять незащищенному протоколу.

У вас может возникнуть вопрос: насколько защищенным станет ваше приложение при использовании SSL и протокола HTTPS? Нужно учитывать, что абсолютно защищенных приложений не бывает. Существует, по крайней мере, потенциальная возможность участия в протоколе HTTPS посредника, который представился бы клиенту в качестве сервера, а серверу в качестве клиента и осуществил бы весь трафик через себя, имея возможность читать и даже изменять сообщения. Это, конечно, может быть в том случае, если сервер не знает сертификат клиента, а клиент не знает сертификат сервера. Впрочем, само управление сертификатами является довольно сложной для обычного пользователя темой, и один раз случайно признав сертификат

злоумышленника, пользователь может и далее принимать его сертификат как имеющий доверие.

Пара ключей — открытый и закрытый — на которых построены современные системы несимметричного шифрования, электронных подписей и которые используются протоколом HTTPS, являются математически зависимыми. Теоретически можно вычислить закрытый ключ простым перебором чисел от единицы и до нахождения верного решения по открытому ключу. Но практически на это уйдет время, нереальное с точки зрения современных вычислительных ресурсов. Тем не менее, если известно, что программное обеспечение для генерации пары ключей пользуется конкретными алгоритмами, в частности конкретными некриптостойкими генераторами случайных чисел, вычисление закрытого ключа по открытому ключу может быть разрешено за обозримое время. Кроме того, никто не может дать гарантий, что используемый Web-сервером ключ не был сгенерирован программой, которая по каким-либо соображениям или вследствие ошибки генерирует ключи из сравнительно небольшого подмножества ключей. Во всяком случае, это подмножество все равно будет достаточно большим, в противном случае программы начали бы генерировать дубликаты ключей, и это быстро получило бы огласку. Так что, периодически меняя ключи, вы можете быть относительно спокойны по поводу их криптостойкости, если только не работаете с очень ответственной информацией.

Ну и, наконец, необходимо четко представлять, что если информация по протоколу HTTPS передается по Интернету в зашифрованном виде, адреса, по которым проходят пакеты, являются такими же открытыми для злоумышленников, как и в протоколе HTTP. Поэтому имеется возможность определить, к каким ресурсам обращался пользователь Web-приложения. А это в некоторых случаях также должно быть предметом защиты информации.

8.3. Реализация авторизации пользователей и защиты Web-приложений средствами PHP 5 и JavaScript

Довольно часто разработчики Web-приложений реализуют аутентификацию и авторизацию пользователей самостоятельно, в частности с помощью механизма сессий PHP 5. Механизм сессий в PHP 5 действует следующим образом. Для каждой сессии генерируется уникальный идентификатор сессии SID. После этого, если пользователь не отключил поддержку cookie в Web-браузере, PHP 5 пытается установить у пользователя cookie `SESSIONID=`*`идентификатор_сессии`*. Если установить cookie не удается, во все ссылки на скрипты PHP 5, которые

используют механизм сессий, к адресной строке дописывается параметр SESSIONID=*идентификатор_сессии*. По полученному cookie или http-параметру SESSIONID, PHP 5 восстанавливает массив $_SESSION, в котором хранятся данные сессии. Процесс сохранения и восстановления данных сессии, реализованный в PHP 5 по умолчанию, может быть переопределен пользователем, ровно как и имя параметра сессии SESSIONID.

Для обеспечения авторизации в каждый (защищаемый) скрипт PHP 5 добавляется проверка на авторизацию. При первом обращении к такому ресурсу пользователю отправляется форма для ввода имени и пароля, а данные о запрошенном ресурсе и параметрах сохраняются в массиве $_SESSION. В случае ввода правильного имени пользователя и пароля в массив $_SESSION добавляется новое значение, например:

```
$_SESSION['SESION_AUTH_USER'] = $_POST['SESION_AUTH_USER'];
```

После этого повторно запрашивается страница, которую пытался получить пользователь до авторизации. На самом деле такой повторный запрос является небольшой проблемой, поскольку Web-браузер при первом доступе к защищенному Web-ресурсу мог отправить данные POST или GET. Вместо ожидаемой Web-страницы пользователь получил форму для ввода имени и пароля. После ввода имени и пароля на Web-сервер отправляется запрос, который в качестве данных POST содержит имя пользователя и пароль. Поэтому данные о первоначально запрашиваемом Web-ресурсе необходимо сохранить в массиве $_SESSION и после успешной регистрации извлечь эти данные из массива $_SESSION. Вторым вариантом является генерация для формы с логином дополнительных HTML-элементов INPUT типа HIDDEN, в которые перезаписываются отправленные пользователем данные. Можно спорить, какой из этих вариантов более подходит для вашего приложения. Я рассмотрю второй вариант, поскольку он больше соответствует тематике книги, посвященной, прежде всего, клиентским технологиям:

```
<FORM action=<?php echo $_SERVER['REQUEST_URI'] ?>>
Имя пользователя <INPUT type="text" name="SESSION_AUTH_USER"><br>
Пароль <INPUT type="password" name="SESSION_AUTH_PASSWORD"><br>
<?php
foreach ($_REQUEST as $param => $value) {
  $param=htmlspecialchars($param);
  $value=htmlspecialchars($value);
  echo "<INPUT type='hidden' name=\"$param\" value=\"$value\">";
}
?>
<INPUT type="submit">
</FORM>
```

Такой вот достаточно нехитрый программный код. В атрибут `action` HTML-элемента `FORM` заносится адрес текущего запроса. Далее, в цикле формируются HTML-элементы `INPUT` типа `HIDDEN`, в которые перезаписываются текущие параметры запроса из массива `$_REQUEST`. Могу лишь добавить, что я отладил этот программный код в нашем с вами "Редакторе кода — отладчике PHP 5" из *главы 6* всего за несколько минут и могу вам рекомендовать проделать ту же самую работу прямо сейчас.

Как вы можете убедиться, при реализации авторизации средствами PHP 5 приходится проделать всю работу, которая обеспечивается базовой аутентификацией и авторизацией вручную. Вспомним, что базовая аутентификация и авторизация после ввода правильного имени пользователя и пароля повторно запрашивают Web-ресурс с текущим состоянием параметров `GET` и `POST`, а также обеспечивают переход на запрошенный до авторизации адрес URL.

Разработчики приложений на PHP 5 для реализации авторизации пользователей применяют не столько сессии, но и базовую аутентификацию и авторизацию. При этом используются не встроенные возможности базовой аутентификации и авторизации Web-сервера Apache, а все управляется скриптами PHP 5. Несмотря на то, что реализовать такую аутентификацию и авторизацию немного проще, чем при помощи сессий, следует признать, что такой способ используется не слишком часто. В качестве объяснения этого факта я могу предположить, что разработчики PHP 5 просто чувствуют себя более уверенно, работая с возможностями PHP 5, и не прибегают к средствам, связанным с использованием протокола HTTP.

Для того чтобы запросить у Web-браузера аутентификацию, скрипт PHP 5 должен отправить Web-браузеру заголовок с требованием аутентификации (точно так это делает и Web-сервер Apache):

```
<?php
header('WWW-Authenticate: Basic realm="realm name"');
header('HTTP/1.0 401 Unauthorized');
?>
```

Как правило, заголовок аутентификации отправляется совместно с заголовком:

```
HTTP/1.0 401 Unauthorized
```

После получения этих заголовков, Web-браузер выведет пользователю диалоговое окно для ввода имени и пароля (см. рис. 8.1). Необходимо ясно представлять, что после аутентификации и авторизации имя пользователя и пароль запоминаются Web-браузером, и не существует надежного пути обеспечить выход из системы, кроме полного завершения работы приложения

Web-браузера. При использовании сессий PHP 5 для этого достаточно удалить из данных сессии соответствующее значение:

```
<?php
unset $_SESSION['PHP_AUTH_USER'];
?>
```

После ввода имени пользователя и пароля для скриптов PHP 5 становятся доступными две серверные переменные, которые содержат имя пользователя и пароль: `$_SERVER['PHP_AUTH_USER']` и `$_SERVER['PHP_AUTH_PW']`. Поэтому отправку заголовков с требованием аутентификации обычно заключают в условный блок:

```
if (isset($_SERVER['PHP_AUTH_USER']) && isset($_SERVER['PHP_AUTH_PW'])){
  if (!validUser($_SERVER['PHP_AUTH_USER'],$_SERVER['PHP_AUTH_PW']){
    header('WWW-Authenticate: Basic realm="Login "');
    header('HTTP/1.0 401 Unauthorized');
    echo 'Отказано в доступе';
    exit();
  }
}else {
  header('WWW-Authenticate: Basic realm="Login "');
  header('HTTP/1.0 401 Unauthorized');
  echo 'Введите имя пользователя и пароль';
  exit();
}
```

Этот программный код достаточно типичный для реализации базовой аутентификации и авторизации средствами PHP 5. В случае если пользователь не ввел имя или пароль, а также в случае если пользователь ввел неправильное имя или пароль (`!validUser()`), скрипт PHP 5 повторно отправляет заголовки с требованием аутентификации Web-браузеру и завершает работу скрипта:

```
header('WWW-Authenticate: Basic realm="Login "');
header('HTTP/1.0 401 Unauthorized');
echo 'Этот текст пользователь получит при отмене аутентификации';
exit();
```

Если пользователь откажется от аутентификации, нажав кнопку **Отменить** (Cancel), Web-браузер все равно получит вывод скрипта вплоть до оператора `exit()`. Об этом следует всегда помнить. Если явно не задать `exit()` после заголовка аутентификации, любой не прошедший аутентификацию пользователь может получить доступ к ресурсу, просто отменив аутентификацию.

И в случае аутентификации пользователей при помощи сессий, и в случае использования базовой аутентификации и авторизации не решается проблема защиты передаваемых по сети данных. Использование протокола HTTPS обеспечивает требуемый уровень защищенности для большинства приложений самым эффективным образом. Наличие экспортных и авторских ограничений на применение SSL и протокола HTTPS, а также обеспокоенность (не всегда обоснованная) разработчиков увеличением нагрузки на Web-сервер при использовании HTTPS привели к появлению библиотек JavaScript, которые могут шифровать обмен данными по протоколу HTTP, делая такой обмен более защищенным. (Еще одна причина, по которой разработчик может пытаться самостоятельно реализовать систему шифрования — простой интерес и тяга ко всему таинственному.) Шифроваться может как весь HTML-документ, так и некоторые HTML-элементы документа, а также данные, передаваемые в http-параметрах запроса. Надо заметить, что реализация протокола HTTPS должна в целом работать более производительно, т. к. шифрование данных при этом осуществляется программными средствами, реализованными не на прикладном уровне (например, при помощи скриптов JavaScript), а на уровне приложения Web-браузера или Web-сервера при помощи средств более низкого уровня (в некоторых случая при помощи средств операционной системы или даже аппаратных средств). Поэтому в общем случае обмен по протоколу HTTPS должен происходить значительно быстрее, чем использование реализации шифрования при помощи JavaScript или одного из серверных скриптовых языков, например PHP 5. Следует ясно представлять, что шифрование средствами JavaScript и PHP 5 приведет не только к замедлению работы Web-браузера и Web-сервера, но и к увеличению сетевого трафика. Дело в том, что при использования SSL и защищенного протокола HTTPS шифрование происходит после сжатия текста средствами `gzip`. Поэтому текст HTML-документа может быть эффективно сжат, поскольку содержит массу повторяющихся фрагментов — тегов HTML. В случае шифрования средствами PHP 5 и JavaScript на вход фильтрам, сжимающим текстовую информацию при помощи `gzip`, подается зашифрованный текст, который является статистически более однородным, чем незашифрованный HTML-документ. Поэтому сжатие `gzip` зашифрованного текста практически не даст желаемого результата, и сетевой трафик будет существенно (в разы) увеличен. Эта проблема отчасти может быть решена тем способом, что шифрованию будут подвергаться только наиболее существенные части текста, например имя пользователя, пароль, информация, введенная в поля `INPUT`.

Для того чтобы зашифровать только часть текста, можно использовать возможности вложенной буферизации PHP 5:

```
<?php
function my_encode($str) {
```

```
  if (isset($_SESSION['SECRET_KEY']))
    return some_shiper_encode($str, $_SESSION['SECRET_KEY']);
  else
    return('*');
}
ob_start();
echo '<ajax_response'>;
ob_start("my_encode");
  echo 'Зашифрованный текст внутри незашифрованных тегов';
ob_end_flush();
echo '</ajax_response'>;
ob_end_flush();
?>
```

В этом примере в качестве обработчика текста вложенного буфера назначается функция `my_encode()`. Мы совершенно не рассматриваем, какой из алгоритмов шифрования может использоваться в этой функции. В данном примере я показываю, как можно обработать часть текста буфера, не затрагивая всего документа.

На стороне клиента Web-браузера зашифрованный текст можно расшифровать в теле функции-обработчика события `onreadystatechange` объекта `XMLHttpRequest`:

```
function callback() {
  var responseText = some_schiper_decode(this.responseText, secret_key);
}
```

Остается решить, по крайней мере, две проблемы: как клиенту и серверу осуществить обмен симметричным ключом для шифрования и дешифровки текста? И как убедиться, что вы вправе использовать ту или иную систему криптографии? Начнем со второй проблемы. На системы шифрования может накладываться целый ряд ограничений, в том числе экспортные ограничения, авторские ограничения, ограничения на использование на территории определенных государств. Ограничения могут касаться как всего алгоритма шифрования, так и использования для шифрования ключей, превышающих допустимую длину. Это связано с тем, что ключи больших размеров являются, как правило, более криптостойкими, в связи с тем, что их дискредитация методом простого перебора потребует привлечения дорогостоящих вычислительных ресурсов.

Первая из упомянутых проблем, а именно секретный обмен ключами между клиентом и сервером, имеет более глобальный характер. Как мы уже упо-

минали, в протоколе HTTPS обмен симметричными ключами происходит в начале сеанса и осуществляется при помощи шифрования с помощью несимметричных ключей (открытого и закрытого ключей), и это еще один плюс в пользу использования протокола HTTPS. В Web-приложении проблема обмена ключами дополняется еще и тем, что Web-браузер имеет ряд ограничений на использование файловой системы. Поэтому ввести ключ в Web-браузер можно, получив его с Web-сервера или введя вручную с клавиатуры. Вводить ключ шифрования вручную по памяти не представляется возможным, поскольку для надежной защиты этот ключ должен быть достаточно большого размера. Для получения с сервера такой ключ должен быть передан при помощи защищенного протокола, т. е. HTTPS. Но тогда возникает вопрос: а есть ли необходимость воздвигать оригинальную систему шифрования, если в начале обмена сообщениями все равно приходится прибегать к услугам защищенного протокола HTTPS?

Все же я допускаю, что вам потребуется организовать симметричное шифрование с использованием протокола HTTP. Как я уже сказал, наиболее простой способ получить надежный ключ для симметричного шифрования — это получить его на сервере при помощи протокола HTTPS. Если вы в своих приложениях используете технологию Ajax, то, скорее всего, предпочтете получить ключ асинхронным запросом к серверу. Но тут вы столкнетесь с одной проблемой. Если основной документ загружен по протоколу HTTP, а асинхронный запрос при помощи объекта `XMLHttpRequest` попытается обратиться к тому же самому домену по протоколу HTTPS, система безопасности Web-браузера, часто именуемая "песочницей", не позволит вам сделать этого. Но существует обходной путь. Асинхронный запрос можно выполнить при помощи загрузки скрипта JavaScript в динамически формируемый HTML-элемент `SCRIPT`. Этот способ позволяет загрузить скрипт с любого домена без всяких ограничений. Такую возможность следует использовать очень осторожно, поскольку можно случайно загрузить скрипт злоумышленника, который прочитает все пароли, введенные в HTML-элементы `INPUT`, и отправит их на свой сервер. В общем, безопасность Web-браузера, связанная с возможностью кроссдоменной загрузки скриптов в HTML-элемент `SCRIPT`, является довольно уязвимой. Я привожу этот пример даже не столько для того, чтобы вы использовали его в каждом своем приложении, сколько для того, чтобы показать, какие проблемы с безопасностью могут возникнуть при использовании JavaScript.

Вспоминая о том, как в *главе 2* мы реализовывали асинхронные запросы к Web-серверу при помощи объекта `XMLHttpRequest`, программный код асинхронных запросов при помощи HTML-элементов `SCRIPT` в листинге 8.1 вам покажется очень простым.

Листинг 8.1. Асинхронные запросы при помощи HTML-элементов SCRIPT

```
bhv.sendScriptRequest =
    function(url, httpParams, callback, callbackArgsArray){
  var currentScript =document.createElement("SCRIPT");
  if (httpParams)
    httpParams="?rand=" + Math.random() + "&" + httpParams;
  else
    httpParams="?rand=" + Math.random();
  currentScript.bhv_readyState = false;
  currentScript.onload =
    bhv.util.scriptCallback(currentScript, callback, callbackArgsArray);
  currentScript.onreadystatechange =
    bhv.util.scriptCallback(currentScript, callback, callbackArgsArray);
  currentScript.src = url + httpParams;
document.getElementsByTagName("script")[0].parentNode
                                          .appendChild(currentScript);
}

bhv.util.scriptCallback = function(currentScript, callback, callbackArgsArray){
  return function(){
    if (currentScript.bhv_readyState)
      return;
    if (! currentScript.readyState
        || currentScript.readyState == "loaded"
        || currentScript.readyState == "complete"){
      currentScript.bhv_readyState = true;
      callback.apply(currentScript, callbackArgsArray)
      currentScript.parentNode.removeChild(currentScript);
    }
  }
}
```

В общем-то, этот код мог бы быть еще проще, если бы все Web-браузеры поддерживали одинаковые события при окончании загрузки скрипта. Разберем особенности работы приведенного кода. HTML-элемент `SCRIPT` создается методом, предусмотренным DOM Level 1:

```
var currentScript = document.createElement("SCRIPT");
```

Если текст программы JavaScript генерируется скриптом PHP 5, ему могут быть переданы параметры:

```
httpParams = "?rand=" + Math.random() + "&" + httpParams;
```

Специальный параметр имеет в качестве значения случайное число `Math.rand()`. Это позволит избежать загрузки скриптов из локального кэша вместо обращения к Web-серверу.

Функция-обработчик запроса назначается сразу двум событиям — `onreadystatechange` и `onload`. Это сделано для поддержки кросс-браузерности.

Загрузка скрипта начинается при выполнении оператора:

```
document.getElementsByTagName("script")[0].parentNode
                                          .appendChild(currentScript);
```

В некоторых Web-браузерах загрузка скрипта начнется чуть раньше при задании атрибута `SRC`:

```
currentScript.src = url + httpParams;
```

Приведем пример использования асинхронной загрузки:

```
<script src="../bhv/util.js"></script>
<input type="button" onclick="sendScriptRequest()"
       value="Отправить запрос">
<script type="text/javascript">
function sendScriptRequest(){
  bhv.sendScriptRequest(
    "https://localhost/bhvajax/profile/scriptResponse.php",
    "var=bhv.secretKey", alertSecretKey,[]);
}
function alertSecretKey(){
  alert(bhv.secretKey)
}
</script>
```

Соответствующий загружаемый скрипт PHP 5, который генерирует программный код на языке JavaScript, может выглядеть так:

```
<?php
header('Content-Type: text/javascript');
echo "{$_GET['var']}='". md5(rand())."'";
?>
```

Разумеется, в реальных приложениях следует пользоваться более защищенными способами генерации секретных ключей.

Замечание

Существует способ более или менее защищенной передачи симметричного ключа сеанса без использования защищенного протокола. Для этого предварительно клиент и сервер должны обменяться секретным ключом, который может быть коротким и легко запоминаемым. Например, обменяться можно при помощи электронной почты или SMS.

Для шифрования такой ключ не может использоваться непосредственно, потому что короткие ключи (а именно такие ключи могут запоминаться пользователем и вводиться с клавиатуры по памяти) легко дискредитируются методом простого перебора за считанные секунды.

Но существует небольшая лазейка, в которую можно протиснуться. Для этого в процесс перебора ключей необходимо включить человека, который должен распознавать результаты перебора всех ключей.

При помощи некриптостойкого ключа можно зашифровать картинку с изображением букв и цифр криптостойкого ключа сеанса и передать пользователю в Web-браузер зашифрованную картинку. Если картинка будет иметь защиту от распознавания программными средствами, злоумышленнику придется просмотреть все варианты расшифрованных картинок для всех возможных сочетаний секретного ключа. Если условно принять, что в качестве ключа используются символы base64 и на просмотр картинки у злоумышленника уходит одна секунда, то на взлом ключа, состоящего из одного символа, потребуется одна минута, из двух символов — один час, из трех символов — 2,5 суток, из четырех символов — 5 месяцев, из пяти символов — 25 лет, из шести символов — полтора тысячелетия.

Конечно, весь вопрос в первоначальном обмене некриптостойким секретным ключом и получении криптостойких картинок, которые не могли бы быть распознаны программным путем. Но это уже отдельная тема. И над ней усиленно работают, что вы можете наблюдать при доступе к некоторым Web-сайтам, защищающимся от роботов при помощи требования ввести рисованный код доступа.

Мы с вами рассмотрели ряд способов защиты приложений при помощи аутентификации пользователей и шифрования Web-приложений средствами PHP 5 и JavaScript. Эти способы сложнее реализовать, и они не так надежны, как реализация базовой аутентификации и авторизации средствами, предоставляемыми Web-сервером Apache и защищенным протоколом HTTPS.

8.4. Особенности использования базовой аутентификации и авторизации в Ajax-приложениях

Использование базовой аутентификации и авторизации в Ajax-приложениях позволяет существенно облегчить разграничения доступа. Для этого просто необходимо настроить права доступа к каталогам Web-приложения в файлах httpd.conf или .htaccess, и Web-сервер сам будет отслеживать доступ к Web-

ресурсам этого каталога. Мы познакомились с обеспечением аутентификации и авторизации Web-сервером Apache с помощью файлового провайдера. Для Web-приложений, в которых необходимо обеспечить аутентификацию и авторизацию тысяч пользователей, реализация при помощи плоских файлов существенно скажется на производительности системы. Кроме того, регистрация пользователей при помощи утилиты командной строки `htpasswd` (htpasswd.exe) не может быть признана достаточно удобной для работы с тысячами учетных записей. Модуль Web-сервера Apache mod_authn_dbd позволяет работать с учетными записями пользователей, которые хранятся в таблицах серверов баз данных, в частности MySQL. К сожалению, для того чтобы воспользоваться этой возможностью, необходимо проделать серьезную работу и перекомпилировать исходный код Web-сервера Apache и драйверов соответствующих баз данных. Поэтому для работы с примерами книги я реализовал разграничение доступа при помощи базовой аутентификации и авторизации средствами PHP 5 и хранением учетных записей пользователей в таблице базы данных MySQL. Это позволит нам, с одной стороны, познакомиться с особенностями реализации ведения учетных записей пользователей при помощи технологии Ajax и, с другой стороны, предоставит вам возможность использовать рассматриваемые решения, когда вы будете разрабатывать свои профессиональные приложения с использованием модуля mod_authn_dbd и сервера баз данных MySQL.

Наша система аутентификации и авторизации будет иметь такие положительные моменты по сравнению с встроенной реализацией Web-сервера, как обеспечение выхода (logout) пользователя и автоматический выход по истечении периода бездействия пользователя. Конечно, будут и отрицательные моменты. Во-первых, таким способом можно защитить только скрипты PHP 5. Если вы обратили внимание, в своих примерах Ajax-приложений мы чаще, чем скрипты PHP 5, используем документы HTML и скрипты JavaScript. Эти Web-ресурсы не будут защищены нашими средствами защиты, реализованными с помощью PHP 5. Разумеется, мы можем все HTML-документы и скрипты JavaScript пропускать через обработчик PHP 5 и задать автодобавление скрипта с требованием аутентификации при помощи директивы `auto_prepend_file` в php.ini. Но это может существенно сказаться на производительности Web-приложения.

Для того чтобы запросить у Web-браузера ввод имени пользователя и пароля, необходимо при помощи серверного скрипта PHP 5 направить следующие заголовки аутентификации:

```
header('WWW-Authenticate: Basic realm="realm name"');
header('HTTP/1.0 401 Unauthorized');
```

После ввода пользователем имени и пароля Web-браузер повторно запрашивает на Web-сервере ту же самую страницу и, что очень удобно для программирования, с теми же самыми параметрами POST, GET и COOKIE. Если пользователь отказывается от ввода, ему направляется заголовок:

```
header('HTTP/1.0 401 Unauthorized');
```

(Вернее говорить, что этот заголовок отправляется вместе с заголовком аутентификации и в случае отказа устанавливается в качестве http-статуса ответа "401 Unauthorized".)

Очевидно, что отправку заголовков необходимо проводить в условном блоке. В противном случае сервер будет требовать аутентификацию вне зависимости от правильности введенного имени и пароля:

```
if (isset($_SERVER['PHP_AUTH_USER']) && isset($_SERVER['PHP_AUTH_PW'])) {
  $user = new profile_User($_SERVER['PHP_AUTH_USER'],
$_SERVER['PHP_AUTH_PW']);
  if (! $user->validUser() ) {
    header('WWW-Authenticate: Basic realm="Login "');
    header('HTTP/1.0 401 Unauthorized');
    exit();
  }
}else {
  header('WWW-Authenticate: Basic realm="Login "');
  header('HTTP/1.0 401 Unauthorized');
  exit();
}
```

Логика работы приведенного фрагмента кода следующая. Если пользователь еще не прошел авторизацию, серверные переменные $_SERVER['PHP_AUTH_USER'] и $_SERVER['PHP_AUTH_PW'] не определены. В этом случае Web-браузеру направляется запрос на аутентификацию, и работа скрипта PHP 5 на этом прекращается. Если же пользователь ввел некоторое имя пользователя и пароль, соответствующие серверные переменные будут определены. Тогда мы создаем объект пользователя:

```
new profile_User($_SERVER['PHP_AUTH_USER'], $_SERVER['PHP_AUTH_PW']);
```

Полный код этого объекта мы рассмотрим далее. В настоящее время нам важно знать, что этот объект имеет метод validUser(), который возвращает логическое значение TRUE, если введенные имя пользователя и пароль совпадают с хранящимися в базе данных. Если метод validUser() возвращает логическое значение FALSE, это приводит к повторной отправке заголовков с требованием аутентификации.

Профили пользователей мы будем хранить в таблице `users` базы данных `bhvdb`. Для создания такой таблицы необходимо в нашей Консоли MySQL, реализованной в *главе 7*, ввести следующий оператор `CREATE`:

```
CREATE TABLE users (
  name char(50) NOT NULL,
  password char(41) NOT NULL,
  info blob,
  PRIMARY KEY (name)
)
```

Имя пользователя будет храниться в поле `name`, которое имеет тип `char(50)`. Поскольку это поле используется в качестве первичного ключа, мы без особых колебаний выбрали тип `char`, а не `varchar`. В данном примере используется естественный первичный ключ. Хотя в реальных приложениях едва ли не чаще будет использоваться суррогатный первичный ключ:

```
CREATE TABLE users (
  id bigint NOT NULL auto_increment,
  name char(50) NOT NULL,
  password char(41) NOT NULL,
  info blob,
  PRIMARY KEY (id)
)
```

Вы можете самостоятельно взвесить оба варианта и выбрать тот, который вас устраивает. С точки зрения логики работы наших примеров это не будет иметь существенного значения.

Поле `password` имеет тип `char(41)`. У вас может возникнуть вопрос: откуда я взял такой размер строкового поля и почему так смело его применил? Ответ достаточно простой. Мы не будем хранить пароли пользователей в открытом виде (мы даже ничего о них не будем знать). В поле `password` будет храниться не сам пароль, а его дайджест, полученный при помощи функции MySQL `PASSWORD()`. Вне зависимости от фактической длины пароля, его дайджест, полученный при помощи функции `PASSWORD()`, всегда будет иметь длину 41 символ. При помощи дайджеста мы никогда не узнаем значение пароля. Зато всегда сможем проверить, правильный ли пароль ввел пользователь, применив к введенному паролю ту же самую функцию `PASSWORD()`. И, наконец, следующее поле, `info`, имеет тип `blob`. Это поле мы будем использовать следующим образом. В него мы будем сохранять в сериализованном виде ассоциативный массив PHP 5, содержащий самую разнообразную информацию о пользователе. Хранить информацию в виде сериализованных объектов PHP 5 в поле `blob` таблицы базы данных часто бывает более подходящим решением,

чем создавать таблицу, содержащую сотни полей, из которых большая часть имеет значение NULL (т. е. не имеет определенного значения).

Сериализовать значение можно при помощи функции PHP 5 serialize():

```
public function saveUser() {
  $mysqli = $this->getMysqli();
  $info = $mysqli->real_escape_string(serialize($this->info));
  if ($this->exist) {
    $mysqli->query("update users set info='$info' ".
    "where name='{$mysqli->real_escape_string($this->name)}'");
  }else {
    $mysqli->query("insert into users (name, password, info)"
      ." values ('{$mysqli->real_escape_string($this->name)}',"
      ." '{$mysqli->real_escape_string($this->input_password)}',"
      ." '$info')");
  }
  $mysqli->close();
}
```

Восстановить сохраненное функцией serialize() значение можно при помощи функции deserialize():

```
private function getUser($name, $input_password = ''){
  $mysqli = $this->getMysqli();
  $result = $mysqli->query("select *,"
    ." password('$input_password') as input_password from users"
    ." where name='$name'");
  if ($result && $row = $result->fetch_array()) {
    $this->exist = true;
    $this->name = $row['name'];
    $this->password = $row['password'];
    $this->input_password = $row['input_password'];
    $info = $row['info'];
    if (gettype($info) == "string") {
      $this->info = unserialize($info);
    }
  }else {
    $result = $mysqli->query
    ("select password('{$mysqli->real_escape_string($input_password)}')"
    ." as input_password");
```

```
    if ($result && $row = $result->fetch_array()){
      $this->input_password = $row['input_password'];
    }
    $this->exist = false;
    $this->name = $name;
    $this->info['logged'] = false;
    $this->info['lastLoginTime'] = time();
    $this->info['createLoginTime'] = time();
  }
  $mysqli->close();
}
```

В приведенном примере функция `getUser()` вызывается с параметрами, содержащими имя пользователя и пароль. При этом пользователь с таким именем уже может быть зарегистрирован в системе, и введенный пароль быть правильным паролем. Имя пользователя может быть зарегистрированным в системе, но введенный пользователем пароль быть неправильным. И, наконец, имя пользователя может отсутствовать в таблице базы данных. Для того чтобы отличить последний из трех вариантов, свойству объекта `$this->exist` присваивается логическое значение `FALSE`.

Предварительно мы познакомились с особенностями реализации объекта, в котором хранится информация о пользователе. В файле User.php приведен полный программный код класса `profile_User`.

Теперь для защиты скрипта PHP 5 от доступа неавторизованного пользователя достаточно в первую строчку файла снести следующий программный код:

```
<?php
require('User.php');
profile_User::validateUser();
?>
```

Это можно сделать и непосредственно в каждом скрипте PHP 5 или для всех скриптов сразу, используя директиву `auto_prepend_file` в php.ini.

Как я уже указывал, у базовой аутентификации и авторизации есть одна проблема. После первой аутентификации в системе данные о введенном имени пользователя и пароле запоминаются Web-браузером до конца сеанса (точнее сказать, до завершения приложения Web-браузера). Удалить их можно, только завершив приложение Web-браузера (т. е. закрыв Web-браузер). В некоторых Web-браузерах реализован пункт меню **logout**. Еще одной проблемой является то, что разные модели Web-браузеров ведут себя немного

по-разному в отношении хранения данных аутентификации. И найдя решение вопроса для одной модели Web-браузера, нельзя быть уверенным, что данное решение будет работать в другой модели Web-браузера, и тем более гарантировать, что эти способы будут применимы и к новым версиям тех же самых Web-браузеров. Поэтому в коде скрипта используется целый комплекс различных средств, позволяющий отследить самые различные сценарии поведения пользователя для обеспечения выхода. Во-первых, проверяется условие на допустимые имя и пароль пользователя методом `validUser()`. К сожалению, после первого правильного входа в систему пользователя эта функция будет всегда возвращать логическое значение `TRUE`. Поэтому предусмотрен еще и метод `loggedUser()`. Он производит проверку свойства `$this->info['logged']`. При выходе из системы при помощи скрипта logout.php, который будет рассмотрен далее в этом разделе, этому свойству будет присвоено логическое значение `FALSE`.

Ну и, наконец, более детально рассмотрим условие, при котором пользователь будет допущен к работе с Web-приложением:

```
if ($user->loggedUser() && isset($user->info['lastLoginTime'])
       && time() - $user->info['lastLoginTime'] <= 60)
```

Условие `loggedUser()` содержит и условие `validUser()`. Кроме того, задано условие, что последний раз активность пользователя была не менее чем за 60 секунд до обращения к Web-приложению. Если со времени последнего обращения прошло больше 60 секунд, пользователю будет отправлен заголовок с требованием аутентификации. При каждом обращении к методу `validateUser()` у допустимых пользователей время последнего доступа будет устанавливаться равным текущему времени при помощи функции `time()`:

```
$user->info['lastLoginTime'] = time();
```

То есть пароль будет запрашиваться не каждые 60 секунд, а только после 60 секунд бездействия пользователя в Web-приложении. Можно, немного изменив код класса, применить другую тактику, и запрашивать пароль всегда по истечении определенного промежутка времени, например 5 минут.

Приведенный код имеет один недостаток, который мне не удалось устранить. При первом входе в систему пользователю будет дважды выдан запрос на ввод имени и пароля, даже если все было введено верно. Все мои попытки избавиться от такого поведения не привели к успеху. Возможно, вам это удастся лучше, чем мне.

8.5. Разработка приложения для регистрации пользователей средствами Ajax

В заключительном разделе главы мы разработаем небольшое приложение, которое позволяет новым пользователям самостоятельно регистрироваться в Web-приложении с помощью технологии Ajax. Чем технология Ajax может помочь в разработке такого приложения и чем оно будет существенно отличаться от регистрации пользователя в классическом Web-приложении? Многие из нас, хотя бы один раз, регистрировались или пытались зарегистрироваться в Web-приложениях в Интернете. Вы, наверное, можете вспомнить, как порой мучительный ввод информации заканчивался нажатием кнопки `SUBMIT` и данные отправлялись на Web-сервер, после чего к вам приходил более или менее содержательный ответ, что данные вами были введены неверно. При этом дружественно разработанные Web-приложения, по крайней мере, сохраняли поля ввода в том виде, в котором вы отправили их на сервер, и результат вашей работы был не совсем напрасным. Но не менее часто в ответ предлагался чистый лист с пустыми HTML-элементами `INPUT`, вместе с "содержательным" сообщением о том, что переменная `$php_user_sfhk` содержит недопустимое (ну никак не допустимое) значение. И вся работа проделывалась заново.

Технология Ajax позволяет совершенно по-иному организовать ввод данных пользователя. При этом правильность проверки данных будет производиться прямо в процессе их ввода, а сообщение об ошибке появится именно в том месте, где, собственно, и была допущена эта ошибка. Такое поведение приложения совсем не означает, что данные будут проверяться только на стороне клиента. Существуют, по крайней мере, три основных способа проверки данных, вводимых пользователем в Web-браузер.

Первый — данные могут проверяться на Web-сервере после нажатия пользователем кнопки `SUBMIT`. В случае правильного ввода эти данные сохраняются, а в случае, если Web-сервер обнаружил неправильные данные, пользователю отправляется сообщение об ошибке. Такой способ является наиболее надежным и должен использоваться всегда, если вы хотите иметь в своих базах данных более достоверную информацию.

Второй — данные могут проверяться на стороне клиента средствами JavaScript без запросов к Web-серверу. В традиционных Web-приложениях, как правило, использовались самые простые проверки на стороне клиента Web-браузера или не использовались вообще. Отчасти это было связано с тем, что разработчики Web-приложений лучше ориентировались в серверных технологиях или в HTML-разметке текста и достаточно поверхностно

владели JavaScript (или даже считали JavaScript языком, не заслуживающим внимания). Поэтому сложилось предубеждение, что проверка введенных данных средствами JavaScript не имеет практического значения.

Приведенные два способа проверки данных только средствами Web-сервера или только средствами Web-браузера использовались в классических Web-приложениях. Кроме этих способов в Ajax-приложениях стала доступна еще одна возможность. Итак, третий способ — проверка введенных пользователем данных может производиться асинхронными запросами к Web-серверу в процессе ввода данных пользователем. Полученное с Web-сервера сообщение об ошибке, обнаруженной в данных, разбирается `onreadystatechange`-функцией объекта `XMLHttpRequest` и сразу же отображается Web-браузером. Так что пользователь может в процессе ввода информации получать сообщения от Web-сервера, помогающие ему вводить правильную информацию, и только после ввода всех правильных данных отправлять Web-серверу запрос на сохранение всех введенных данных. Разумеется, после этого Web-сервер все равно должен проверить правильность введенных данных перед сохранением, т. е. все три способа проверки должны работать совместно, выполняя проверку доступными для своего уровня возможностями.

Давайте постепенно познакомимся с нашим приложением. Сначала рассмотрим наиболее простой программный код — JavaScript объект (класс) `Validator` (см. файл validator.js).

Функция-конструктор `Validator()` принимает три параметра. Первый параметр представляет собой HTML-элемент типа `INPUT`, данные которого (свойство `value`) будут проверяться. Второй параметр в виде строки содержит наименование события, по которому будет происходить вызов функции `callback`, передаваемой в третьем параметре. Чаще всего в качестве события будет применяться событие `onkeyup`, которое позволяет проверять данные непосредственно после ввода каждой буквы. В некоторых случаях, если данные проверяются при помощи запроса к Web-серверу, обрабатывать каждое нажатие нерационально и необходимо применять специальные меры против обработки каждого нажатия. Один из способов мы рассмотрели в *главе 7*. В общем виде обработка повторных нажатий клавиш будет рассмотрена в *главе 9*. Но сейчас мы абстрагируется от этой проблемы, чтобы не утерять основную нить повествования.

В функции-конструкторе `Validator()` для HTML-элемента `INPUT`, переданного в качестве первого параметра, создается HTML-элемент `SPAN`, в котором будет отображаться сообщение об ошибке. Этот элемент имеет абсолютное позиционирование и может быть выведен функцией `show()` как раз рядом со связанным HTML-элементом `INPUT`. Функция `hide()` делает этот элемент

SPAN невидимым, присваивая свойству стиля этого элемента display значение none.

Теперь рассмотрим объект Validator в действии:

```
Имя пользователя <input type=text id=user><br>
<script>
new Validator (document.getElementById("user"), "onkeyup", function(){
  var self = this;
  bhv.sendRequest("get", "validate_user.php",
    "user="+encodeURIComponent(self.value),true,
    validateUser,null,[self])
}
)
function validateUser(self){
  var dom =this.responseXML;
  var errors = dom.getElementsByTagName("error");
  if (typeof errors == "object" && errors.length) {
    var message = dom.getElementsByTagName("message")[0].firstChild.data;
    self.validator.show(message)
  }else
    self.validator.hide();
}
</script>
```

В данном примере для HTML-элемента INPUT, в который будет вводиться имя пользователя при регистрации, в качестве обработчика события onkeyup, назначена анонимная функция, которая отправляет запрос на проверку введенных данных к серверному скрипту validate_user.php. Ответ Web-сервера обрабатывается функцией JavaScript validateUser(), которая задается в качестве пятого параметра функции bhv.sendRequest(). Серверный скрипт validate_user.php возвращает пользователю XML-документ, который в случае обнаружения ошибки будет содержать пустой элемент <error/>, а в теле элемента <message> будет передано сообщение об ошибке.

Результат работы такой проверки на Web-сервере вы можете увидеть на рис. 8.2.

Еще проще реализовать проверку данных на стороне Web-браузера без обращения к Web-серверу. Например, простейшая проверка правильности введенного адреса электронной почты может включать проверку на отсутствие пробелов и присутствие символа @, который должен отделять непустое имя пользователя от непустого имени сервера:

```
E-mail <input type=text id=email><br>
<script>
```

```
new Validator (document.getElementById("email"), "onkeyup", function(){
  if (! this.value.match(/^[^@ ]{1,64}@[^@ ]{1,255}$/))
    this.validator.show("Ошибка в e-mail")
  else
    this.validator.hide()
}
)
</script>
```

На самом деле, строгая реализация проверки на правильность написания адреса в соответствии со спецификацией включает куда более обширное количество проверок. Я встречал реализации такой проверки на Perl, которые не помещались на экране монитора. Разумеется, правильность с точки зрения синтаксиса не означает еще существование конкретного почтового ящика. Для проверки существования используются функции работы с сетью. Я предлагаю воспользоваться сравнительно простым методом, который состоит в проверке доменного имени сервера:

```
if (! gethostbynamel($domain)){
  $isError = TRUE;
  echo ("<error/><message id='email'>Сервер: домен $domain недоступен"
        ."</message>");
}
```

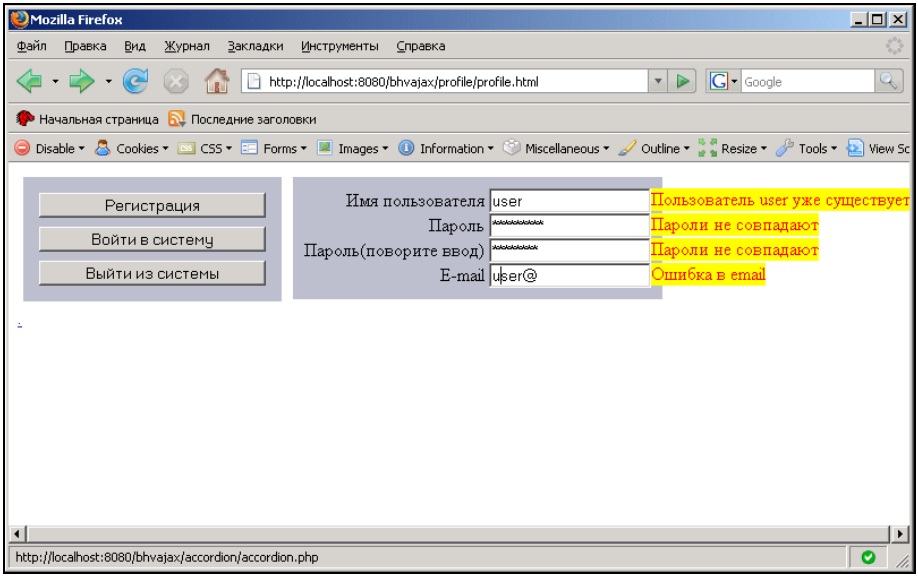


Рис. 8.2. Проверка данных регистрации пользователя средствами Ajax

Существование доменного имени еще не гарантирует существование на этом сервере почтовой службы и тем более указанного почтового ящика. Но как наиболее простой вариант проверки такой способ вполне приемлем. Во всяком случае, этим кодом не смогут воспользоваться для рассылки спама.

В реальном Web-приложении, как правило, на указанный электронный адрес высылают почтовое сообщение с просьбой перейти по ссылке по адресу регистрации с определенным http-параметром. Если почтовый ящик доступен и пользователь не является роботом, пользователь переходит по указанной ссылке и его учетная запись активируется.

После ввода всех параметров пользователь нажимает кнопку **Регистрация** (см. рис. 8.2), после чего данные отправляются на сервер Ajax-запросом:

```
<input type="button" value="Регистрация" onclick="register()">
<script>
function register() {
  var param = "user=" +
              encodeURIComponent(document.getElementById("user").value)
              + "&password1=" +
          encodeURIComponent(document.getElementById("password1").value)
              + "&password2=" +
          encodeURIComponent(document.getElementById("password2").value)
              + "&email=" +
          encodeURIComponent(document.getElementById("email").value);

  bhv.sendRequest("get", "register.php", param, false, callbackReigster)
}

function callbackReigster() {
  var dom = this.responseXML;
  var message = dom.getElementsByTagName("message");
  if (typeof message == 'object')
    for (var i = 0; i< message.length; i++)
      document.getElementById(message[i].getAttribute('id'))
        .validator.show(message[i].firstChild.data);
  var alertMessage = dom.getElementsByTagName("alert");
  if ((typeof alertMessage == 'object') && alertMessage.length)
    alert(alertMessage[0].firstChild.data)
}
</script>
```

Обработчику события `onclick` HTML-элемента `INPUT` типа `BUTTON` задана функция `register()`. Эта функция формирует строку запроса к серверу, используя функцию `encodeURIComponent()` для приведения строки к виду, допустимому для MIME-типа `application/x-www-form-urlencoded`. Полученная строка отправляется на сервер методом `bhv.sendRequest()`. В качестве обработчика ответа Web-сервера задана функция `callbackRegister()`. Эта функция ожидает, что ответ Web-сервера будет содержать XML-документ:

```
var dom = this.responseXML;
```

Этот XML-документ будет содержать сообщения об ошибках ввода, если таковые будут обнаружены сервером. Сообщения об ошибках будут содержаться в элементах `<message>`:

```
var message = dom.getElementsByTagName("message");
```

Тело элементов `<message>` будет содержать текст сообщения об ошибке, а атрибут `id` будет указывать, для какого из элементов `INPUT` отправлено сообщение об ошибке:

```
for (var i = 0; i< message.length; i++)
  document.getElementById(message[i].getAttribute('id'))
    .validator.show(message[i].firstChild.data);
```

Как и в предшествующих примерах, такую относительно сложную конструкцию приходится применять для доступа к текстовым данным, которые содержатся в теле элемента XML-документа:

```
message[i].firstChild.data
```

Для того чтобы привлечь особое внимание пользователя, в XML-документе может содержаться один элемент `<alert>`. Если такой элемент содержится в теле XML-документа, Web-браузер выведет текст сообщения при помощи функции `window.alert()`:

```
var alertMessage = dom.getElementsByTagName("alert");
if ((typeof alertMessage == 'object') && alertMessage.length)
  alert(alertMessage[0].firstChild.data)
```

В файле register.php вы можете ознакомиться с текстом скрипта PHP 5, который обрабатывает данные на сервере, создает (или не создает) учетную запись нового пользователя и формирует XML-документ с ответом.

Разумеется, в приведенном тексте все проверки носят достаточно условный характер. Например, не были проверены пароли на соответствие минимальным требованиям защиты, а также проверка синтаксиса адреса электронной почты дана самая примитивная. В данном случае я преследовал определенную цель — показать способ взаимодействия Ajax-приложения с Web-сервером при регистрации пользователя и проверке введенных данных.

В реальных приложениях текст программы каждой из таких проверок может не уместиться на экране монитора. Обратите внимание, что последовательность проверок такова, что проверки, не требующие обращения к внешним ресурсам, проводятся в самом начале, а проверка допустимого доменного имени непосредственно пред регистрацией пользователя и проводится только в том случае, если все заполнено правильно и будет производиться регистрация нового пользователя:

```
if (! $isError) {
  if (! gethostbynamel($domain)) {
    $isError = TRUE;
    echo ("<error/><message id='email'>"
      ."Сервер: домен $domain недоступен</message>");
  }
}
```

Результаты работы серверной проверки представлены на рис. 8.3. Для того чтобы разработчику было удобнее разобраться, что отображены результаты серверной проверки, все сообщения на рис. 8.3 снабжены пометкой "Сервер:". В реальных приложениях это делать совсем не обязательно, т. к. это может сбить с толку пользователя.

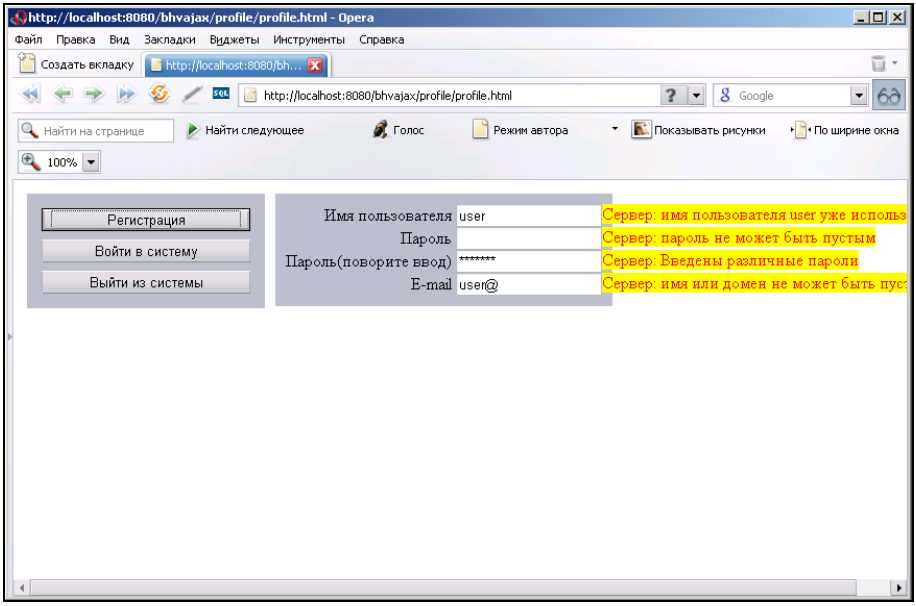


Рис. 8.3. Проверка данных регистрации пользователя на сервере

Для регистрации нового пользователя и для проверки допустимости создания нового пользователя с заданным именем (чтобы не было повторяющихся имен), мы использовали возможности класса `profile_User` (см. файл \bhvajax\profile\User.php).

```
$user = new profile_User($_REQUEST['user'], $_REQUEST['password1'] );
if ($user->exist) {
  $isError = TRUE;
  echo ("<error/><message id='user'>Сервер: имя пользователя "
    ."{$_REQUEST['user']} уже используется</message>");
}
if (! $isError) {
  $user->info['email'] = $_REQUEST['email'];
  $user->saveUser();
  echo('<register/>');
  echo('<alert>Пользователь '
    . htmlspecialchars($user->name).' зарегистрирован</alert>');
}
```

Если вы помните, свойство `info` класса `profile_User` мы договорились сохранять в сериализованном виде в поле таблицы `users`, которое имеет тип `BLOB`. Я надеюсь, что вы уже оценили преимущества этого способа хранения данных, т. к. мы можем достаточно легко добавить любое свойство к объекту `profile_User` (вернее сказать, к свойству `info` этого объекта), не задумываясь о том, как это свойство будет сохранено в таблице базы данных. Разумеется, как всегда есть и отрицательные стороны у такого способа хранения информации. По сериализованным полям нельзя сделать выборку информации средствами SQL. Этот момент как раз и должен служить критерием, по которому можно определить, какие свойства хранимого объекта нуждаются в хранении в отдельных полях базы данных, а какие могут храниться в сериализованном виде в поле типа `BLOB`.

Вторым недостатком хранения в таблице сериализованных объектов является то, что в сериализованных полях без дополнительных усилий со стороны разработчика можно хранить только скалярные свойства (числовые, строковые, даты). Если существует необходимость хранить ссылки на другие объекты, которые, в свою очередь, могут храниться в базе данных, методы сериализации должны быть явно заданы разработчиком, и пользоваться встроенными возможностями функций `serialize()` и `deserialize()` не имеет смысла.

Мы рассмотрели приемы, при помощи которых разработчики Ajax-приложений могут организовать регистрацию пользователей. Были приведены фрагменты

HTML-документа profile.html. Продолжим знакомство с текстом этого HTML-документа.

Из файла profile.html мы еще не рассмотрели две функции: `login()` и `logout()`. Первая из них должна обеспечить вход пользователя в систему, а вторая — выход пользователя из системы. Функция `login()` обращается к скрипту login.php:

```
function login() {
  bhv.sendRequest("get", "login.php", "", false,
    callbackLogin, callbackLogin)
}
```

Обратите внимание, что это как раз один из тех немногочисленных случаев, когда применяется синхронный (не асинхронный) запрос `XMLHttpRequest`, что задается параметром `async`, равным логическому значению `false`. В качестве обработчика ответа Web-сервера задана функция `callbackLogin`. В файле profile.html это пустая функция:

```
function callbackLogin() {}
```

Откровенно говоря, я не предусмотрел при разработке функции `bhv.sendRequest()` возможность отправки запроса к Web-серверу без обработки ответа. Поэтому пришлось создавать пустую функцию. На самом деле, такая функция может иметь некоторую функциональность, например, уведомлять пользователя о том, по каким причинам запрос на аутентификацию не был принят.

Функция `login()` вызывает серверный скрипт PHP 5 login.php.

Точно такой код, как в файле login.php, нужно вставить в начало каждого скрипта PHP 5, к которому должен быть обеспечен авторизованный доступ.

Функция `logout()` имеет более сложный код:

```
function logout() {
  while (true) {
    var http = bhv.util.getXMLHttpRequest();
    http.open("post", "logout.php", false);
    http.send("rand="+Math.random());
    if (http.status == 200)
      break;
  }
}
```

Дело в том, что выход пользователя из системы не предусмотрен базовой аутентификацией и авторизацией. Введенные единожды имя пользователя

и пароль будут сохранены Web-браузером и выдаваться Web-серверу в том случае, если только они будут затребованы. Для надежного выхода мы будем заносить специальный признак в учетную запись пользователя:

```
$user->info["logged"] = FALSE;
$user->saveUser();
```

С учетом кроссбраузерности, достаточно сложно придумать надежный вариант выхода пользователя. Поэтому функция `logout()` будет в "бесконечном" цикле вызывать серверный скрипт logout.php до тех пор, пока пользователь действительно не введет правильные данные и не нажмет кнопку **OK**. Я думаю, это один из немногих случаев, когда "бесконечный" цикл не будет выглядеть неуместно. Только при правильном выходе пользователя запрос будет иметь HTTP-статус 200 и произойдет выход из цикла:

```
if (http.status == 200)
  break;
```

В файле logout.php приведен код серверного скрипта, который заносит сведения о том, что пользователь вышел из Web-приложения, в базу данных.

На этом мы завершим рассмотрение работы с учетными записями пользователей при помощи технологии Ajax. Мы рассмотрели базовую (Basic) аутентификацию и авторизацию пользователей, как их можно использовать при помощи встроенных средств Web-сервера Apache и реализовать программно средствами PHP 5. Я придерживаюсь той точки зрения, что средства аутентификации и авторизации должны обеспечиваться не на уровне прикладного приложения, а на более низком уровне — Web-сервера, контейнера приложения, что повышает надежность защиты приложений и позволяет разработчикам прикладной части приложения сосредоточиться на вопросах, связанных с прикладной сферой. Несмотря на то, что это мнение является общепризнанным, по разным причинам разработчики часто реализуют аутентификацию и авторизацию на уровне Web-приложения, например, при помощи механизма сессий PHP 5. Этому причиной является недостаточная защищенность базовой аутентификации и авторизации, а также отсутствие у некоторых провайдеров удобных средств для работы с базовой авторизацией. В настоящее время, в поставку Apache 2.2 включен модуль mod_authn_dbd, который позволяет реализовать базовую аутентификацию и авторизацию с помощью серверов SQL баз данных, в частности MySQL. С использованием этого модуля аутентификацию можно было бы задать указанием следующих параметров в конфигурационном файле Apache httpd.conf:

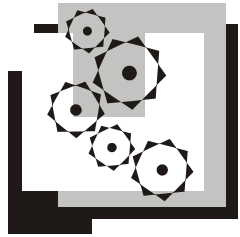
```
DBDriver mysql
DBDParams "host=localhost dbname=bhvdb user=bhv pass=d57LU293gxd"
AuthType Basic
```

```
AuthName "private zone"
AuthBasicProvider dbd
Require valid-user
AuthDBDUserPWQuery "SELECT password FROM users WHERE name = %s"
```

Такой способ аутентификации и авторизации будет действовать и быстрее, и надежнее любых средств, реализованных на PHP 5, хотя в нем будут присутствовать все те же недостатки с точки зрения незащищенности, присущие базовой аутентификации и авторизации.

* * *

Что касается рассмотренных средств защиты с применением шифрования, то наиболее надежным и быстрым, вопреки все еще распространенному предубеждению, является использование SSL и защищенного протокола HTTPS. Кроме этого, технология Ajax позволяет использовать шифрованный обмен данными между клиентом и Web-сервером при помощи симметричного шифрования. Такой обмен будет действовать существенно медленнее, чем протокол HTTPS, поскольку методы шифрования будут реализованы скриптовыми языками. Кроме того, шифрованные данные практически не будут сжиматься средствами `gzip`, поскольку содержат статистически более однородный текст, чем открытый тест, а тем более HTML-документ, изобилующий повторяющимися тегами. Используя средства шифрования, всегда следует тщательно прорабатывать вопрос о допустимости использования тех или иных средств с учетом экспортных ограничений, авторских прав, международного и отечественного законодательства. То, что вы можете свободно получить эти средства в Интернете, еще не означает, что вы можете так же свободно их использовать в своих Web-приложениях. В целом, если вы хотите, чтобы восторженные отзывы о группе "Мымын утюжок" на форуме панкрокеров случайно не прочитали ценители рэпа, будет достаточно самого простого перестановочного шифра или шифра Цезаря, описанного Светонием, на который вряд ли найдется сторона, которая потребует возместить ущерб в судебном порядке (хотя быть уверенным нельзя никогда, я *так* думаю).



Глава 9

Разработка компонента Lookup Combobox для доступа к базам данных

При вводе данных в компьютер часто приходится организовывать выбор данных из списка. Это может быть список товаров, список государств, городов или просто список ключевых слов, по которым производится поиск информации, как это реализовано в популярной службе Google Suggest. Организация поиска в списке по первым введенным буквам — это тот способ использования технологии Ajax, с которого начиналась эта технология, и которая принесла этой технологии популярность и всеобщее признание.

При реализации нашего компонента *Lookup Combobox* я постараюсь использовать лучшие стороны известных решений и внести некоторые новые моменты в реализацию, которые использую в своей практической работе и которые кажутся мне достаточно полезными для организации удобного пользовательского интерфейса.

9.1. Создание таблицы базы данных для тестирования компонента Lookup Combobox

В *главе 7* мы создали таблицу `products` в базе данных `bhvdb`. Код создания этой таблицы и тестовые данные находятся в файле \bhvajax\mysql\test_data.sql. Вот фрагмент этого файла:

```
CREATE TABLE products (
  kod bigint(20) NOT NULL auto_increment,
  name varchar(100) default NULL,
  search char(100) default NULL,
  PRIMARY KEY  (kod)
);


insert into products (search) values ("абажур");
```

```
insert into products (search) values ("абонемент");
insert into products (search) values ("абрикос");
# и так далее

update products
  set name = CONCAT_WS(" ", "Товар ", kod, "(", search, ")");
```

Для заполнения таблицы тестовыми данными используются некоторые имена существительные в алфавитном порядке. В результате получим таблицу, фрагмент которой приведен в табл. 9.1.

Таблица 9.1. *Тестовые данные для компонента Lookup Combobox (фрагмент)*

kod	name	search
1	Товар 1 (абажур)	абажур
2	Товар 2 (абонемент)	абонемент
3	Товар 3 (абрикос)	абрикос

Вы можете использовать подготовленные данные с прилагаемого к книге компакт-диска или занести в таблицу другие данные. Если у вас есть готовая база данных, можете использовать любую существующую в ней таблицу, поскольку наш компонент будет гибко настраиваться под любые данные. Разумеется, использовать для обучения, разработки и тестирования следует не рабочую базу данных, а ее копию, чтобы в процессе обучения не испортить рабочие данные.

В таблице может присутствовать произвольное количество полей. Но для компонента Lookup Combobox будут использоваться ровно три поля. Первое поле — первичный ключ, которое однозначно будет идентифицировать запись таблицы. Это поле будет использоваться в условии `WHERE` запросов SQL и возвращаться в качестве значения компонента. В приведенной таблице это поле `kod`, хотя это поле может иметь произвольное имя. Мы же создаем компонент, который может использоваться с произвольной таблицей базы данных. (На самом деле, JavaScript-код компонента может быть использован для любого сервера баз данных и с любым серверным языком программирования.) Второе поле — полное наименование, которое представляет строку таблицы в экранных формах. Это поле в нашей тестовой таблице называется `name`. Наконец, третье поле служит для инкрементного поиска строки записи в списке. Это поле имеет в нашей тестовой таблице имя `search`. Очень часто инкрементный поиск реализуют по полному наименованию. Как показывает опыт внедрения приложений, работающих с базами данных, использовать

полное наименование для этих целей зачастую бывает неудобным с точки зрения пользователя. Полное наименование, как правило, должно соответствовать официально утвержденному, порой зарегистрированному в государственных органах или взятому из конструкторской документации полному наименованию объекта. Такое наименование, как правило, содержит мелкие детали, такие как точки, кавычки, ссылки на стандарт или название предприятия-изготовителя. Такие наименования могут иметь достаточно нерегулярную структуру, и пользователю сложно запомнить все тонкости наименований по всей номенклатуре. Я исхожу из тех соображений, что поиск в списке товаров организован для продавца супермаркета, который должен обслужить одного покупателя за две-три минуты, в противном случае образуется необслуженная очередь, и должен быть, по крайней мере, не менее удобным, чем поиск, реализованный на поисковых машинах в Интернете. И все проблемы тут исходят из того, что разработчики пытаются реализовать поиск так, как им удобнее его запрограммировать.

С другой стороны, отступать от официально утвержденного полного наименования никто не имеет права, поскольку именно официально утвержденное наименование может использоваться в экранных формах и в печатных документах. Для того чтобы сделать работу пользователя со списком номенклатуры более удобной, часто добавляют специальное поле для поиска. Такое поле может строиться по-разному. Например, с номенклатурной позицией может быть связано несколько ключевых слов. При этом как с каждой номенклатурной позицией может быть связано несколько ключевых слов, так и каждое ключевое слово может быть связано с несколькими номенклатурными позициями. При вводе начальных букв ключевого слова в списке могут отражаться либо все строки, либо часть строк таблицы, связанных с этим ключевым словом. Я не буду использовать поиск по ключевым словам, а реализую несколько иной подход. С каждой номенклатурной позицией будет, кроме полного *Наименования*, связано и *Краткое наименование для поиска*. Это *Краткое наименование для поиска* будет похоже на полное *Наименование*, но будет иметь более простой вид — без мелких подробностей, точек, кавычек, дефисов, пробелов.

Когда я обеспечивал аналогичную функциональность при помощи стандартных компонентов настольных (desktop) приложений, работающих с базами данных, то сталкивался с тем, что далеко не все системы быстрой разработки приложений обеспечивают возможность работать в выпадающих списках помимо *Кода* и полного *Наименования* еще и с *Кратким наименованием для поиска*. Но если такая возможность была доступна, можно было разработать более удобный пользовательский интерфейс.

На рис. 9.1 вы можете познакомиться с внешним видом компонента Lookup Combobox, который реализует такую возможность и который мы будем разрабатывать в этой главе. Вот как работает компонент Lookup Combobox. В поле, реализованное HTML-элементом `INPUT`, мы вводим первые буквы слова, например, `агат`. При этом запрос к Web-серверу ищет подходящие слова по первым введенным буквам и выводит в раскрывающемся списке ближайшие значения в алфавитном порядке. Но в списке отображается не *Краткое наименование для поиска*, а полное *Наименование* (поле `name`).

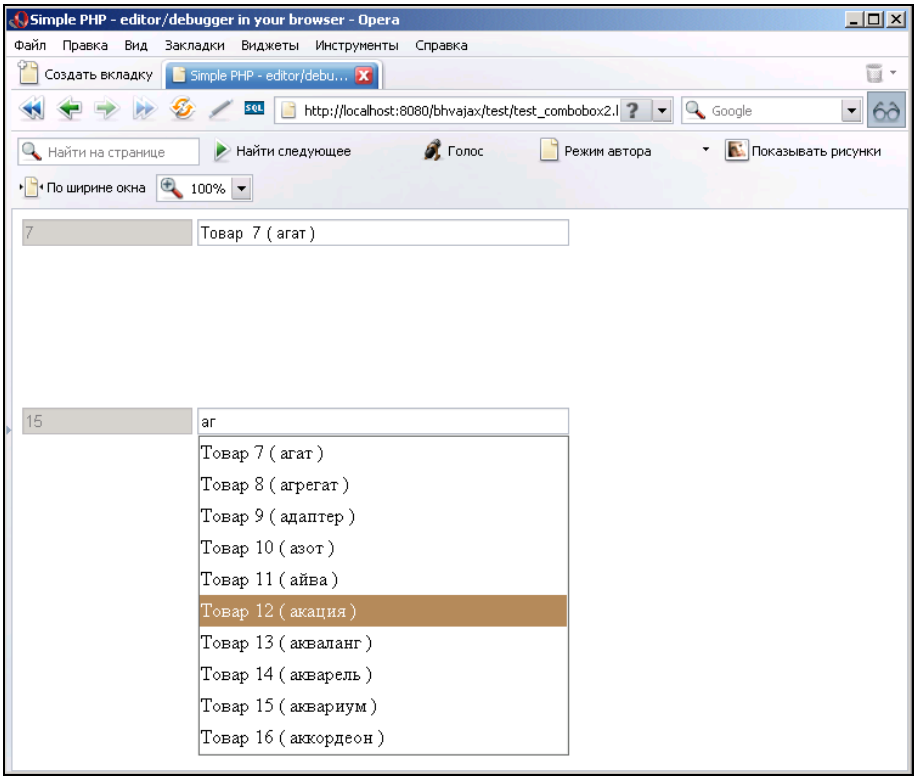


Рис. 9.1. Компонент Lookup Combobox в окне Web-браузера

Если вы никогда не занимались вводом данных из списков, содержащих сотни тысяч наименований, то можете сразу и не оценить все выгоды такого подхода. Можете поверить, что такой подход может очень вам пригодиться. Кстати, никто не заставляет использовать компонент именно в таком виде.

На самом деле, *Код*, полное *Наименование* и *Краткое наименование для поиска* не обязательно должны быть представлены разными полями. Вы можете использовать только *Код* и *Наименование* или даже только *Наименование* (без *Кода*). В этом случае *Наименование* должно быть первичным ключом и не иметь повторяющихся значений. (В принципе, это и так неявно подразумевается, но иногда могут быть исключения.)

Теперь переходим к самому важному моменту — к обеспечению защиты данных. Наш компонент Lookup Combobox может извлекать данные из таблиц и полей, имена которых задаются из программного кода JavaScript, что потенциально несет опасность. Злоумышленник может попытаться использовать эту особенность реализации компонента и запросить данные, которые мы совсем не предполагали открывать для широкого круга пользователей. Как же обеспечить защиту, чтобы наш компонент не создал брешь в безопасности приложения? Самый надежный способ — воспользоваться системой безопасности сервера MySQL. Для компонента Lookup Combobox имеет смысл создать специального пользователя:

```
GRANT SELECT (kod, name, search) on bhvdb.products
  TO 'lookup_combobox'@'localhost' IDENTIFIED BY 'cg5t6Idfk39'
```

В скрипте, который запрашивает из таблицы данные для компонента Lookup Combobox, следует явно задать пользователя:

```
<?php
$host = 'localhost';
$database = 'bhvdb';
$username = 'lookup_combobox';
$password = ' cg5t6Idfk39';
$mysqli = new mysqli($host, $username, $password);
?>
```

Теперь, когда мы будем создавать оператором `new` новый экземпляр компонента для таблицы `products`:

```
var combo1 = new bhv.Combobox("combobox1", "comb1", 7, 10,
    "products", "kod", "name", "search")
```

запросы к базе данных будут успешными. А если злоумышленик попытается запросить данные из таблицы базы данных, для которой пользователю `lookup_combobox@localhost` не предоставлены права доступа на выборку данных, запрос закончится с ошибкой времени выполнения и доступ к данным злоумышленник не получит.

9.2. Вспомогательные функции JavaScript для разработки компонента Lookup Combobox

Мы уже останавливались в *главе 7* на вопросе о том, что требуется обеспечить средства, которые предотвращают слишком частые запросы к Web-серверу при нажатии пользователем каждой клавиши. Для этого функция-обработчик нажатия клавиш не вызывают запросы к Web-серверу сразу после нажатия, а вызов запроса к Web-серверу откладывается на время заданного тайм-аута при помощи функции `window.setTimeout()`. Если пользователь успеет до истечения тайм-аута нажать очередную клавишу, предыдущий запрос отменяется, поскольку введенные данные изменились, и предыдущий запрос потерял актуальность. Работа с функцией `window.setTimeout()` не совсем удобна для пользователя, поскольку она вызывается в глобальном контексте JavaScript и имеет доступ только к глобальным переменным. В некоторых версиях Web-браузеров в функцию `window.setTimeout()` можно передать третий параметр, который определяет контекст объекта, в котором будет вызвана функция (подобно тому, как это делает функция `apply()`). В ближайшее время широко использовать эту возможность нельзя, потому что не все популярные Web-браузероы поддерживают ее. Сложности с использованием функции `window.setTimeout()` аналогичны сложностям, связанным с применением `onredystatechange`-функции объекта `XMLHttpRequest`, которые мы рассмотрели в *главе 2*.

Чтобы решить проблему, мы создадим вспомогательную функцию, которая будет обеспечивать постановку вызова прикладной функции-обработчика в очередь, передачу этой функции параметров, возможность отмены выполнения прикладной функции-обработчика и разрыв неизбежно возникающих при работе с замыканиями циклических ссылок для облегчения утилизации памяти сборщиком мусора JavaScript. Код этой функции разместим в файле bhv\util.js, поскольку такая функциональность нам потребуется не только для компонента Lookup Combobox. В листинге 9.1 приведен JavaScript код функции `bhv.setCommand()` и связанных с ней функций и объектов, которые обеспечивают необходимую функциональность.

Листинг 9.1. Реализация очереди команд для компонента Lookup Combobox

```
bhv.commandQueue={};
bhv.commandId = 0;


bhv.callCommand=function(name, id){
```

```
  if (bhv.commandQueue[name] && bhv.commandQueue[name][id])
    var currentCommand = bhv.commandQueue[name][id];
  else
    return;
  delete bhv.commandQueue[name][id];
  currentCommand.command.apply(currentCommand.context, currentCom-
mand.args);
  delete currentCommand.command;
  delete currentCommand.context;
  delete currentCommand.args;
}

bhv.setCommand=function(command, context, args, timeout, name){
  var id = "id"+ (++bhv.commandId%1000);
  if (! timeout && (timeout !== 0))
    timeout = 1000;
  if (! name)
    name = "default";
  else if (bhv.commandQueue[name])
    delete bhv.commandQueue[name];
  if (! bhv.commandQueue[name])
    bhv.commandQueue[name] = {};
  bhv.commandQueue[name][id] = {};
  bhv.commandQueue[name][id]["command"] = command;
  bhv.commandQueue[name][id]["context"] = context;
  bhv.commandQueue[name][id]["args"] = args;

  setTimeout("bhv.callCommand('" + name+ "', '" + id + "')", timeout);
}

bhv.unsetCommand=function(name){
  bhv.commandQueue[name] = null;
  delete bhv.commandQueue[name];
}
```

Для начала мы определяем пустой объект и счетчик команд:

```
bhv.commandQueue = {};
bhv.commandId = 0;
```

В свойствах объекта `bhv.commandQueue` мы будем сохранять запросы, а идентификатор команд `bhv.commandId` будем приращивать на единицу для получения уникальных идентификаторов команд.

Функция `bhv.setCommand()` имеет следующую сигнатуру:

```
bhv.setCommand = function(command, context, args, timeout, name)
```

Параметр `command` содержит ссылку на вызываемую функцию, параметр `context` — объект, который будет использоваться в качестве контекста функции при вызове методом `apply()`. Параметр `args` задает параметры функции `command`, параметр `timeout` — тайм-аут, после которого вызывается функция, а параметр `name` — имя команды. Мы условимся, что каждая команда, у которой явно задано имя, будет отменять предыдущую команду с таким же именем:

```
if (! name)
  name = "default";
else if (bhv.commandQueue[name])
  delete bhv.commandQueue[name];
```

Параметры функции сохраняются в свойствах объекта `bhv.commandQueue[name]` до момента вызова:

```
if (! bhv.commandQueue[name])
  bhv.commandQueue[name] = {};
bhv.commandQueue[name][id] = {};
bhv.commandQueue[name][id]["command"] = command;
bhv.commandQueue[name][id]["context"] = context;
bhv.commandQueue[name][id]["args"] = args;
```

В теле функции `bhv.callCommand()` эти свойства используются для вызова функции с параметрами в контексте определенного объекта:

```
var currentCommand = bhv.commandQueue[name][id];
delete bhv.commandQueue[name][id];
currentCommand.command.apply(currentCommand.context, currentCom-
mand.args);
```

Обратите внимание в коде функции на оператор `delete`, который удаляет свойства объектов и переменные. Использование оператора `delete` позволяет сборщику мусора удалять из памяти объекты, которые находились в очереди и могут содержать циклические ссылки.

Команду можно отменить по имени явно из прикладного приложения методом `bhv.unsetCommand()`.

Вызов команды при помощи функции `bhv.setCommand()` имеет следующий вид:

```
bhv.setCommand(bhv.sendRequest, bhv,
    ["get", this.getServerScript(),
      this.getHttpParams(additions, command), true,
      this.handleRequest, function(){alert(this.responseText)},
      [the, selected]],
    timeout, "bhv_combobox_" + this.element.id);
}
```

В примере вызывается функция `bhv.sendRequest()` в контексте объекта `bhv` со следующими параметрами, заданными в виде массива `Array`:

```
["get", this.getServerScript(),
  this.getHttpParams(additions, command), true,
  this.handleRequest, function(){alert(this.responseText)},
  [the, selected]
]
```

Такой синтаксис может показаться немного тяжеловесным, но полученная функциональность стоит тех усилий, которые вы затратите на его изучение.

9.3. Реализация компонента Lookup Combobox при помощи HTML-элементов

К сожалению, программный код компонента Lookup Combobox более объемный, чем код компонентов, рассмотренных нами в предыдущих главах. Поэтому стиль изложения материала данной главы будет несколько отличаться от стиля изложения предыдущих глав. Не будет приведен полный листинг программного кода компонента, и не будет разобрана каждая строчка кода. С полным кодом компонента вы можете познакомиться на прилагаемом к книге компакт-диске. Код компонента расположен в папке \bhvajax\combobox, а тестовый пример использования компонента находится в файле \bhvajax\test\test_combobox.html. Разместить тестирующий код в папке, отличной от папки, в которой расположен компонент, потребовалось для того, чтобы исключить ошибки с использованием путей в компоненте, которые могли бы и не проявиться при работе в пределах одной папки.

Наш компонент Lookup Combobox создается следующим образом. В HTML-документе явно определяется HTML-элемент `SPAN` или `DIV` в том месте, где вы собираетесь расположить компонент Lookup Combobox. У этого элемента

должен быть задан атрибут `ID` и, по возможности, задано свойство стиля `width`, определяющее ширину компонента:

```
<span id="combo1" style="width:300px"></span>
```

Компонент Lookup Combobox будет встраиваться в этот элемент `SPAN`. Для Web-дизайнера такой подход позволит располагать в HTML-документе компонент Lookup Combobox, ничего не зная об особенностях его реализации, которая будет в последующем добавлена Web-программистом.

Для ввода строки поиска будет использоваться HTML-элемент `INPUT` типа `text`. Выпадающий список будет реализован HTML-элементами `DIV`, расположенными в охватывающем элементе `DIV`. На рис. 9.2 представлена схема построения компонента Lookup Combobox при помощи HTML-элементов, и тут же представлены некоторые свойства JavaScript объекта `bhv.Combobox`, которые хранят ссылки на эти HTML-элементы.

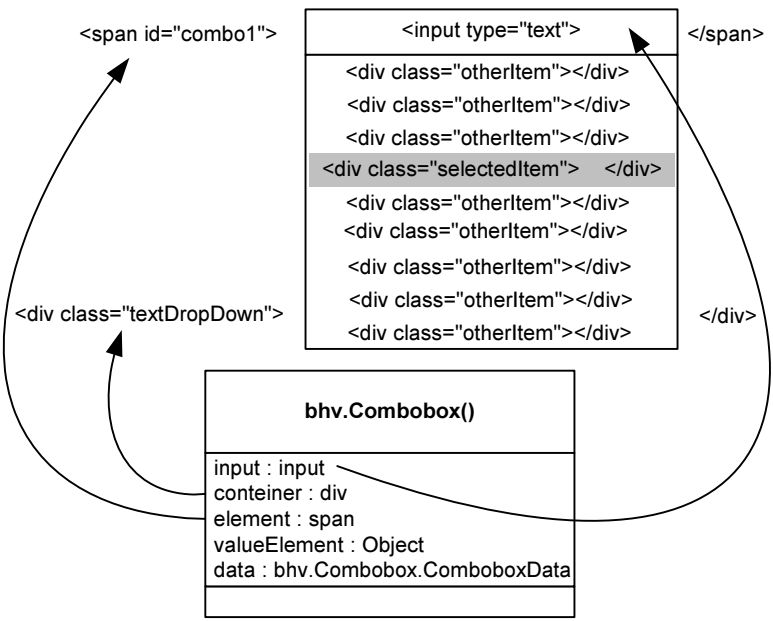


Рис. 9.2. Схема построения компонента Lookup Combobox при помощи HTML-элементов

Фрагмент кода, реализующий создание HTML-представления (Вида) компонента, имеет следующий вид:

```
bhv.Combobox = function(element, valueElement, initialValue, count,
          table, keyColumn, displayValueColumn, searchValueColumn){
```

```
  this.init(element, valueElement, initialValue, count,
           table, keyColumn, displayValueColumn, searchValueColumn);
};

bhv.Combobox.prototype = {
constructor: bhv.Combobox
,
init: function(element, valueElement, initialValue, count,
                 table, keyColumn, displayValueColumn, searchValueColumn)
{
this.data = new bhv.Combobox.ComboboxData(this.count);
if (typeof element == "string")
  this.element = document.getElementById(element);
else
  this.element = element;
this.input = document.createElement("INPUT");
this.input.type = "text";
this.input.style.width = this.element.style.width ;
this.element.appendChild(this.input);
if (typeof valueElement == "string")
  this.valueElement = document.getElementById(valueElement);
else
  this.valueElement = valueElement;
if (! this.valueElement)
  this.valueElement = {};
this.conteiner = document.createElement("DIV");
this.conteiner.className = "textDropDown"
for (var i = 0; i < this.count; i++)
  this.conteiner.appendChild(document.createElement("DIV"))
for (var i = 0; i < this.count; i++) {
  this.conteiner.childNodes[i].className = "otherItem" ;
  this.conteiner.childNodes[i].onmouseover =
    function() {the.selectOption(this);};
}
this.element.appendChild(this.conteiner)
this.table = table;
this.keyColumn = keyColumn;
this.displayValueColumn = displayValueColumn;
this.searchValueColumn = searchValueColumn;
```

```
this.requestedKey = null;
this.requestedSearchValue = null;
if (typeof initialValue != "undefined" && initialValue != null) {
  this.valueElement.value = initialValue;
  this.requestedKey = initialValue;
}
if (typeof this.valueElement.value != "undefined"){
  this.getValueFromServer("currentKey="
    + encodeURIComponent(this.valueElement.value));
}
}
```

Конструктор объекта вызывает функцию `init()`, передавая ему без изменений список параметров. Почему разработчики на JavaScript часто выносят код инициализации в отдельный метод, например `init()`, мы поговорим в *главе 11*. Код функции приведен с сокращениями, которые касаются назначения обработчиков событий. Это сделано для того, чтобы вам удобнее было проследить программный код создания HTML-элементов, рисующих компонент в окне Web-браузера. С полным кодом функции вы можете ознакомиться на прилагаемом к книге компакт-диске в скрипте \bhvajax\combobox\Combobox.js.

Как мы уже говорили, наш компонент Lookup Combobox работает с тремя полями таблицы базы данных: с *Кодом*, полным *Наименованием* и *Кратким наименованием для поиска*. В выпадающем списке всегда отображается полное *Наименование*, в HTML-элементе `INPUT` в режиме ввода вводится и отображается *Краткое наименование для поиска*, а в обычном режиме, после выбора значения из списка и при инициализации компонента — полное *Наименование*. Код, который часто не несет никакой смысловой информации из прикладной области приложения и автоматически генерируется последовательностью целых чисел или глобально-уникальными идентификаторами GUID, явно на мониторе, как правило, не отображается. Но именно это значение будет передаваться на сервер в качестве выбранного значения компонента Lookup Combobox. Для удобства работы с компонентом Lookup Combobox, с ним можно связать HTML-элемент типа `INPUT`, свойство `value` которого будет содержать выбранное значение. Этот элемент передается во втором параметре функции-конструктора. Если такой элемент не задан, будет создан объект JavaScript типа `Object`, в свойстве `value` которого будет храниться выбранное или задаваемое по умолчанию значение:

```
if (typeof valueElement == "string")
  this.valueElement = document.getElementById(valueElement);
```

```
else
  this.valueElement = valueElement;
if (! this.valueElement)
  this.valueElement = {};
if (typeof initialValue != "undefined" && initialValue != null) {
  this.valueElement.value = initialValue;
  this.requestedKey = initialValue;
}
if (typeof this.valueElement.value != "undefined"){
  this.getValueFromServer("currentKey="
    + encodeURIComponent(this.valueElement.value));
}
```

В нашем примере на рис. 9.1 в качестве такого элемента использованы HTML-элементы `INPUT` типа `text` для того, чтобы при тестировании изменения значения компонента были более наглядными:

```
<input type= text id="comb1" disabled>
<span id="combobox1" style="background: blue;width:300px"> </span>
<br><br><br><br><br><br><br><br>
<input type= text id="comb2" disabled>
<span id="combobox2" style="background: blue;width:300px"> </span>

<script>
var combo1 = new bhv.Combobox("combobox1", "comb1", 7, 10,
     "products", "kod", "name", "search")
var combo2 = new bhv.Combobox("combobox2", "comb2", 15, 10,
     "products", "kod", "name", "search")
</script>
```

В реальном приложении, для этого лучше воспользоваться элементами `INPUT` типа `hidden`.

Такие компоненты, как Lookup Combobox, всегда следует тестировать по два и более одновременно в одном HTML-документе, чтобы избежать ошибок, связанных с использованием глобальных переменных.

Код обработчиков событий сравнительно сложен, потому что необходимо с учетом кроссбраузерности обеспечить удобную для пользователя работу с компонентом. Особенно это касается работы с событиями от клавиатуры: листанием, использованием клавиш управления курсором, клавиши <Enter>, <Tab>.

Клавиша <Enter> — предмет постоянного внимания при разработке HTML-документов. По умолчанию эта клавиша приводит к выполнению действия SUBMIT, если оно задано, или к повторному запросу страницы. Предполагалось, что такое поведение клавиши <Enter> сделает работу с HTML-формами более удобной. Возможно, так оно и есть для простейших форм. Но для более сложных ("богатых") Ajax-интерфейсов такое поведение привносит неудобства. Значительно удобнее был бы переход по клавише <Enter> между полями INPUT, как это осуществляется по клавише <Tab>. В некоторых Web-браузерах для этого достаточно в обработчике события изменить код клавиши на 9 (<Tab>). Но для других Web-браузеров такой подход не подойдет, поскольку код клавиши реализован у них как свойство только для чтения. Поэтому следует действовать более культурно. В компоненте Lookup Combobox обработка нажатия клавиши <Enter> выглядит так:

```
if (event0.keyCode == bhv.key.ENTER) {
    if (this.enabled) {
        this.assignValue();
        this.hideComboBox();
        this.enabled = false;        }
    else {
        bhv.selectNextInput(this.input);
        return false;
    }
    return false;
}
```

Коды клавиш, bhv.key.ENTER и др., определены в нашем главном скрипте bhv\util.js. Там же определена и функция bhv.selectNextInput():

```
bhv.selectNextInput = function(elem){
if (! elem)
  return true;
var allInput = document.getElementsByTagName("input");
var isNext = false;
if (allInput && allInput.length > 0)
  for (var i = 0 ; i < allInput.length; i++)try{
    if (isNext && bhv.isVisible(allInput[i]) && !allInput[i].disabled){
      allInput[i].focus();
      return true;
    }
    if (! isNext && allInput[i] == elem)
```

```
      isNext = true;
  }catch (ex){}
  elem.focus();
  return true;
}
```

Немного проще устроена обработка событий, поступивших от мыши. При щелчке кнопкой мыши по HTML-элементу `INPUT` вызывается такой код:

```
onclick: function() {
  this.enabled = true;
  this.showComboBox();
  this.input.value = this.data.getCurrentSearchValue();
  this.input.select();
}
```

Сначала HTML-элемент `INPUT` становится доступным для ввода значений, далее раскрывается выпадающий список. После этого значение свойства `value` меняется с полного *Наименования* на *Краткое наименование для поиска*, и это значение выделяется при помощи функции `select()`. Текст выделяется для того, чтобы пользователь мог вводить значение для поиска поверх имеющегося значения, без нажатия клавиш <Del> и <Backspace>.

В режиме раскрытого списка, щелкнув кнопкой мыши по списку, можно выбрать значение строки и присвоить его компоненту Lookup Combobox:

```
this.conteiner.onmousedown = function(){
  the.assignValue();
  the.hideComboBox();
}
```

Код функции `assignValue()` следующий:

```
assignValue: function(selected) {
  this.input.value = this.data.getCurrentDisplayValue();
  this.valueElement.value = this.data.getCurrentKey();
}
```

Для обработки щелчка кнопкой мыши по списку мы используем не событие `onclick`, а событие `onmousedown`, потому что событие `onclick` в некоторых Web-браузерах будет относиться к HTML-элементу `INPUT`, который имеет фокус ввода, а не к HTML-элементам `DIV`, при помощи которых реализован выпадающий список.

Как видите, в теле функции `assignValue()` мы используем функции-методы свойства `data`. В свойстве `data` хранится модель данных компонента Lookup Combobox. В следующем разделе мы поговорим о модели данных компонента.

9.4. Использование паттерна MVC при разработке компонента Lookup Combobox

Паттерн MVC мы в самом упрощенном виде использовали во всех компонентах книги. Мы уже знаем, что Модель данных Ajax-приложения можно хранить на сервере, в свойствах DOM (объектной модели документа), а также в пользовательских объектах JavaScript. Последней из этих возможностей мы пока еще не пользовались на практике, поскольку все наши компоненты, разработанные до этого момента, были сравнительно простыми. Разработка компонента Lookup Combobox — это как раз тот случай, когда применение пользовательских объектов JavaScript в качестве Модели данных приведет к существенному упрощению программного кода компонента.

Модель данных компонента Lookup Combobox хранится в объекте `bhv.Combobox.ComboboxData`. В листинге 9.2 приведен полный программный код этого объекта.

Листинг 9.2. Модель данных компонента Lookup Combobox

```
bhv.Combobox.ComboboxData = function(count){
  this.count = count;
  this.currentCount = -1;
  this.currentIndex = -1;
  this.data = [];
  for (var i = 0; i < count; i++)
    this.data[i] = [];
};
bhv.Combobox.ComboboxData.prototype = {
parseXML: function(xmlDocument) {
  var rows = xmlDocument.getElementsByTagName("row");
  if (rows && rows.length && (rows.length > 0)) {
    this.currentIndex = 0;
    this.currentCount = rows.length;
    for (var i = 0; i < rows.length; i++)
      for (var j = 0; j < 3; j++)
        if (rows[i].childNodes[j].firstChild)
          this.data[i][j] = rows[i].childNodes[j].firstChild.data;
        else
          this.data[i][j] = "<i>Empty</i>";
  }else {
```

```
    this.currentIndex = -1;
    this.currentCount = 0;
  }
},
getKeyValue: function(rowIndex) {
  if (rowIndex >= this.currentCount)
    return false;
  return this.data[rowIndex][0];
},
getDisplayValue: function(rowIndex) {
  if (rowIndex >= this.currentCount)
    return false;
  return this.data[rowIndex][1];
},
getSearchValue: function(rowIndex) {
  if (rowIndex >= this.currentCount)
    return false;
  return this.data[rowIndex][2];
},
getKeyIndex: function(rowKey) {
  for (var i = 0; i < this.currentCount; i++)
    if (this.getKeyValue(i) == rowKey)
      return i
  return -1;
},
getCurrentKey: function() {
  return this.data[this.currentIndex][0];
},
getCurrentDisplayValue: function() {
  return this.data[this.currentIndex][1];
},
getCurrentSearchValue: function() {
  return this.data[this.currentIndex][2];
}
}
```

Вы можете убедиться, что код Модели данных достаточно прост. Собственно, данные хранятся в массиве `this.data`. Этот массив создается функцией `parseXML()`,

которая разбирает ответ, полученный в виде объекта DOM (т. е. в виде XML-документа) с Web-сервера:

```
parseXML: function(xmlDocument){
  var rows = xmlDocument.getElementsByTagName("row");
  if (rows && rows.length && (rows.length > 0)) {
    this.currentIndex = 0;
    this.currentCount = rows.length;
    for (var i = 0; i < rows.length; i++)
      for (var j = 0; j < 3; j++)
        if (rows[i].childNodes[j].firstChild)
          this.data[i][j] = rows[i].childNodes[j].firstChild.data;
        else
          this.data[i][j] = "<i>Empty</i>";
  }else {
    this.currentIndex = -1;
    this.currentCount = 0;
  }
}
```

Важнейшим свойством этого объекта Модели данных является свойство `this.currentIndex`, которое содержит индекс текущей строки документа. Этот индекс устанавливается кодом Контроллера в зависимости от действий пользователя. Например, индекс может устанавливаться на строку выпадающего списка, над которой находится указатель мыши:

```
selectOption: function(selectedOption) {
  for (var i = 0; i < this.count; i++) {
    if (this.conteiner.childNodes[i].className == "selectedItem")
        this.conteiner.childNodes[i].className = "otherItem"
    if (this.conteiner.childNodes[i] == selectedOption) {
        this.data.currentIndex = i;
        selectedOption.className = "selectedItem";
    }
  }
}
```

В объекте Модели данных имеются два варианта функций-методов, возвращающих значения. Часть методов возвращает значение по номеру строки:

```
getKeyValue: function(rowIndex) {
  if (rowIndex >= this.currentCount)
    return false;
```

```
  return this.data[rowIndex][0];
}
```

Другая часть функций-методов возвращает значение для текущей строки:

```
getCurrentKey: function(){
  return this.data[this.currentIndex][0];
}
```

Откровенно говоря, я даже не анализировал, какие из этих функций-методов я окончательно использовал в программе компонента, а какие из них мне не понадобились. Я просто реализовал их, а затем, в процессе разработки компонента, использовал те или другие функции-методы, в зависимости от конкретных потребностей.

Я не сторонник стрелять из пушек по воробьям, но для данного компонента использование пользовательского объекта JavaScript в качестве Модели данных компонента существенно облегчило разработку компонента.

9.5. Взаимодействие Web-браузера и Web-сервера при работе компонента Lookup Combobox

К Web-серверу при работе компонента Lookup Combobox отправляются запросы на выборку данных. Количество строк обрабатываемых данных ограничивается параметром `count` функции-конструктора объекта `bhv.Combobox`.

Существует несколько стратегий выборки данных на сервере. Мы рассмотрим наиболее простую из них. Количество строк таблицы, запрошенных с Web-сервера, будет точно соответствовать количеству строк выпадающего списка, отображаемых на экране и задаваемых параметром `count` функции-конструктора.

Кроме такого способа существуют еще и другие варианты. Могут запрашиваться данные с некоторым запасом, для того чтобы наиболее вероятные действия пользователя, например, листания на несколько страниц вперед или назад, обрабатывались без запроса новой порции данных с Web-сервера. Кроме того, один раз запрошенные данные могут кэшироваться в памяти интерпретатора JavaScript. Мы не будем обеспечивать кэширования. Сервер всегда будет возвращать одинаковое количество записей, если только они присутствуют в таблице базы данных. Запросы могут быть следующие:

- запрос конкретного значения по ключу при инициализации компонента значением, заданным в конструкторе (возвращается заданное значение и следующие за ним строки);

- запрос данных по совпадению первых введенных букв (возвращается заданное значение и следующие за ним строки);
- запрос следующей в порядке сортировки порции данных в процессе листания клавишей <PgDn>;
- запрос предыдущей в порядке сортировки порции строк в процессе листания клавишей <PgUp>.

Последнюю из возможностей реализовать немного сложнее, чем первые три, поскольку сервер баз данных не имеет встроенного оператора для запроса данных в обратном направлении, и, кроме того, можно выйти на начало таблицы и процесс листания необходимо будет "упереть" в первую строку таблицы.

В листинге 9.3 приведен программный код JavaScript, который отправляет запросы к серверному скрипту combobox\combobox_query.php, формирующему XML-документ с данными.

Листинг 9.3. Запрос данных компонента LookupCombobox с Web-сервера

```
SERVER_SCRIPT: bhv.getApplicationFolder()+"combobox/combobox_query.php",
getServerScript: function() {
  return this.SERVER_SCRIPT;
},
getHttpParams: function(additions, command){
var params = "";
  params = "table=" + this.table
    + "&keyColumn=" + this.keyColumn
    + "&displayValueColumn=" + this.displayValueColumn
    + "&searchValueColumn=" + this.searchValueColumn
    + "&table=" + this.table
    + "&count=" + this.count

  if (additions)
    params += "&"+additions;

  if (command)
    params += "&command=" + command;
    return params;
},
getValueFromServer: function(additions, command, selected, timeout){
```

```
  bhv.unsetCommand("bhv_combobox_" + this.element.id);
  var the = this;
  var timeout = 0;
  if ((typeof  additions == "string")
      && additions.indexOf("currentSearchValue") >= 0)
    timeout = 1000;
  bhv.setCommand(bhv.sendRequest, bhv,
      ["get", this.getServerScript(),
      this.getHttpParams(additions, command), true,
      this.handleRequest,function(){alert(this.responseText)},
      [the, selected]],
    timeout, "bhv_combobox_" + this.element.id);
},
getValueFromServerSync: function(additions, command, selected){
  bhv.unsetCommand("bhv_combobox_" + this.element.id);
  var the = this;
  bhv.sendRequest("get", this.getServerScript(),
      this.getHttpParams(additions, command), false,
      this.handleRequest, null, [the, selected]);
},
handleRequest: function(combobox, selected){
  combobox.data.parseXML(this.responseXML);
  if (combobox.enabled) {
    combobox.element.value = combobox.data.getCurrentKey();
    combobox.showComboBox(selected);
    var matchedChar = bhv.compareString(combobox.input.value,
      combobox.data.getCurrentSearchValue());
    if (combobox.input.createTextRange){
      textRange = combobox.input.createTextRange();
      textRange.moveStart('character', matchedChar);
      textRange.select();
    }else
      combobox.input.setSelectionRange(matchedChar,
           combobox.input.value.length);
  }else
    combobox.input.value = combobox.data.getCurrentDisplayValue();
}
```

Путь к серверному скрипту определяется при помощи функции `bhv.getApplicationFolder()`:

```
bhv.getApplicationFolder()+"combobox/combobox_query.php"
```

На протяжении всей книги реализованы две основные концепции задания адресов URL к Web-ресурсам приложений:

- путь в URL может быть задан в форме относительно каталога текущего HTML-документа или скрипта PHP5;
- путь может быть задан относительно каталога приложения. В нашем примере это каталог \bhvajax. Для определения каталога приложения в библиотеке bhv\util.js определена функция `bhv.getApplicationFolder()`.

И единственный способ задания адресов URL, который мы принципиально не используем в книге, — это задание абсолютных URL (включая имя протокола, хост и порт) и задание URL в виде абсолютных путей, заданных относительно корневого каталога Web-сервера (которые начинаются со знака `/`). Исключение из использования абсолютных путей позволяет легко переносить приложение из одного каталога в другой каталог и размещать на одном хосте несколько Web-приложений в разных каталогах без изменения программного кода.

> ***Замечание***
>
> Абсолютные URL включают имя протокола, хост, порт и путь к ресурсу относительно корневого каталога Web-сервера. Если URL задан без указания протокола, хоста и порта, такой URL называется *относительным* URL. Относительный URL может быть задан абсолютным путем от корневого каталога Web-сервера и начинается символом / (слэш, косая черта). Кроме того, относительный URL может быть задан относительным путем, заданным относительно текущего каталога документа. С точки зрения разработчика, использовать абсолютные URL и относительные URL, заданные абсолютными путями от корневого каталога Web-сервера, неудобно, потому что перенос приложения в другой каталог влечет за собой изменение текста всех документов. Поэтому разработчики предпочитают использовать относительные URL, заданные путями относительно текущего каталога документа.
>
> Каталог приложения — это возможность, которая не обеспечивается протоколом HTTP и реализуется на уровне приложения, библиотеки или сервера приложений. Использование каталогов приложений позволяет на одном хосте размещать произвольное количество Web-приложений — каждое в своем каталоге, и переносить приложения из одного каталога в другой без изменения кода приложения. Для этого адреса, заданные относительно каталога приложения, должны программно преобразоваться к абсолютным или относительным URL.

Программный код функции `bhv.getApplicationFolder()` был рассмотрен в *главе 3*. В нашем случае функция будет возвращать путь к каталогу \bhvajax\, который для примеров книги является каталогом приложения. Если вы не хотите разбираться с программной реализацией этой функции,

можете задать ее в упрощенном варианте, подставив в коде функции необходимое имя каталога приложения:

```
bhv.getApplicationFolder(){
  return "/bhvajax/";
}
```

Запросы к серверу отправляет функция `getValueFromServer()` или `getValueFromServerSync()`. Первая из них вызывается асинхронно и с небольшим запаздыванием, что реализовано при помощи функции `bhv.setCommand()`, приведенной в листинге 9.1. Это позволяет исключить лишние запросы к Web-серверу.

Кроме асинхронных запросов, иногда приходится использовать синхронные запросы без тайм-аута. Это может быть необходимо при инициализации компонента значением по умолчанию. Если такие запросы выполнять асинхронно и с тайм-аутом, пользователь может успеть воспользоваться совершенно случайной информацией.

Запрос данных на сервере может инициироваться событиями с клавиатуры:

```
if (event0.keyCode == bhv.key.PAGEDOWN){
  if (this.data.currentIndex < this.data.currentCount - 1)
    this.data.currentIndex = this.data.currentCount - 1
  else
    this.getValueFromServer("currentKey=" + this.data.getCurrentKey());
}else if (event0.keyCode == bhv.key.PAGEUP){
  if (this.data.currentIndex > 0)
    this.data.currentIndex = 0
  else
    this.getValueFromServer("currentKey=" + this.data.getCurrentKey(),
      "previous", "first");
}else if (event0.keyCode == bhv.key.DOWN){
  if (this.data.currentIndex < this.data.currentCount - 1)
    this.data.currentIndex ++;
  else
    this.getValueFromServer("currentKey=" + this.data.getCurrentKey());
}else if (event0.keyCode == bhv.key.UP){
  if (this.data.currentIndex > 0)
    this.data.currentIndex --;
  else
    this.getValueFromServer("currentKey=" + this.data.getCurrentKey(),
```

```
      "previous", this.data.getCurrentKey());
}else {
  if (! this.enabled)
    combobox.input.value = combobox.data.getCurrentSearchValue();
  else
    this.getValueFromServer("currentSearchValue="
        + encodeURIComponent(this.input.value));
}
```

Строки таблицы могут запрашиваться по первым введенным символам:

```
this.getValueFromServer("currentSearchValue="
        + encodeURIComponent(this.input.value));
```

Для обработки листания клавишей <PgDn> вызывается функция со значением ключа последней строки, и Web-сервер вернет очередную порцию строк, начиная с последней:

```
this.getValueFromServer("currentKey=" + this.data.getCurrentKey());
```

Для обработки листания клавишей <PgUp> вызывается функция со значением ключа первой строки, и Web-сервер вернет предшествующую первой строке порцию строк, заканчивая первой:

```
this.getValueFromServer("currentKey=" + this.data.getCurrentKey(),
      "previous", "first");
```

В некоторых случаях обработка нажатий клавиш не приводит к новому запросу к Web-серверу, и просто изменяется индекс текущей строки:

```
if (event0.keyCode == bhv.key.UP){
  if (this.data.currentIndex > 0)
    this.data.currentIndex --;
  else
    this.getValueFromServer("currentKey=" + this.data.getCurrentKey(),
      "previous", this.data.getCurrentKey());
}
```

В приведенном фрагменте кода обработка нажатия клавиши <↑> приводит к изменению текущего индекса модели данных:

```
this.data.currentIndex --;
```

Когда будет достигнута первая запись (с индексом ноль), будет отправлен запрос на сервер. В листинге 9.4 приведен серверный код формирования XML-документа.

Листинг 9.4. Серверный скрипт combobog_query.php

```
<?php
header('Content-type: text/xml; charset=\"windows-1251\"?');
require_once('../bhv/errorhandler.php');

$host = 'localhost';
$database = 'bhvdb';
$username = 'lookup_combobox';
$password = '123';

error_reporting(E_ALL |  E_STRICT);
echo "<?xml version=\"1.0\" encoding=\"windows-1251\"?>\n\n";
echo '<response>';

$mysqli = new mysqli($host, $username, $password);
$mysqli->query("set character set cp1251");
$mysqli->select_db($database);

$table = $_REQUEST['table'];
$keyColumn = $_REQUEST['keyColumn'];
$displayValueColumn = $_REQUEST['displayValueColumn'];
$searchValueColumn = $_REQUEST['searchValueColumn'];
$count = $_REQUEST['count'];

if (isset($_REQUEST['command']) &&  $_REQUEST['command'] == 'previous'){
  $currentKey = $_REQUEST['currentKey'];
  $result = $mysqli->query(
    "select $keyColumn as field1, $displayValueColumn as field2,"
    . " $searchValueColumn as field3 from $table"
    . " where $keyColumn = '$currentKey' limit 1");
  if ($result and $row = $result->fetch_row())
    $currentSearchValue = $row[2];
  else{
    echo '<error>Not foud key</error></response>';
    die();
  }
  $result = $mysqli->query(
    "select $keyColumn as field1, $displayValueColumn as field2,"
```

```
    . " $searchValueColumn as field3 from $table"
    . " where $searchValueColumn < '$currentSearchValue'"
    . " or $searchValueColumn = '$currentSearchValue'"
    . " and $keyColumn <= '$currentKey'"
    . " order by field3 desc, field1 desc limit $count"
    . '');
  while ($result and $row = $result->fetch_row()){
    $currentSearchValue = $row[2];
    $currentKey = $row[0];
  }
  $sqlwhere = " where $searchValueColumn > '$currentSearchValue'"
    . " or $searchValueColumn = '$currentSearchValue'"
    . " and $keyColumn >= '$currentKey'";
}elseif (isset($_REQUEST['currentKey'])){
  $currentKey = $_REQUEST['currentKey'];
  $result = $mysqli->query(
    "select $keyColumn as field1, $displayValueColumn as field2,"
    . " $searchValueColumn as field3 from $table"
    . " where $keyColumn = '$currentKey' limit 1");
  if ($result and $row = $result->fetch_row())
    $currentSearchValue = $row[2];
  else {
    echo '<error>Not foud key</error></response>';
    die();
  }
  $sqlwhere = " where $searchValueColumn > '$currentSearchValue'"
    . " or $searchValueColumn = '$currentSearchValue'"
    . " and $keyColumn >= '$currentKey'";
}elseif (isset($_REQUEST['currentSearchValue'])){
  $currentSearchValue = $_REQUEST['currentSearchValue'];
  $result = $mysqli->query(
    "select $keyColumn as field1, $displayValueColumn as field2,"
    . " $searchValueColumn as field3 from $table"
    . " where  $searchValueColumn like '$currentSearchValue%'"
    . " order by field3, field1 limit 1");
  if ($result and $row = $result->fetch_row())
    $currentKey = $row[0];
  else {
    $result = $mysqli->query(
```

```
      "select $keyColumn as field1, $displayValueColumn as field2,"
      . " $searchValueColumn as field3 from $table"
      . " where  $searchValueColumn <= '$currentSearchValue'"
      . " order by field3 desc, field1 desc limit 1");
    if ($result and $row = $result->fetch_row()){
      $currentSearchValue = $row[2];
      $currentKey = $row[0];
    }else {
      $result = $mysqli->query(
         "select $keyColumn as field1, $displayValueColumn as field2,"
        . " $searchValueColumn as field3 from $table"
        . " order by field3 asc, field1 asc limit 1");
      if ($result and $row = $result->fetch_row()){
        $currentSearchValue = $row[2];
        $currentKey = $row[0];
      }else {
        echo '<error>Not foud value</error></response>';
        die();
      }
    }
  }
  $sqlwhere = " where $searchValueColumn > '$currentSearchValue'"
    . " or $searchValueColumn = '$currentSearchValue'"
    . " and $keyColumn >= '$currentKey'";
}else {
  $result = $mysqli->query(
    "select $keyColumn as field1, $displayValueColumn as field2,"
    . " $searchValueColumn as field3 from $table"
     . " limit 1");
  if ($result and $row = $result->fetch_row()){
    $currentSearchValue = $row[2];
    $currentKey = $row[0];
  }else {
    echo '<error>Not foud value</error></response>';
    die();
  }
  $sqlwhere = " where $searchValueColumn > '$currentSearchValue'"
    . " or $searchValueColumn = '$currentSearchValue'"
    . " and $keyColumn >= '$currentKey'";
```

```
}
$result = $mysqli->query(
  "select $keyColumn as field1, $displayValueColumn as field2,"
  . " $searchValueColumn as field3 from $table"
  . " where $searchValueColumn > '$currentSearchValue'"
  . " or $searchValueColumn = '$currentSearchValue'"
  . " and $keyColumn >= '$currentKey'"
  . " order by field3, field1 limit $count"
  . '');
while ($result and $row = $result->fetch_row()){
  echo '<row>';
  foreach ($row as $field)
    echo "<field>$field</field>";
  echo '</row>';
}
echo '</response>';
?>
```

Я отдаю себе отчет в том, что разбирать чужой код генерирования SQL-команд — работа очень и очень неблагодарная. Самое Основное действие происходит в последних строчках кода, когда собственно и формируется XML-документ.

```
while ($result and $row = $result->fetch_row()){
  echo '<row>';
  foreach ($row as $field)
    echo "<field>$field</field>";
  echo '</row>';
}
```

Существует хорошая шутка о том, что лучшим комментарием к программе является программа, написанная на другом языке программирования. Существует, и это уже не шутка, мнение, что любая программа на языке программирования Python более лаконична, чем аналогичная программа на любом другом языке программирования. Я неоднократно упоминал в книге, что вы можете использовать произвольный Web-сервер, сервер баз данных и серверный язык программирования, практически ничего не меня в программах JavaScript (кроме расширения серверного скрипта). В листинге 9.5 я привожу реализацию серверной обработки запросов компонента Lookup Combobox, реализованную на языке программирования Python при помощи замечательной библиотеки Spyce (одна из реализаций Server Page на Python).

Листинг 9.5. Реализация серверного скрипта на языке Python при помощи библиотеки Spyce

```
[[\
import bhv.util
response.setContentType('text/xml; charset=windows-1251')
response.uncacheable()

response.write('<?xml version="1.0" encoding="windows-1251"?>\n\n')
response.write('<response>')

table = request.getpost1('table')
keyColumn = request.getpost1('keyColumn')
displayValueColumn = request.getpost1('displayValueColumn')
searchValueColumn = request.getpost1('searchValueColumn')
count = request.getpost1('count')
command = request.getpost1('command', '')
currentKey = aovc.py.util.encodeUTF8(request.getpost1('currentKey', ''))
currentSearchValue =
aovc.py.util.encodeUTF8(request.getpost1('currentSearchValue', ''))

if command == 'previous':
    result = db.engine.execute('select'
        ' %(keyColumn)s as field1, %(displayValueColumn)s as field2,'
        ' %(searchValueColumn)s as field3 from %(table)s'
        " where %(keyColumn)s = '%(currentKey)s'"
        ' order by field3, field1 limit 1' % vars())
    currentSearchValue = result.fetchall()[0].field3
    result = db.engine.execute('select'
        ' %(keyColumn)s as field1, %(displayValueColumn)s as field2,'
        ' %(searchValueColumn)s as field3 from %(table)s'
        " where %(searchValueColumn)s < '%(currentSearchValue)s'"
        " or %(searchValueColumn)s = '%(currentSearchValue)s'"\
        " and %(keyColumn)s <= '%(currentKey)s'"
        ' order by field3 desc, field1 desc limit %(count)s' % vars())
    if result:
        result = result.fetchall()
        result.reverse()
        for field1, field2, field3 in result:
```

```
            response.write('<row>'
                '<field>%(field1)s</field>'
                '<field>%(field2)s</field>'
                '<field>%(field3)s</field>'
                '</row>' % vars())
    response.write('</response>')
    raise spyceDone
if currentKey == '':
    result = db.engine.execute('select'
        ' %(keyColumn)s as field1, %(displayValueColumn)s as field2,'
        ' %(searchValueColumn)s as field3 from %(table)s'
        " where %(searchValueColumn)s >= '%(currentSearchValue)s'"
        ' order by field3, field1 limit %(count)s' % vars())
else :
    if currentSearchValue == '':
        result = db.engine.execute('select'
            ' %(keyColumn)s as field1, %(displayValueColumn)s as field2,'
            ' %(searchValueColumn)s as field3 from %(table)s'
            " where %(keyColumn)s = '%(currentKey)s'"
            ' order by field3, field1 limit 1' % vars())
        result = result.fetchall()
        if len(result) > 0 :
            currentSearchValue = result[0].field3
    result = db.engine.execute('select'
        ' %(keyColumn)s as field1, %(displayValueColumn)s as field2,'
        ' %(searchValueColumn)s as field3 from %(table)s'
        " where %(searchValueColumn)s > '%(currentSearchValue)s'"
        " or %(searchValueColumn)s = '%(currentSearchValue)s'"
        " and %(keyColumn)s >= '%(currentKey)s'"
        ' order by field3, field1 limit %(count)s' % vars())
if result:
    for field1, field2, field3 in result.fetchall():
        response.write('<row>'
            '<field>%(field1)s</field>'
            '<field>%(field2)s</field>'
            '<field>%(field3)s</field>'
            '</row>\n\n' % vars())
response.write('</response>')
]]
```

Итак, компонент Lookup Combobox разработан и работает, как и предполагалось. Признаюсь, что это мой любимый компонент, который позволяет существенно упростить работу с базами данных. В следующей главе при разработке компонента "Редактируемые таблицы данных" мы используем этот компонент для редактирования данных в таблице при помощи значений из связанной таблицы. Для того чтобы иметь доступ к компоненту Lookup Combobox из других компонентов, необходимо реализовать *интерфейс редактора значений*. В заключение главы рассмотрим его:

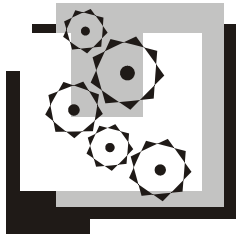
```
setValue: function(value){
  this.valueElement.value = value;
  this.requestedKey = value;
  this.getValueFromServerSync("currentKey=" + encodeURIComponent(value)
    , null, null,1);
},
getValue: function(){
  return this.valueElement.value ;
}
,
show: function() {
  this.element.style.display = "block";
},
hide: function() {
    this.element.style.display = "none";
}
```

Как вы можете убедиться, интерфейс редактора значений очень прост и включает четыре функции-метода. Функция `setValue()` задает значение компоненту при помощи синхронного запроса, потому что сразу после выполнения метода мы должны быть уверены, что компонент имеет установленное значение. Функция `getValue()` возвращает текущее значение свойства `this.valueElement.value`. Наконец, функции `show()` и `hide()` отвечают за отображение компонента в окне Web-браузера. Мы реализуем такой интерфейс не только для компонента Lookup Combobox, но и для других редакторов значений. Например, редактором значений по умолчанию для таблиц будет простое текстовое поле `INPUT`:

```
bhv.Table.TableDefaultEditor.prototype = {
setValue: function(value){
  this.input.value = value;
},
getValue: function(){
```

```
  return this.input.value;
},
show: function() {
  this.element.style.display = "block";
},
hide: function() {
  this.element.style.display = "none";
}
}
```

Такой подход достаточно привычен для программистов на языках программирования, использующих интерфейсы. Теперь вы знаете, что и JavaScript способен обеспечить аналогичную функциональность. В следующей главе при создании компонента "Редактируемые таблицы данных" мы будем широко использовать эту возможность.



Глава 10

Разработка Ajax-компонента "Редактируемые таблицы данных"

При работе с базами данных в Web-приложениях таблицы данных — это тот компонент, который используется чаще всего. Причины этого лежат на поверхности. Ведь реляционные базы данных основаны на таблицах. Технология Ajax может сделать работу с таблицами более удобной как для пользователя, так и для разработчика Web-приложений.

Стандартный HTML-элемент `TABLE` удобно использовать для отображения данных таблицы. Но реализация редактирования данных в таблице до появления технологии Ajax работала довольно тяжеловесно. Эта тяжеловесность касалась как разработчиков, так и пользователей Web-приложений. Вход в режим редактирования таблицы, сохранение данных сопровождались перезагрузкой HTML-документа с неизбежными накладками, связанными с этим. В этой главе мы рассмотрим, как можно организовать редактирование данных при помощи Ajax-компонента "Редактируемые таблицы данных".

Для организации редактирования записей могут использоваться разные подходы. Наиболее легко реализуемым и защищенным от неверных действий пользователя подходом является редактирование записи в отдельном окне, в котором отображается ровно одна запись, и текущее действие всегда можно отметить, нажав кнопку **Cancel**, или зафиксировать на сервере, нажав кнопку **Save**.

В некоторых случаях такая организация редактирования данных таблицы может оказаться слишком громоздкой. Смена окон, переход от представления в виде таблицы к представлению в виде формы ввода может отвлекать пользователя. Поэтому можно применить другой вариант, когда данные таблицы редактируются на месте, прямо в таблице, как это сделано в табличных процессорах. В этой главе мы рассмотрим именно такой способ редактирования данных.

Код компонента на прилагаемом к книге компакт-диске содержится в папке bhvajax\table. Для тестирования компонента "Редактируемые таблицы данных" используется HTML-документ \bhvajax\test\test_table.html.

10.1. Определение конфигурации таблицы данных при помощи XML-документа

В *главе 9* мы разработали компонент Lookup Combobox, для которого имя таблицы и имена полей задавались прямо в программном коде JavaScript при создании компонента. Такое решение могло бы повлечь за собой нарушение безопасности системы, потому что злоумышленник может подставить свои значения этих параметров и получить доступ к произвольной таблице базы банных. Для того чтобы избежать этого, мы создали пользователя сервера баз данных MySQL, от имени которого производились запросы к серверу баз данных в компоненте Lookup Combobox. Этому пользователю были предоставлены минимальные права только на выполнение SQL-команды `SELECT` для конкретных полей необходимых таблиц.

При создании компонента "Редактируемые таблицы данных" мы используем немного другой подход. Мы не будем определять имя таблицы и полей таблицы в HTML-документе или программном коде JavaScript. Конфигурацию компонента "Редактируемые таблицы данных" мы поместим в XML-документ, расположенный на Web-сервере. И клиент, и сервер смогут прочитать этот документ и создать на основании этого документа компонент "Редактируемые таблицы данных". При этом злоумышленник не сможет заставить Web-сервер прочитать данные из таблицы, которая не была предусмотрена конфигурацией XML-документа. Очевидно, что данные этого XML-документа будут нужны как Web-серверу для формирования запросов к серверу баз данных, так и Web-браузеру для формирования внешнего вида таблицы.

Прежде чем рассмотреть пример конфигурационного документа, оценим, насколько защищено такое решение. Как я уже сказал, злоумышленник не сможет запросить данные, отличные от тех, которые указаны в конфигурации в XML-документе, однако он может узнать URL-адрес XML-документа с конфигурацией, обеспечив себе доступ к таблице, для которой определена конфигурация. Для защиты данных следует в конфигурации в XML-документе задать список пользователей или групп пользователей, которым разрешен доступ к соответствующим данным. С примером конфигурации в XML-документе вы можете ознакомиться по листингу 10.1.

Листинг 10.1. Пример конфигурации таблицы данных в XML-документе

```
<?xml version="1.0" encoding="windows-1251"?>

<table_definition>
```

```
<groups>
  <group>admins</group><group>group001</group>
</groups>

<users>
  <user>admin</user><user>user001</user>
</users>

<sql>
select kod_order, kod_product,count,price, p.name as name,
  concat('kod_order=', kod_order, ' and ', 'kod_product=', kod_product)
  as where_clause from orders_details od inner join products p
  on od.kod_product = p.kod
</sql>

<update>
  update orders_details set kod_order=$kod_order,
         kod_product=$kod_product,count=$count,price=$price
</update>
<count>
 5
</count>

<order>
 order by kod_order, kod_product
</order>

<whereClause>
 kod_order=$kod_order and kod_product=$kod_product
</whereClause>

<columns>

<column>
<header>
Номер&lt;br&gt;документа
</header>
<value>
kod_order
```

```
</value>
</column>

<column>
<header>
Товар
</header>
<displayValue>
name
</displayValue>
<value>
kod_product
</value>
<editor>
new bhv.Combobox("combobox1", "comb1", 44685, 10,
  "products", "kod", "name", "search");
</editor>
</column>

<column>
<header>
Количество
</header>
<value>
count
</value>
</column>

<column>
<header>
Цена
</header>
<value>
price
</value>
</column>

</columns>

</table_definition>
```

XML-документ с конфигурацией имеет имя `table_definition`, которое совпадает с именем корневого элемента. В XML-элементе `sql` определен SQL-запрос к серверу баз данных. Я пытался сделать конфигурацию как можно более простой и поэтому ввел некоторые соглашения об именовании. Например, в SQL-запросе должно задаваться специальное поле `where_clause`, которое служит для идентификации строки при обновлении таблицы базы данных:

```
concat('kod_order=', kod_order,' and ','kod_product=', kod_product)
  as where_clause from orders_details od inner join products p
```

> ***ЗАМЕЧАНИЕ***
>
> В этой главе термин "таблица" применяется в нескольких значениях. Для того чтобы различить эти словоупотребление, Ajax-компонент будет всегда называться как "Редактируемые таблицы данных". Для таблиц MySQL будет использоваться термин "таблица базы данных". Наконец, HTML-элемент будет называться `TABLE`. В том случае, если при слове "таблица" не будет уточняющих слов, имеется в виду таблица в нестрогом, обыденном понимании этого слова или таблица с точки зрения пользователя, которому при работе с Web-браузером безразлично, что таблица базы данных отображается при помощи HTML-элемента `TABLE`, сгенерированным компонентом "Редактируемые таблицы данных".

XML-элемент `update` содержит оператор SQL, который обновляет данные на сервере без условия `where`:

```
update orders_details set kod_order=$kod_order,
  kod_product=$kod_product,count=$count,price=$price
```

Условие `where` для обновляемой строки данных будет взято из специального поля `where_clause`, которое содержит по нашему соглашению об именовании строку в том виде, как ее необходимо задавать в опции `where` SQL-оператора `SELECT`.

С формированием условия `where` связан также XML-элемент `whereClause`:

```
<whereClause>
 kod_order=$kod_order and kod_product=$kod_product
</whereClause>
```

Если SQL-оператор `update` изменит значение ключевых полей, при помощи значения XML-элемента `whereClause` будет сформировано новое условие `where` с учетом изменившихся ключевых полей запроса.

XML-элемент `count` содержит количество одновременно выводимых в таблице строк (для постраничного вывода данных таблицы).

XML-элемент `order` задает порядок вывода строк запроса в форме, близкой к опции `order by` SQL-оператора `SELECT`:

```
<order>
 order by kod_order, kod_product
</order>
```

XML-элемент `columns` содержит XML-элементы `column` с определением колонок компонента "Редактируемые таблицы данных". Для колонки может быть задан заголовок `header`, значение `value`. Значение колонки (ячейки) может отличаться от выводимого на экран значения колонки (ячейки). Это не исключение, а, скорее, правило при использовании таблиц реляционных баз данных, например, когда значение колонки (ячейки) представлено в виде *Кода*, а на экран требуется вывести полное *Наименование* из связанной таблицы базы данных. Если необходимо задать выводимое на экран значение, отличное от `value`, задается XML-элемент `displayValue`. В приведенном XML-документе (листинг 10.1) задан XML-элемент `displayValue`:

```
<column>
<header>
Товар
</header>
<displayValue>
name
</displayValue>
<value>
kod_product
</value>
<editor>
new bhv.Combobox("combobox1", "comb1", 44685, 10,
  "products", "kod", "name", "search");
</editor>
</column>
```

Для того чтобы поле `name` было доступно, его следует явно задать в XML-элементе `sql`, который содержит SQL-оператор `SELECT` с условием соединения двух таблиц базы данных:

```
<sql>
select kod_order, kod_product,count,price, p.name as name,
  concat('kod_order=', kod_order,' and ','kod_product=', kod_product)
  as where_clause from orders_details od inner join products p
  on od.kod_product = p.kod
</sql>
```

Поскольку мы решили редактировать значения таблицы на месте, для каждой колонки "Редактируемых таблиц данных" можно задать редактор значений. Если редактор значений не задан явно, для колонки будет назначен редактор значений по умолчанию, который представляет собой текстовое поле `INPUT`.

Если редактор значения задается явно, то его определение в теле XML-элемента `editor` должно точно соответствовать оператору создания компонента в программном коде JavaScript:

```
<editor>
new bhv.Combobox("combobox1", "comb1", 44685, 10,
    "products", "kod", "name", "search");
</editor>
```

Предлагаемый XML-документ с конфигурацией компонента содержит сравнительно небольшой объем конфигурационных параметров, заданных в XML-элементах. Известную сложность представляет то, что текст конфигурационных параметров при выполнении программного кода будет встраиваться в исполняемый код. Следовательно, текст XML-документа должен быть записан в строгом соответствии с синтаксисом запросов к базе данных MySQL, языка программирования PHP 5 (XML-элементы `sql`, `update`, `order`, `whereClause`) и синтаксису языка программирования JavaScript (XML-элемент `editor`). Такой подход делает настройку компонента более гибкой, а программный код — более простым. Если вы работали с библиотеками, которые обеспечивают ORM (отображение баз данных на объекты языка программирования), и вспомните все множество конфигурационных параметров, задаваемых конфигурационных XML-документах, вы, наверное, согласитесь, что у автора не было другого пути, кроме как задать "магические" XML-элементы, которые позволяют уйти от сложного разбора элементов и их атрибутов.

10.2. Реализация компонента "Редактируемые таблицы данных" средствами HTML и JavaScript

Как и в случае с компонентом Lookup Combobox, программный код компонента "Редактируемые таблицы данных" слишком объемный для того, чтобы представить полный листинг в тексте книги. С полным текстом программ вы можете ознакомиться на прилагаемом к книге компакт-диске. Поэтому я остановлюсь на наиболее показательных фрагментах кода.

При создании компонента "Редактируемые таблицы данных" оператором JavaScript `new` вызывается функция-метод `init()`:

```
bhv.Table.prototype = {
init: function(element, definition){
  if (! bhv.contentPane)
    bhv.contentPane=bhv_contentPane;
```

```
var the = this;
this.currentOffset = 0;
this.script = script;
this.script = bhv.getApplicationFolder()+"table/table_query.php";
this.definition = definition;
if (typeof element == "string")
  this.element = document.getElementById(element);
else
  this.element = element;

bhv.sendRequest("get", bhv.getApplicationFolder()+this.definition,
   null, false, function(){the.xmlDefinition = this.responseXML;});
try{
  this.count = parseInt(
    bhv.getElementData(this.xmlDefinition, "count"));
}catch(e){
  this.count = 7;
}

this.columns = [];

var xmlColumns = this.xmlDefinition.getElementsByTagName("column");

for (var i = 0; i < xmlColumns.length; i++){
  var xmlColumn = xmlColumns[i];
  var column = {};
  column.header = bhv.getElementData(xmlColumn, "header");
  column.value = bhv.getElementData(xmlColumn, "value")
    .replace(/^\s+|\s+$/g,"");

  try{
    column.displayValue = bhv.getElementData(xmlColumn, "displayValue")
     .replace(/^\s+|\s+$/g,"");
  }catch(e){
    column.displayValue = bhv.getElementData(xmlColumn, "value")
     .replace(/^\s+|\s+$/g,"");
  }
  this.columns[i] = column;

  try{
```

```
      this.columns[i].editor = bhv.getElementData(xmlColumn, "editor");
    }catch(e){
      this.columns[i].editor =
       "new bhv.Table.TableDefaultEditor(table_editor)";
    }
  }

  this.createHtmlTable(this);

  for (var i = 0; i < xmlColumns.length; i++){
    var table_editor = document.createElement("span");
    table_editor.style.position = "absolute";
    this.columns[i].editorComponent = eval(this.columns[i].editor);
    bhv.contentPane.appendChild(table_editor);
    this.columns[i].editorComponent.hide();
    if (this.columns[i].editorComponent.element.parentNode
                !==bhv.contentPane)
      bhv.contentPane
        .appendChild(this.columns[i].editorComponent.element);
  }

  this.getDataFromServer();
}
```

Функция `init()` вызывается с двумя параметрами. В первом параметре передается HTML-элемент, который служит контейнером для компонента "Редактируемые таблицы данных", а второй параметр содержит URL-адрес XML-документа, который хранит конфигурационные параметры нашего компонента. Адрес конфигурационного файла задается относительно *каталога приложения*, поэтому при вызове функции `bhv.sendRequest()` путь к файлу приводится к абсолютному виду:

```
bhv.sendRequest("get", bhv.getApplicationFolder()+this.definition,
   null, false, function(){the.xmlDefinition = this.responseXML;});
```

Аналогично преобразуется имя серверного скрипта, который будет обрабатывать Ajax-запросы в процессе работы компонента "Редактируемые таблицы данных":

```
this.script = bhv.getApplicationFolder()+"table/table_query.php";
```

Ссылка на полученный с Web-сервера конфигурационный XML-документ (объект типа `DOMDocument`) сохраняется в свойстве `xmlDefinition` компонента в обработчике ответа Web-сервера:

```
function(){the.xmlDefinition = this.responseXML;}
```

Запрос к Web-серверу при помощи функции `bhv.sendRequest()` отправляется синхронно, поскольку для дальнейшей работы кода инициализации компонента нам нужно иметь уже загруженный XML-документ, содержащий конфигурационные параметры компонента.

Конфигурационные параметры из XML-документа разбираются методами работы с DOM и преобразуются к более удобному для использования в JavaScript виду:

```
this.columns = [];
var xmlColumns = this.xmlDefinition.getElementsByTagName("column");
for (var i = 0; i < xmlColumns.length; i++){
  var xmlColumn = xmlColumns[i];
  var column = {};
  column.header = bhv.getElementData(xmlColumn, "header");
  column.value = bhv.getElementData(xmlColumn, "value")
    .replace(/^\s+|\s+$/g,"");
```

В программном коде JavaScript разбираются и запоминаются не все XML-элементы конфигурационного XML-документа, а только те, которые необходимы для работы клиентской части приложения. Часть конфигурационных параметров предназначена только для работы серверной части кода, например, XML-элементы `sql` и `update`.

После разбора всех необходимых конфигурационных параметров вызывается функция, создающая HTML-элементы компонента "Редактируемые таблицы данных":

```
this.createHtmlTable(this);
```

Программный код создания HTML-элементов `TABLE`, `TR` и `TD`, при помощи которых компонент "Редактируемые таблицы данных" отображается в окне Web-браузера, содержит вызовы методов DOM Level 1 `createElement()` и `appendChild()`. С ним вы можете познакомиться в файле \bhvajax\table\Table.js. А вот уже нечто более интересное. Так к ячейке привязывается обработчик нажатия кнопки мыши, по которому инициируется редактирование строки таблицы:

```
for (var i = 0; i < table.columns.length; i++){
  field = document.createElement("TD");
  field.onmousedown = function(){
    table.editRow(table, this);
  }
  row.appendChild(field);
}
```

При нажатии кнопки мыши становится доступным для редактирования вся строка таблицы. Для этого каждому столбцу назначается редактор значения (editor), указанный в конфигурационном файле или задаваемый по умолчанию. Вот как происходит назначение редакторов значений в программном коде JavaScript:

```
try{
  this.columns[i].editor = bhv.getElementData(xmlColumn, "editor");
}catch(e){
  this.columns[i].editor =
   "new bhv.Table.TableDefaultEditor(table_editor)";
}
```

В этом фрагменте кода редактор значения еще не создается как объект JavaScript, а только задается строка, которая затем будет выполняться встроенной JavaScript-функцией `eval()`:

```
for (var i = 0; i < xmlColumns.length; i++){
  var table_editor = document.createElement("span");
  table_editor.style.position = "absolute";
  this.columns[i].editorComponent = eval(this.columns[i].editor);
  bhv.contentPane.appendChild(table_editor);
  this.columns[i].editorComponent.hide();
  if (this.columns[i].editorComponent.element.parentNode
             !==bhv.contentPane)
    bhv.contentPane
        .appendChild(this.columns[i].editorComponent.element);
}
```

Для колонок, которым явно не назначен редактор значений, создается редактор значений по умолчанию, реализованный в виде HTML-элемента `INPUT`:

```
new bhv.Table.TableDefaultEditor(table_editor);
```

Программная реализация редактора по умолчанию проста:

```
bhv.Table.TableDefaultEditor = function(element){
  this.element = element;
  this.input = document.createElement("INPUT");
  this.input.onkeyup = function(event0){
    event0 = arguments[0] || window.event;
    if (event0.keyCode == bhv.key.ENTER){
      bhv.selectNextInput(this);
      return false;
```

```
    }
  }
  this.input.type = "TEXT";
  this.element.appendChild(this.input);
}
bhv.Table.TableDefaultEditor.prototype = {
setValue: function(value){
  this.input.value = value;
},
getValue: function(){
  return this.input.value;
},
show: function() {
  this.element.style.display = "block";
},
hide: function() {
    this.element.style.display = "none";
}
}
```

Редактор по умолчанию реализует минимальный интерфейс редактора значений, который имеет четыре метода: `setValue()`, `getValue()`, `hide()` и `show()`. Напоминаю, что точно такой же интерфейс мы реализовали и в компоненте Lookup Combobox из *главы 9*:

```
setValue: function(value){
  this.valueElement.value = value;
  this.requestedKey = value;
  this.getValueFromServerSync("currentKey=" + encodeURIComponent(value),
    null, null,1);
},
getValue: function(){
  return this.valueElement.value;
},
show: function(){
  this.element.style.display = "block";
},
hide: function(){
  this.element.style.display = "none";
}
```

Теперь вы можете самостоятельно реализовать редактор значений, имеющий эти четыре метода, и использовать его с компонентом "Редактируемые таблицы данных". Это может быть календарь для ввода значений типа `Date`. Впрочем, такой редактор для даты может показаться вам слишком сложным для начала работы, поэтому вы можете реализовать более простой редактор значений. Например, всем известно неудобство, связанное с тем, что в качестве десятичной точки может использоваться точка или запятая, в зависимости от региональных настроек и текущей раскладки клавиатуры. Реализация редактора числовых значений может исключить ошибки ввода, связанные с этим, дополнив функцию-метод `getValue()`:

```
getValue: function(){
  return this.valueElement.value.replace(/,/, ".");
},
```

Разумеется, для правильной работы редактора числовых значений придется проделать еще очень много работы. Следует отменить ввод с клавиатуры нечисловых символов и учесть возможность присвоить редактору неправильное значение методами вставки текста из буфера (<Ctrl>+<V>).

При нажатии кнопки мыши на строке таблицы вызывается функция `editRow()`, которая инициирует процесс редактирования строки:

```
editRow: function(table, ceil){
  var index = this.getCeilIndex(ceil)[0];
  var column = this.getCeilIndex(ceil)[1];
  this.currentRow = index;
  this.currentValues = {};
  this.currentEditors = [];
  this.whereClause = this.data[index]["where_clause"];
  var htmlRow = ceil.parentNode;
  var htmlCeils = htmlRow.getElementsByTagName("TD");
  for (var i = 0; i < htmlCeils.length; i++)
    if (htmlCeils[i] !== ceil)
      try{
        var editor = table.getCeilEditor(htmlCeils[i]);
        this.currentEditors[this.currentEditors.length] = editor;
        this.currentValues[editor.tableField] = editor.getValue()
      } catch(e){}
  var editor = table.getCeilEditor(ceil)
  editor.input.focus();
  if (editor.input.onclick)
```

```
      editor.input.onclick();
  this.currentEditors[this.currentEditors.length] = editor;
}
```

В первых строках функции `editRow()` определяется индекс текущей строки и колонки и подготавливаются пустые объекты `currentValues` и `currentEditors`, в которых будут сохраняться ссылки на текущие значения полей таблицы и объекты редакторов значений. Далее, функция `getCeilEditor()` инициализирует редактор значений правильным значением и визуально выравнивает его по границам соответствующей ячейки таблицы:

```
getCeilEditor: function(ceil){
  var index = this.getCeilIndex(ceil);
  var editor = this.columns[index[1]].editorComponent;
  editor.tableField = this.columns[index[1]].displayValue;
  if (editor.setValue)
    editor.setValue(this.data[index[0]][this.columns[index[1]].value]);
  editor.input.style.position = "absolute";
  editor.input.style.left = bhv.left(ceil,true) + "px";
  editor.input.style.top = bhv.top(ceil,true) + "px";
  editor.input.style.width = ceil.offsetWidth + "px";
  editor.show();
  return editor;
}
```

Как и в случае с компонентом Lookup Combobox, данные компонента хранятся в *Модели данных* и присваиваются редактору значений при помощи интерфейсной функции-метода `setValue()`:

```
if (editor.setValue)
  editor.setValue(this.data[index[0]][this.columns[index[1]].value]);
```

Модель данных компонента "Редактируемые таблицы данных" имеет более простой интерфейс, чем Модель данных компонента Lookup Combobox:

```
bhv.Table.TableData = function(json){
  this.data = eval(json);
}
```

Данные формируются на сервере скриптом PHP 5 и возвращаются в виде объекта в JSON-нотации:

```
$result = $mysqli->query(
    "$table_sql $table_order limit $table_count offset $table_offset");
#echo json_encode($mysqli->store_result());
echo "([";
```

```
$my_counter = 0;
while ($result and $row = $result->fetch_array()) {
  if ($my_counter == 0){
    $my_counter++;
    echo '{';
  }else {
    echo ', {';
  }
  $i=0;
  foreach ($row as $key => $value){
    if ($i++ == 0){
      echo "$key : '$value'";
    } else {
      echo ", $key : '$value'";
    }
  }
  echo '}';
}
echo "])";
```

Возвращаемый объект является массивом с количеством элементов, не превышающим количество строк, отображаемых в таблице. Я не случайно сказал, что количество элементов не превышает заданное количество строк, ведь оно может быть меньше, если таблица базы данных содержит меньше строк, чем допускается для вывода в компоненте "Редактируемые таблицы данных".

В PHP 5 имеются функции для работы с объектами в JSON-нотации — `json_encode()` и `json_decode()`, которые я планировал использовать в компоненте. Но оказалось, что один из сборщиков операционной системы Linux, на которой я отлаживал компонент, не скомпилировал модуль, который поддерживает работу этих двух функций, поэтому я решил реализовать формирование JSON-объекта "вручную". Вариант с использованием функций `json_encode()` и `json_decode()` я оставил в комментариях, и вы можете воспользоваться им, изменив программный код компонента.

10.3. Сохранение данных таблицы на сервере

После того как инициирован процесс редактирования ячеек таблицы, вся работа сосредотачивается в компонентах редакторов значений. Извлекать измененные значения из редакторов значений мы будем интерфейсной функцией-

методом `getValue()`. Для прекращения редактирования строки и сохранения измененных значений на сервере в базе данных мы предусмотрели в каждой строке таблицы кнопку **save** (рис. 10.1). Для того чтобы отменить редактирование строки без сохранения данных, предусмотрена кнопка **cancel**. Обе кнопки реализованы при помощи HTML-элементов `SPAN` (а не `INPUT` или `BUTTON`). Элемент `SPAN` более легковесный и не получает фокус ввода. Поэтому вызов функции возможен только при помощи нажатия кнопки мыши. Это исключает случайное нажатие кнопки **save**. Программный код JavaScript, создающий кнопки **save** и **cancel**, таков:

```
var td = document.createElement("TD");
field = document.createElement("SPAN");
field.className = "silverButton";
field.innerHTML = "save";
field.onclick = function(){
  var the=this;
  this.className="pressedSilverButton";
  table.saveCurrentRow();
  window.setTimeout(function(){the.className="silverButton";},1000);
}
td.appendChild(field);

field = document.createElement("span");
field.innerHTML="cancel";
field.className = "silverButton";
field.onclick = function(){
  var the=this;
  this.className="pressedSilverButton";
  table.cancelCurrentRow();
  window.setTimeout(function(){the.className="silverButton";},1000);
}
td.appendChild(field);
row.appendChild(td);
```

Для анимации нажатия кнопок **save** и **cancel** используются стилевые классы `silverButton` и `pressedSilverButton`. При нажатии кнопки мыши кнопка визуально "вдавливается" при помощи изменения стилевого класса и по таймеру возвращается в исходное состояние.

Сохранение введенных при редактировании значений выполняет функция `table.saveCurrentRow()`:

```
saveCurrentRow: function(){
```

```
  var params =
    "command=update&where_clause=" +encodeURIComponent(this.whereClause)
    + "&definition="+bhv.getApplicationFolder() + this.definition;
  for (var i = 0; i < this.currentEditors.length; i++)
    params += "&" + this.columns[i].value + "="
      + encodeURIComponent(this.columns[i].editorComponent.getValue());
  var the = this;
  bhv.sendRequest("get", this.script, params, false,
    this.handleRequestRow, null, [the]);
  for (var i = 0; i < this.currentEditors.length; i++)
    this.currentEditors[i].hide();
}
```

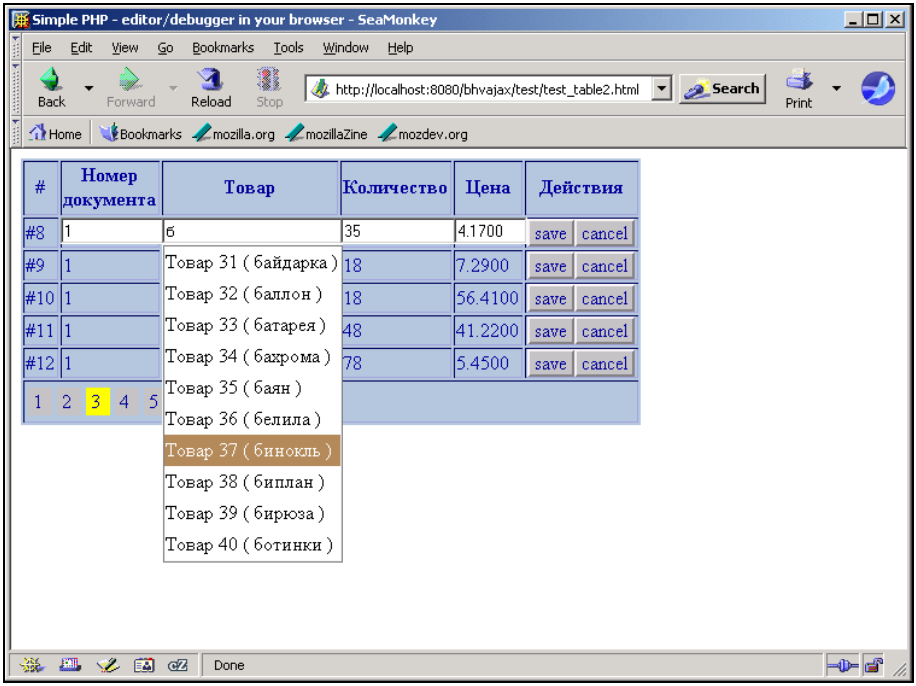


Рис. 10.1. Компонент "Редактируемые таблицы данных" в окне Web-браузера

Эта функция формирует строку параметров для http-запроса. Параметр `command=update` показывает Web-серверу, что пришел запрос на сохранение данных. В параметре `definition` передается URL-адрес XML-документа, который содержит конфигурацию "Редактируемых таблиц данных".

Среди параметров особо выделяется `where_clause`, содержащий значения ключевых полей записи, для которой выполняется SQL-команда `UPDATE`. Оставшиеся параметры содержат имена переменных, которые будут созданы в PHP 5 для выполнения запроса `UPDATE`:

```
<update>
  update orders_details set kod_order=$kod_order,
    kod_product=$kod_product, count=$count,price=$price
</update>
```

В скрипте PHP 5 на основании этих параметров будут созданы переменные при помощи программного кода, характерного для PHP.:

```
foreach ($_REQUEST as $key => $value)
  $$key = $mysqli->real_escape_string($value);
if (isset($_REQUEST["command"])  && ('update' == $_REQUEST["command"])){
  $table_update = $table_definition->getElementsByTagName('update')
    ->item(0)->firstChild->data;
  eval("\$table_update = \"$table_update\";");
  $result = $mysqli->query("$table_update where $where_clause");
  eval("\$table_where_clause = \"$table_where_clause\";");
  $result = $mysqli->query("$table_sql where $table_where_clause");
```

Сначала для каждого параметра запроса создается одноименная переменная, значение которой сразу же защищается специальной функцией модуля `mysqli`:

```
$$key = $mysqli->real_escape_string($value);
```

Далее считывается XML-элемент `update` из конфигурационного XML-документа:

```
$table_update = $table_definition->getElementsByTagName('update')
    ->item(0)->firstChild->data;
```

Потом строка, полученная из файла конфигурации, вычисляется с текущими значениями параметров запроса:

```
eval("\$table_update = \"$table_update\";");
```

Наконец, выполняется запрос на изменение данных на сервере баз данных:

```
$result = $mysqli->query("$table_update where $where_clause");
```

Переменная `$where_clause` содержит значение ключа изменяемой записи до изменения записи и задается в одноименном http-параметре запроса. Разрабатываемый компонент "Редактируемые таблицы данных" допускает,

что ключевые параметры также могут измениться. С учетом этого вычисляется новое значение ключевых параметров:

```
eval("\$table_where_clause = \"$table_where_clause\";");
```

Переменная `$table_where_clause` содержит значение XML-элемента `whereClause` из конфигурационного XML-документа:

```
<whereClause>
 kod_order=$kod_order and kod_product=$kod_product
</whereClause>
```

После изменения данные из таблицы запрашиваются по новому значению ключевых полей:

```
$result = $mysqli->query("$table_sql where $table_where_clause");
```

и отправляются в виде JSON-объекта в Web-браузер.

10.4. Постраничный вывод информации в таблице данных

На каждой Web-странице без полос прокрутки может быть отображено около десяти строк таблицы. Как правило, таблицы баз данных содержат на несколько порядков больше записей, чем может поместиться на одном экране. Поэтому в большинстве случаев таблицы отображают постранично. По умолчанию выводится первая страница, и пользователю предоставляют средства для навигации. Давайте рассмотрим, какие средства навигации могут быть использованы.

Может быть организована панель управления, похожая на стандартную панель навигации в базах данных. Такая панель предусматривает переход на первую страницу, на последнюю страницу, листание на страницу вперед и на страницу назад. Может быть реализовано более мелкое смещение на одну строку вверх или вниз. Для перехода на произвольную страницу иногда предусматривают поле для ввода номера страницы, на которую следует осуществить переход.

Другой способ — прокрутка данных при помощи полоски скроллинга или компонента `slidebar`. Переход между страницами становится интуитивно понятным. Но для очень больших таблиц малое смещение на полоске прокрутки дает слишком большое количество пролистываемых страниц. Это не позволяет точно попасть на требуемую страницу и создает у пользователя негативное ощущение неуправляемости компонента.

Кроме этого, можно реализовать переход на страницы при помощи ссылок или кнопок с номерами страниц, выбирая требуемую страницу по номеру.

Если страниц очень много, например порядка тысячи, создание кнопки или ссылки для перехода на каждую страницу приведет к большим накладным расходам. Поэтому придумывают алгоритмы сокращения списков. Например, первые десять страниц от текущей страницы идут подряд от 1 до 10. Последующие — с интервалом 10 от 10 до 100. Далее с интервалом 100 от 100 до 1000 и т. д. В приведенном примере шкала задана от текущей страницы с номером 1. Но при выборе очередной текущей страницы шкала смещается, но принцип ее организации остается тем же. Тогда для того чтобы выбрать страницу 2781, нам придется выбрать страницу 2000, затем страницу 2700, 2780 и, наконец, 2781. Это чисто умозрительный пример. Реализация подобной навигации по страницам таблицы в реальных приложениях в большинстве случаев потребует больших усилий и от разработчика, и от пользователя, чтобы получить нужную страницу.

Для того чтобы генерировать список страниц для отображения ссылок, разработаем функцию, которой в первом параметре `page` будет предаваться номер текущей страницы, а во втором параметре `count` — общее количество страниц:

```
function bhv$table$pages(page, count){
  var pages=[];
  pages.push(page);
  var factor = 1;
  while (pages[0]>1 || pages[pages.length-1] < count){
    for(var i= -1; pages[0] !=1 && i > -5; i--){
       var current = page + i * factor;
       if (current < 1)
         current = 1;
       pages.unshift(current);
       if (current == 1)
         break;
    }
    for(var i= 1; pages[pages.length-1]!=count && i < 5; i++){
       var current = page + i * factor;
       if (current > count)
         current = count;
       pages.push(current);
       if (current == count)
         break;
    }
    factor *=10;
```

```
    page = Math.round(page/factor)*factor;
  }
  return pages;
}
```

Функция `bhv$table$pages()` формирует и возвращает оператором `return` массив, содержащий список страниц в таком виде, что страницы, близкие к текущей, идут подряд, более далекие с интервалом в 10 страниц, далее 100 страниц 1000 и т. д., пока не будет достигнут номер последней страницы. На основе этого списка в нижней части таблицы будут отображены HTML-элементы `SPAN` с номерами страниц, попавших в список (см. рис. 10.1). Вот как создаются эти элементы в программном коде JavaScript:

```
handleRequest: function(table){
  table.data = eval (this.responseText);
  var count=table.countAll/(table.count-1);
  var pages = table.htmlFooter.childNodes;
if (!table.currentPage)
  table.currentPage = 1
var aPages = bhv$table$pages(table.currentPage,count)
for (var i=pages.length;i<aPages.length;i++){
  var span = document.createElement("SPAN");
  span.innerHTML=i+" "
  span.className = "page"
  table.htmlFooter.appendChild(span)
  span.onmousedown = function(){
    table.currentPage = parseInt(this.innerHTML);
    table.getDataFromServer(table.currentPage);
  }
}

for (var i=0;i<aPages.length;i++){
  pages[i].innerHTML= aPages[i]
  if (aPages[i] == table.currentPage)
    pages[i].className = "selectedPage";
  else
    pages[i].className = "page";
}
  table.displayHtmlTable(table);
}
```

Каждый из созданных HTML-элементов SPAN содержит номер страницы, который при вызове функции-обработчика события onclick извлекается функцией parseInt() и передается в функцию table.getDataFromServer():

```
getDataFromServer: function(page){
if (! page)
  page = 1;
var queryString = "page=" + page +
    "&definition="+bhv.getApplicationFolder() + this.definition
var the = this;
bhv.sendRequest("get", this.script, queryString, true,
        this.handleRequest, null, [the]);
}
```

На основании переданного в качестве параметра функции номера страницы формируется строка http-запроса, содержащая параметр page. Этот параметр разбирается серверным скриптом PHP 5, после чего из базы данных выбирается только необходимая информация по запрошенной странице:

```
if (isset($_REQUEST['page'])){
  $table_offset = ($_REQUEST['page'] - 1) * ($table_count - 1);
}
echo "table.currentOffset = $table_offset;";
$result = $mysqli->query(
  "$table_sql $table_order limit $table_count offset $table_offset");
```

Задание в SQL-операторе SELECT уточнений limit и offset предусмотрено далеко не всеми серверами баз данных, потому что эта возможность не предусмотрена стандартом SQL-92. Как правило, серверы баз данных позволяют разработчику делать выборки ограниченного количества записей. Но в зависимости от конкретного используемого сервера используется различный синтаксис. Например, не limit 10, а top 10. Гораздо меньшее количество серверов баз данных позволяет не только ограничивать количество строк в запросе, но и задать смещение (offset) от начала выборки. Отсутствие опций limit и offset в стандартном операторе SQL SELECT имеет под собой серьезную почву. Как мы рассматривали в *главе 7*, таблица базы данных в теории реляционных баз данных представляет неупорядоченный набор строк. Существует мнение, что необходимость использования номеров строк свидетельствует об ошибках в структуре базы данных. Тем не менее, поддержка листания таблиц в Web-приложениях при помощи баз данных, которые не позволяют задавать смещение, приводит к более сложному программному коду и усилению нагрузки на сервер баз данных и на Web-сервер.

Нам остается рассмотреть, как мы получаем общее количество записей, чтобы отобразить в подножии таблицы ровно столько страниц, сколько данных имеется в таблице. Можно было бы задать еще один XML-элемент в конфигурационном файле. Но мне показалось, что добавление еще одного параметра в конфигурационный файл усложнит конфигурирование компонента. И я решил воспользоваться регулярными выражениями для извлечения подстроки из строки, заданной в XML-элементе `sql`:

```
$result = $mysqli->query( preg_replace( "/[\s\S]* from /i",
  "select count(*) as count_all from ",
  "$table_sql"));
$row = $result->fetch_array();
echo "table.countAll = {$row['count_all']}; \n";
```

Я убираю из запроса SQL `SELECT` весь текст от начала до слова `from` включительно и заменяю этот текст новым текстом:

```
select count(*) as count_all
```

Теперь я получаю значение из запроса и выполняю оператор `echo`, который формирует фрагмент кода JavaScript:

```
echo "table.countAll = {$row['count_all']}; \n";
```

Этот фрагмент кода будет выполнен при помощи встроенной функции JavaScript `eval()`, а Web-браузер узнает точное число строк, которые содержатся в таблице, и вычислит на основании количества строк количество страниц, на которых может быть отображена таблица:

```
table.data = eval (this.responseText);
var count = table.countAll / (table.count - 1) + 1;
```

В выражении свойство `count` уменьшается на единицу: `table.count - 1`. Это сделано для того, чтобы при листании последняя строка предыдущей страницы отображалась как первая строка следующей страницы. Мне такой подход к организации листания представляется более интуитивно понятным для пользователя. Пользователь получает уверенность, что перед ним верные данные и он не пропустил страницу или несколько строк таблицы.

10.5. Серверная часть компонента "Редактируемые таблицы данных"

Мы уже рассмотрели многое из того, что касается серверной части кода, когда обсуждали реализацию программного кода JavaScript. Но некоторые моменты еще не были освещены, поэтому рассмотрим их.

В начале скрипта PHP 5 мы задаем параметры соединения с базой данных и присоединяемся к базе данных:

```
$host = 'localhost';
$database = 'bhvdb';
$username = 'bhv';
$password = 'd57LU293gxz';

$mysqli = new mysqli($host, $username, $password);
$mysqli->query("set character set cp1251");
$mysqli->select_db($database);
```

После этого мы создаем объект для работы с конфигурационным XML-документом, загружаем этот XML-документ методом `load()` и разбираем его средствами работы с DOM:

```
$table_definition = new DOMDocument();
$table_definition->load($_REQUEST["definition"]);
$table_sql = $table_definition->getElementsByTagName('sql')
  ->item(0)->firstChild->data;
$table_count = $table_definition->getElementsByTagName('count')
  ->item(0)->firstChild->data;
$table_order = $table_definition->getElementsByTagName('order')
  ->item(0)->firstChild->data;
$table_where_clause = $table_definition
  ->getElementsByTagName('whereClause')
  ->item(0)->firstChild->data;
```

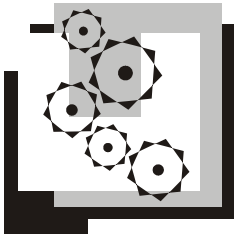
Как я уже говорил, потенциально опасным решением является то, что клиент при обращении к Web-серверу задает имя конфигурационного XML-документа в коде JavaScript. Злоумышленник может воспользоваться этой особенностью программного решения и запросить данные, задав самостоятельно нужный ему адрес конфигурационного XML-документа. Для того чтобы избежать этого, мы договорились в конфигурационном XML-документе задавать список пользователей или групп пользователей, которым разрешен доступ к компоненту "Редактируемые таблицы данных". Мы не реализовали программную обработку списка пользователей, поскольку связь с вопросами аутентификации пользователей в этой главе значительно усложнила бы и без того сложное программное решение. И, главное, мы рассмотрели в книге несколько различных способов аутентификации, для каждого из которых код защиты существенно различался.

Для реально работающего приложения можно рекомендовать сделать другой вариант защиты. Адрес конфигурационного файла лучше задать не в HTML-документе, а в скрипте PHP 5:

```
<?php
$table_definition = new DOMDocument();
$table_definition->load('definition.xml');
require("../table/table_query.php");
?>
```

Теперь в качестве адреса серверного скрипта следует задавать не table\table_query.php, а адрес скрипта, где определен адрес документа с конфигурацией, в котором файл table\table_query.php выполняется при включении кода скрипта table\table_query.php оператором `require()`. Для этого скрипта можно установить настройку доступа и тем самым отсечь доступ злоумышленников. Именно таким способом я и пользуюсь в реальных приложениях. Но для материала книги такой способ показался мне слишком уж запутанным. Поэтому я немного переделал программный код, в котором адрес конфигурационного XML-документа задается из JavaScript и передается Web-серверу в http-параметре запроса.

Впрочем, это позволило нам немного поговорить о защищенности решений при реализации "Редактируемых таблиц данных" и рассмотреть несколько вариантов защиты. Теперь вы, добившись работы упрощенного с точки зрения защиты варианта компонента, исходя из действующей на вашем Web-сервере системы защиты, можете, в качестве домашнего задания, усилить программное решение компонента "Редактируемые таблицы данных", сделав его более защищенным.



Глава 11

Модульное программирование на JavaScript

Когда объем программного кода, разработанного на JavaScript, превышает некоторую критическую величину, у разработчика возникает необходимость в упорядочивании процесса разработки. Скрипты JavaScript, состоящие из сотен строк кода, становятся сложными в отладке. Когда разработчик начинает разбивать скрипты на более мелкие модули, возникает другая проблема. Эти модули необходимо загружать при помощи HTML-элементов `SCRIPT` HTML-документа. Заголовки HTML-документов превращаются в нескончаемые списки HTML-элементов `SCRIPT`, в атрибутах `src` которых указаны URL-адреса загружаемых скриптов. Но и это еще не все. Между скриптами, как правило, существуют связи. Для работы одного скрипта необходимо, чтобы были загружены все связанные с ним скрипты. Эти связи приходится учитывать при формировании списка загружаемых скриптов. Если зависимости между скриптами изменились, придется вносить изменения во все HTML-документы, использующие эти скрипты. В некоторых случаях ситуация еще более сложная. Это происходит, когда зависимые скрипты должны быть не просто загружены, но еще и загружены в определенном порядке. То есть проблем с обеспечением модульного программирования в JavaScript хватает.

Стандартные средства JavaScript (ECMAScript-262) не были рассчитаны на поддержку модульного программирования. Но когда у разработчиков возникла в этом потребность, они нашли решения, которые обеспечивают поддержку модульного программирования, не выходя за рамки спецификации JavaScript (ECMAScript-262). В этом проявилась необычайная сила и гибкость языка программирования JavaScript, который еще недавно считался недостойным внимания серьезных разработчиков. В этой главе мы ознакомимся с некоторыми из таких решений и разработаем систему модульного программирования, которая будет, наряду с хорошо известными приемами, содержать и оригинальные решения, которые могут на первый взгляд показаться непривычными. С более традиционной трактовкой вы можете познакомиться в статье автора "Пакеты, модули, загрузчики, пространства имен,

классы и множественное наследование в JavaScript", опубликованной на Web-ресурсе **http://webscript.ru/stories/07/05/10/5470561**.

11.1. Обеспечение модульной разработки в современных библиотеках JavaScript

Я приступаю к изложению материала, который вызывает горячие споры в среде программистов. Среди JavaScript-программистов достаточно распространено мнение, что динамическая (в процессе работы приложения) загрузка модулей JavaScript отрицательно сказывается на работе приложения в среде Web-браузера, а использование классов и тем более наследования — это просто игры в объектно-ориентированное программирование, которое чуждо природе истинного JavaScript. Но, тем не менее, многие библиотеки используют такой подход, и мы рассмотрим, как они это делают. Разумеется, окончательный выбор, как всегда, остается за вами.

В современных языках программирования часто используются такие средства как:

- пространства имен (namespace);
- модули (module);
- классы (class);
- пакеты (package);
- загрузчики (loader).

Эти средства не связаны неразрывно между собой, но, как правило, используются совместно. Рассмотрим, как это все может работать в единой системе.

Модули — это фрагменты программы, которые могут создаваться относительно независимо друг от друга и храниться в отдельных файлах, а при соединении воедино обеспечивают нужную функциональность и решают поставленную задачу. Слово "модуль" дало имя языку программирования Модула. Использование модулей, в свое время, поставило перед разработчиками проблему с именованием переменных. В разных модулях могли использоваться переменные с одинаковыми именами, и это приводило к нежелательным явлениям при совместном использовании таких модулей. Как выход, в некоторых языках программирования каждый модуль действовал в своей (локальной) области видимости переменных, благодаря чему использование одноименных переменных не приводило к недоразумениям. При этом возникала другая проблема. Модуль превращался в некую вещь в себе, к локальному контексту которого нельзя было получить доступ из другого модуля. В качестве решения были предложены публичные интерфейсы

модулей и пространства имен. Публичные интерфейсы модуля включают имена, которые доступны не только в теле модуля, но и извне, из других модулей. Такие имена должны быть уникальны, это достигается путем задания пространства имен. Мы на протяжении всей книги использовали эмуляцию пространств имен при помощи объектов пространства имен JavaScript. Например, наша функция `bhv.sendRequest()` расположена в искусственно организованном пространстве имен `bhv`, а некоторые вспомогательные функции, например, `bhv.util.getXMLHttpRquest()` — в пространстве имен `bhv.util`.

Для того чтобы пространства имен не повторялись у различных разработчиков, рекомендуется использовать доменное имя организации, прочитанное с конца в качестве корневого пространства имен. Например, если мы создаем библиотеку ajax, то могли бы использовать для пространства имен `ru.bhv.ajax` в качестве префикса для всех имен нашей библиотеки. С точки зрения программирования JavaScript, такое решение обладает одним существенным недостатком. Использование каждой точки в имени для JavaScript означает дополнительную операцию доступа к свойству объекта. Поэтому в настоящее время такие строгие именования в библиотеках JavaScript практически не используются. Но в последние годы наблюдается тенденция применять все более и более строго систему именований в профессиональных библиотеках JavaScript. Если первые библиотеки смело создавали на глобальном уровне такие переменные как `Ajax`, `Class`, `Module`, `Element`, что уже сегодня делает некоторые библиотеки несовместимыми, то уже очень скоро библиотеки стали пользоваться именем библиотеки в качестве префикса для всех объектов пространства имен. Так поступают библиотеки dojo, YUI (YAHOO) и многие другие, хотя строгое именование по доменному имени в настоящее время используется все еще достаточно редко.

Для того чтобы легче управлять модулями и пространствами имен внутри своего доменного имени, принято простое отображение имен модулей на имена папок (каталогов) и файлов операционной системы. Например, модуль, в котором используется пространство имен `bhv.util`, будет находиться в папке bhv в файле bhv\util.js.

Когда пришло время использовать классы и объекты, не сразу возникла идея, что каждый класс (или публичный класс, интерфейс) можно хранить в отдельном файле. Тем не менее, многие современные системы поступают именно так (например, Java, Perl, Python). С учетом этого, класс `bhv.widget.BaseWindow` должен храниться в файле bhv\widget\BaseWindow.js. Такое соглашение об именовании позволяет достичь сразу две цели. Во-первых, по имени класса загрузчик кода будет знать, где именно этот класс необходимо искать, и, во-вторых, такое именование создает естественную (на уровне файловой системы) защиту от повторного использования одинаковых

пространств имен (namespace) в разных модулях. Поскольку в файловой системе не может быть двух файлов с одинаковыми путем и именем, разработчик частично гарантирован от повторного использования пространства имен.

Когда было принято соглашение хранить каждый класс или каждый публичный класс в отдельном файле, возникла необходимость в пакетах (package). Пакет — это множество классов, которые расположены в одном пространстве имен и (в некоторых языках программирования) в одной папке (каталоге) файловой системы. Как правило, эти классы имеют функционально связанное назначение, и некоторые языки программирования допускают для классов одного пакета доступ к именам с областью видимости в пределах пакета, что облегчает обмен информацией между классами, входящими в один пакет.

Для того чтобы имена классов можно было отличить от имен пакетов и имен модулей, имена пакетов и модулей записывают, как правило, строчными буквами (в нижнем регистре), а имена классов начинают с прописной буквы (в верхнем регистре).

Как мы убедились, пространства имен, модули, классы, пакеты в современном программировании используются в тесной связи. Такой подход дает дополнительные выгоды для разработчика, упорядочивая программный код приложения и позволяя использовать библиотеки от различных разработчиков, построенные на соблюдении правил об именовании, без опасения переопределить переменные и, тем самым, нарушить работу Web-приложения.

При работе с интерпретатором JavaScript Web-браузера, модуль может быть загружен в интерпретатор путем задания его в атрибуте `src` HTML-элемента `SCRIPT`:

```
<script type="text/javascript" src="../bhv.util.js"></script>
```

Мы уже перечислили проблемы, возникающие при этом. В заголовке HTML-документа приходится задавать большой список HTML-элементов `SCRIPT`, который должен включать не только непосредственно используемые скрипты, но и все скрипты, от которых эти скрипты зависят, иногда в строго определенном порядке. Последнее требование, вообще, труднореализуемо, т. к. порядок выполнения скриптов в некоторых типах Web-браузеров зависит не от того, в каком порядке эти скрипты заданы в HTML-документе, а от того, какой скрипт раньше был загружен в Web-браузер. Особенно это касается HTML-элементов `SCRIPT`, которые созданы при помощи метода `document.createElement("SCRIPT")`. Единственное, на что можно рассчитывать, так это на то, что скрипты, расположенные в теле HTML-элемента `HEAD` HTML-документа, будут загружены до начала отображения элемента `BODY` HTML-документа.

Замечание

При разработке Web-приложения и его тестировании во внутренней сети или на локально работающем Web-сервере, порядок загрузки скриптов, как правило, совпадает с порядком, заданным в HTML-документе. При перенесении Web-приложения в среду Интернета из-за большего времени, требуемого для загрузки скрипта, увеличивается вероятность того, что в некоторых типах Web-браузеров скрипты будут загружены не в том порядке, в котором они заданы в HTML-документе или созданы методом `document.createElement("SCRIPT")`. Неопределенность порядка выполнения загружаемых скриптов может вызвать проявление неожиданных ошибок в работе Web-приложения, которые не могут быть выявлены при тестировании во внутренней сети или на локальном Web-сервере. Синхронная загрузка скриптов запросами `XMLHttpRequest` снимает неопределенность порядка выполнения загружаемых скриптов JavaScript.

Для Ajax-приложений появилась еще одна проблема. Объем JavaScript-кода Ajax-приложений значительно превышает объем JavaScript-кода классических Web-приложений. В отличие от классического Web-приложения, Ajax-приложение работает в пределах одной первоначально загруженной страницы и все запросы к серверу производит без перезагрузки основной страницы. А это означает, что весь код JavaScript, который, возможно, и не будет использован в течение сеанса пользователя, должен быть загружен при первоначальной загрузке страницы. Уже это показывает, что в Ajax-приложении никак не обойтись без загрузки JavaScript-кода динамически, в процессе работы приложения, по мере запроса пользователем тех или иных возможностей Web-приложения. При помощи объекта `XMLHttpRequest` сделать это достаточно просто:

```
function loadScript(scriptPath){
  if (isLoaded(scriptPath))
    return;
  if (window.XMLHttpRequest)
    var xmlHttpRequest = new window.XMLHttpRequest();
  else
    var xmlHttpRequest = new ActiveXObject("Msxml2.XMLHTTP");
  xmlHttpRequest.open("GET", scriptPath, false);
  xmlHttpRequest.send(null);
  eval(xmlHttpRequest.responseText);
}
```

Для загрузки JavaScript-кода в приведенном примере мы использовали синхронный (не асинхронный) запрос, что определяется третьим параметром метода `open()` объекта `XMLHttpRequest`. Для загрузки программного кода JavaScript синхронные запросы более удобны, чем асинхронные, поскольку

сразу после выполнения запроса можно использовать загруженный программный код в тексте программы:

```
loadScript("../combobox/Combobox.js");
var combo1 = new bhv.Combobox("combobox1", "comb1", 7, 10,
     "products", "kod", "name", "search")
```

Если бы загрузка скрипта производилась асинхронным запросом, этот код бы не работал, т. к. на момент выполнения оператора, следующего за вызовом функции `loadScript()`, код загружаемого скрипта еще не был бы выполнен, и обращение к функциям, определенным в загружаемом скрипте, вызвало бы ошибку времени выполнения.

Мы реализуем в нашей библиотеке похожий подход загрузки программного кода JavaScript, но прежде чем перейти к изложению материала, перечислим другие способы, которые используются в библиотеках JavaScript.

Кроме загрузки при помощи объекта `XMLHttpRequest`, можно динамически (по ходу работы Ajax-приложения) загрузить скрипт посредством динамически создаваемых элементов `SCRIPT`, как мы это делали в *главе 8* при помощи функции, код которой я приведу для вашего удобства еще раз:

```
bhv.sendScriptRequest = function(url, httpParams, callback, callbackArgsArray){
  var currentScript =document.createElement("SCRIPT");
  if (httpParams)
    httpParams="?rand=" + Math.random() + "&" + httpParams;
  else
    httpParams="?rand=" + Math.random();
  currentScript.bhv_readyState = false;
  currentScript.onload =
    bhv.util.scriptCallback(currentScript, callback, callbackArgsArray);
  currentScript.onreadystatechange =
    bhv.util.scriptCallback(currentScript, callback, callbackArgsArray);

  currentScript.src = url + httpParams;
  document.getElementsByTagName("script")[0].parentNode
    .appendChild(currentScript);
}

bhv.util.scriptCallback = function(currentScript, callback, callbackArgsArray){
  return function(){
    if (currentScript.bhv_readyState)
      return;
```

```
    if (! currentScript.readyState
            || currentScript.readyState == "loaded"
            || currentScript.readyState == "complete"){
      currentScript.bhv_readyState = true;
      callback.apply(currentScript, callbackArgsArray)
      currentScript.parentNode.removeChild(currentScript);
    }
  }
}
```

Код этой функции немного перегружен для того, чтобы иметь возможность кроссбраузерно вызвать функцию по событию `onload` (или `onreadystatechange`) после полной загрузки скрипта. Если это не требуется, код функции будет совсем простым:

```
bhv.sendSimpleScriptRequest = function(url, httpParams){
  var currentScript =document.createElement("SCRIPT");
  if (httpParams)
    httpParams="?rand=" + Math.random() + "&" + httpParams;
  else
    httpParams="?rand=" + Math.random();
  currentScript.src = url + httpParams;
  document.getElementsByTagName("script")[0].parentNode
    .appendChild(currentScript);
}
```

По сравнению с загрузкой кода синхронным запросом `XMLHttpRequest`, положительным моментом является то, что используются средства, специфицированные в DOM Level 1. Но есть и отрицательные стороны такого подхода:

- скрипт нельзя загрузить синхронно (не асинхронно);
- событие, наступающее при полной загрузке скрипта, к сожалению, не специфицировано в HTML. Поэтому для поддержки кроссбраузерности приходится использовать трюки;
- в процессе работы приложения могут создаваться сотни и тысячи новых элементов `SCRIPT`, которые в некоторых Web-браузерах нельзя повторно использовать для загрузки новых скриптов (организовав что-то вроде пула HTML-элементов `SCRIPT`). Даже после удаления из HTML-документа методом `removeChild()` эти HTML-элементы `SCRIPT` могут не удаляться из памяти.

Впрочем, наиболее существенным недостатком является как раз невозможность задать синхронное (не асинхронное) выполнение загружаемого в HTML-

элемент SCRIPT программного кода JavaScript. Особенностью использования HTML-элемента SCRIPT является возможность кроссдоменных запросов.

Аналогично HTML-элементу SCRIPT для динамической загрузки скриптов можно использовать HTML-элемент IFRAME. В HTML-элемент IFRAME загружается не скрипт JavaScript, а HTML-документ, в котором программный код JavaScript задается между открывающим тегом <SCRIPT> и закрывающим тегом </SCRIPT>. Использование HTML-элементов IFRAME, так же как и HTML-элементов SCRIPT, исключает синхронное выполнение загружаемых скриптов. Но в отличие от HTML-элемента SCRIPT, HTML-элементы IFRAME можно неоднократно использовать повторно для загрузки очередного HTML-документа, и для HTML-элемента IFRAME имеются стандартизированные в спецификации HTML обработчики событий, наступающие при загрузке документа в IFRAME. Особенностью скриптов, загруженных в HTML-элемент IFRAME, является то, что каждый документ будет иметь свое пространство глобальных имен переменных, и к переменным из родительского документа придется обращаться с префиксом parent.

В заключение рассмотрим еще два варианта загрузки скриптов, которые используются в замечательной библиотеке x.js (Michael Foster, Cross-Browser.com). Во-первых, в библиотеке x.js реализована функция xInclude(), которая выводит в документ при помощи функции document.writeln() определение HTML-элемента SCRIPT. Эта функция реализует проверку, исключающую повторную загрузку одних и тех же скриптов. Разумеется, использование оператора document.writeln() допустимо только в процессе первоначальной загрузки HTML-документа, и применять функцию document.writeln() для динамической загрузки скриптов нельзя.

Второй способ, предлагаемый в библиотеке x.js, также заслуживает особого внимания. Это использование препроцессора, который анализирует применяемые в тексте документа функции библиотеки x.js и зависимости, которые существуют между этими функциями и формируют файл, содержащий только необходимые функции JavaScript. Впрочем, с учетом современной скорости соединения с Интернетом и лаконичности этой замечательной библиотеки такая экономность кажется излишней.

11.2. Работа с пространствами имен в JavaScript

Теперь давайте шаг за шагом создадим загрузчик модулей JavaScript. Наш загрузчик модулей будет загружать модули с определением объектов (собственно, классов). Поэтому наш загрузчик модулей можно было бы назвать загрузчиком объектов (классов), если бы я не боялся навлечь на свою голову

гнев той части программистов JavaScript, у которых одно только слово "класс" применительно к JavaScript вызывает бурный поток эмоций и слов, среди которых слово "прототип" было бы только вторым по частоте употребления. Мы уже выяснили, что модули, классы, пакеты, пространства имен и загрузчики могут применяться в единой системе. И это делает их применение особенно удобным. Поэтому начнем с пространств имен.

Если строго следовать спецификации JavaScript, то нужно сказать, что в JavaScript имеются два контекста имен — глобальное и локальное. К именам глобального контекста принадлежат переменные и функции, определенные вне тела функции. К именам локального контекста принадлежат переменные, определенные с ключевым словом `var` в теле функции, а также формальные параметры функции и внутренние функции, определенные в теле функции.

Подчеркну особо, что операторные скобки не приводят к созданию нового локального контекста, как это бывает в некоторых языках программирования:

```
for (var i =0; i<10; i++) {
  var j = i;
}
alert(j); // j == 9
alert(i); // i == 10
```

В приведенном примере и переменная цикла `i`, и переменная внутри цикла определены в текущем для цикла контексте и не удаляются из памяти после выхода из цикла, как это было бы верно для некоторых языков программирования. Все же программисты нашли способ эмулировать ставшее привычным поведение блоков с локальными именами следующим способом:

```
void function(){
  for (var i =0; i<10; i++) {
    var j = i;
  }
}()
alert(typeof j); // "undefined"
alert(typeof i); // "undefined"
```

Такая конструкция определяет новую анонимную функцию и тут же ("на лету") вызывает ее. Такую химеру используют очень часто, поскольку это единственный способ определить все переменные в локальном контексте и не засорять глобальный контекст новыми именами.

Кроме глобальной и локальной областей видимости, в JavaScript не предусмотрено никаких других областей видимости переменных. Тем не менее,

разработчики уже давно используют объекты JavaScript для хранения имен переменных, что создает полнейшую иллюзию работы с пространствами имен:

```
var bhv = {};
bhv.util = {};
bhv.util.nullFunction = function(){};
```

Такая эмуляция пространств имен имеет два недостатка. Во-первых, объекты пространств имен сложно инициализировать, и, во-вторых, доступ к значениям, хранящимся в таких объектах, требует дополнительной операции доступа к свойству объекта, что может существенно снизить производительность работы приложения.

Давайте решим сначала первую из проблем. Для создания объектов пространства имен можно использовать самую простую функцию, создающую в случае необходимости и возвращающую в качестве значения объект пространства имен по его строковому представлению, например `"bhv.util"`:

```
bhv.ns = function(sNamespace) {
  var aNamespace = sNamespace.split(".");
  var currentNamespace = "";
  var oNamespace = null;
  for (var i = 0; i < aNamespace.length; i++) {
    if (i == 0)
      currentNamespace += aNamespace[i];
    else
      currentNamespace += "."+aNamespace[i];
    if (eval("typeof("+currentNamespace + ")") == "undefined")
      oNamespace = eval(currentNamespace + "= new Object()");
    else
      oNamespace = eval(currentNamespace);
  }
  return oNamespace;
}
```

Приведенная функция разбивает строку, содержащую строковое представление пространства имен по разделителю "точка", и создает массив типа `Array`, который содержит строки с именами, составляющими пространство имен. Далее, в цикле эти имена перебираются, и проверяется наличие в памяти интерпретатора JavaScript-объекта с заданным именем:

```
if (eval("typeof("+currentNamespace + ")") == "undefined")
  oNamespace = eval(currentNamespace + "= new Object()");
```

```
else
  oNamespace = eval(currentNamespace);
```

Если объект пространства имен уже был создан ранее, он просто возвращается после выхода из цикла:

```
return oNamespace;
```

Если объект с заданным именем не был еще создан, то создается новый объект:

```
oNamespace = eval(currentNamespace + "= new Object()");
```

Давайте посмотрим трассировку выполнения этой функции при первом вызове для такого пространства имен:

```
bhv.ns("bhv.combobox");
```

Сначала функция `split()` создаст массив, содержащий строки, составляющие пространство имен `["bhv", "combobox"]`. Далее будет выполнена проверка:

```
if (eval("typeof(bhv)") == "undefined")
```

Эта проверка вернет логическое значение `false`, потому что объект `bhv` был нами проинициализирован в самом начале скрипта util.js, который мы договорились подключать в начале каждого HTML-документа. Поэтому новый объект `bhv` не создается. Далее проверка условия:

```
if (eval("typeof(bhv.combobox)") == "undefined")
```

вернет логическое значение `true`, потому что при первом вызове функции `bhv.ns()` с параметром `"bhv.combobox"` такой объект пространства имен не существует, следовательно, будет выполнен код:

```
oNamespace = eval("bhv.util = new Object()");
```

Этот фрагмент практически эквивалентен выполнению следующего кода:

```
oNamespace = bhv.util = new Object();
```

или, более подробно (с учетом правой ассоциативности операции присваивания):

```
bhv.util = new Object();
oNamespace = bhv.util;
```

Этот объект и будет возвращен оператором `return` в качестве результата вызова функции `bhv.ns("bhv.combobox")`.

Итак, объект пространства имен `bhv.combobox` создан. Но все же остается второй недостаток применения таких объектов, который заключается в том, что использование каждой точки означает дополнительный оператор извлечения свойства из объекта. И этот лишний оператор может существенно

сказываться на эффективности работы скрипта. Как же решить эту проблему? Оказалось, что программисты сами создали эту проблему и пытаются теперь ее решить (случай не редкий в программировании). Что я имею в виду, вы сейчас поймете. На самом деле, нет необходимости использовать именно точку в качестве разделителя имен в объекте пространств имен. Для этого подойдет любой другой знак. Например, это может быть знак подчеркивания. Но я выбрал знак $, потому что подчеркивание часто используют для разделения составляющих сложных имен. С учетом этого, вместо имени пакета `bhv.combobox` и имени класса `bhv.combobox.Combobox` следует использовать просто обычные имена переменных `bhv$combobox` и `bhv$combobox$Combobox`. Ведь единственная цель хранения имен в объектах пространств имен — упорядочить глобальные имена и исключить переопределение объектов при одновременном использовании двух и более библиотек от разных разработчиков.

> ***Замечание***
>
> Сложности, возникающие при совместном использовании двух и более библиотек от разных разработчиков, — это не перестраховка, а реальность. Существуют библиотеки, и даже хорошие библиотеки, которые используют имена глобальных переменных Ajax, Class, Module, Element, Form и другие имена, которые можно назвать тривиальными. Естественно, такие библиотеки совместно, без специальных мер, без специальных ухищрений, применять нельзя. Второй проблемой совместного использования двух и более библиотек разных разработчиков является определение дополнительных методов для встроенных объектов JavaScript — `Object`, `Array`, `String`, `Number`, `Date`, `Boolean`. Конечно, для определения таких методов также следует использовать префиксы в стиле `SomeObject.bhv$someMethod()`, которые позволяют каждому разработчику действовать в своем пространстве имен, не разрушая работу программного кода библиотек других разработчиков.

Все же необходимо учитывать и традиции разработчиков, привыкших видеть имена пакетов, разделенные точками, а не знаком $. Поэтому я разработал функцию, которая должна помочь разработчику мыслить уже ставшими привычными объектами пространства имен с точками в качестве разделителей, а получать в итоге имена переменных, в которых имена пространств имен разделены знаком $:

```
bhv.namespace = function(sNamespace){
  var aNamespace = sNamespace.replace(/\./g,"$");
  var oNamespace = null;
  if (eval("typeof("+aNamespace + ")") == "undefined")
    oNamespace = eval(aNamespace + "= new Object()");
  else
    oNamespace = eval(aNamespace);
  return oNamespace;
}
```

Для введенного значения `bhv.namespace("bhv.combobox.Combobox")` будет возвращена ссылка на объект `bhv$combobox$Combobox`.

Вы обратили внимание, что я продолжаю использовать и точечную нотацию, и пишу не `bhv$namespace`, а `bhv.namespace` при определении функции. Дело в том, что я считаю использование знака $ вместо точек в качестве разделителей имен пакетов экспериментальным вариантом, который еще следует обсудить. Пока я не нашел каких-либо недостатков этого варианта в сравнении с применением точек в качестве разделителя имен, кроме того, что такая запись непривычна для программистов JavaScript.

Я включил на прилагаемом к книге компакт-диске две реализации библиотеки для работы с модулями. Одна из библиотек, написанная в более привычном стиле, находится в файле \bhvajax\bhv\classes.js. Эта библиотека использует традиционные объекты пространств имен с именами, разделенными точкой. И рядом же находится новая версия библиотеки, использующая в качестве разделителя имен знак $. Файл, который содержит код этой библиотеки, называется \bhvajax\bhv\classe$.js. Вы можете использовать или ту или другую библиотеку (но не обе одновременно). Код примеров с прилагаемого к книге компакт-диска будет работать совершенно одинаково. Если вам понравится использование вместо точек знака $, вы можете путем незначительных изменений в коде библиотек util.js и classe$.js перейти полностью на новый стиль именования пространств имен с применением знака $ в качестве разделителя. Если вам моя идея покажется ненужной — просто не используйте библиотеку classe$.js и не обращайте внимания на то, что я говорил в этом разделе. Используйте вместо этого библиотеку classes.js. Дальнейшее изложение материала практически не будет зависеть от того, какой из вариантов библиотек, classes.js или classe$.js, вы используете.

11.3. Объекты и наследование в JavaScript

В JavaScript реализована объектная модель, основанная на прототипах, а не на классах. На вершине иерархии объектов находится объект типа `Object`. Для создания нового экземпляра объекта типа `Object` достаточно выполнить один из двух операторов:

```
var object1 = new Object();
var object2 = {};
```

Эти операторы являются в данном фрагменте полными синонимами, но второй вариант позволяет дополнительно инициализировать объект новыми свойствами и функциями-методами:

```
var object3 = {
  prop1: 2,
```

```
  prop2: 2,
  prop1_x_prop2: function(){return this.prop1 * this.prop2;}
}
```

Справа от оператора `new` может присутствовать любая функция JavaScript, определенная пользователем:

```
function Class1(prop1, prop2){
  this.prop1 = prop1;
  this.prop2 = prop2;
}
Class1.prototype. prop1_x_prop2 = function(){return this.prop1 *
this.prop2;}
var object4 = new Class1(2, 2);
var object5 = new Class1(3, 3);
alert(typeof(object5) == "object");
alert(object5 instanceof Class1);
```

У каждой функции существует свойство `prototype`, которое имеет в качестве значения объект типа `Object`. К этому объекту вы можете добавлять новые свойства и функции-методы, и они становятся сразу же доступными для всех уже созданных экземпляров объекта и для тех экземпляров объекта, которые будут создаваться впоследствии, если только это свойство или функцию-метод не переопределить явно для экземпляра объекта:

```
var object6 = new Class1(4, 4);
object6.prop1_x_prop2 = function(){alert("Надо подумать...");}
```

Такое поведение прототипа объекта немного напоминает наследование. Поэтому некоторые библиотеки JavaScript реализуют наследование следующим способом:

```
function Class2(prop1, prop2, prop3){
  Class1.call(this, prop1, prop2)
  this.prop3 = prop3
}
Class2.prototype = new Class1(null, null);
Class2.prototype.constructor = Class2;
```

Такой подход имеет очевидные недостатки. Во-первых, создается лишний объект-прототип `new Class1(null, null)`, выполняется код инициализации этого объекта. Во-вторых, определенный по умолчанию прототип объекта переписывается новым объектом, в связи с чем уничтожается некоторая функциональность, например свойство `prototype.constructor`, которое нам пришлось явно восстанавливать в исходное значение:

```
Class2.prototype.constructor = Class2;
```

Существует и другой вариант организации наследования в JavaScript, который кажется мне более перспективным и который я буду использовать в своей библиотеке. Это простой перебор свойств объекта-родителя в цикле `for` и переписывание его свойств в объект-потомок:

```
bhv.isa = function(toObject, fromObject) {
  for (var p in fromObject)
  if (typeof toObject[p] == "undefined")
    toObject[p] = fromObject[p];
}

bhv.ISA = function(toObject, fromObject) {
  for (var p in fromObject)
    toObject[p] = fromObject[p];
}
```

Эти две функции имеют название `isa()` и `ISA()` отчасти потому, что в языке программирования Perl используется для похожих целей массив `@ISA`, отчасти потому, что после выполнения функции первый параметр становится *похожим (is a)* на второй параметр. Обе функции делают примерно одинаковую работу, но функция `isa()` делает немного меньше работы и переписывает только отсутствующие в объекте-потомке свойства, что более подходит для организации наследования. С помощью функции `isa()` мы можем переписать программный код, организующий наследование следующим образом:

```
function Class2(prop1, prop2, prop3){
  Class1.call(this, prop1, prop2)
  this.prop3 = prop3
}
bhv.isa(Class2.prototype, Class1.prototype);
```

Как видите, код организации наследования стал более аккуратным, не требует создания лишних объектов, и прототип останется "родным", заданным системой. Но вот вам еще один положительный момент. Таким способом вы можете "смешать" (mix) сразу несколько объектов, т. е. организовать что-то похожее на множественное наследование:

```
function Class2(prop1, prop2, prop3){
  Class0.call(this, prop1)
  Class1.call(this, prop1, prop2)
  this.prop3 = prop3
}
bhv.isa(Class2.prototype, Class0.prototype);
bhv.isa(Class2.prototype, Class1.prototype);
```

В общем, такой код уже достаточно удобен для использования. Но в перспективе мы совместим создание объекта с загрузкой кода этого объекта, а наследование свойств родительских объектов — с загрузкой кода родительских объектов. Для этого наше решение нужно немного доработать. И вот по каким соображениям.

До вызова оператора `new` код определения объекта должен быть уже загружен в интерпретатор JavaScript. Наиболее простой вариант — использовать такое решение:

```
load("Class3");
var object7 = new Class3();
```

Очевидно, что этот код все еще неудобно использовать, во всяком случае, его можно упростить до такого варианта:

```
var object8 = create("Class3");
var object9 = create(Class3);
```

В простейшем случае код такой функции `create()`, загружающий JavaScript-код объектов и создающий эти объекты, может быть следующим:

```
function create(className){
  if (typeof className == "string") {
    load(className);
    var classConstructor = eval(className)
  }else if (typeof className == "function")
    var classConstructor = className;
  var objRef = new classConstructor()
  var args = [];
  for (var i = 1; i < arguments.length; i++)
    args[i-1] = arguments[i];
  objRef.init.apply(objrRef, args)
  return objRef;
}
```

У такого кода существует одна проблема: функции-конструктору `classConstructor()` нельзя передать переменное число параметров, как это можно сделать при помощи функции `apply()` для любой другой функции, например для функции `init()`:

```
objRef.init.apply(objrRef, args)
```

Поэтому код инициализации объекта часто выносят в функцию `init()` или `initialize()`, оставляя конструктор пустым:

```
function Class4(){}
Class4.prototype.init = function(prop1, prop2){
```

```
  this.prop1 = prop1;
  this.prop2 = prop2;
}
var object10 = create(Class4, 2, 2);
```

В теле метода `create()` вызывается конструктор в операторе `new` без параметров, а затем и функция `init()`, которая вызывается методом `apply()` со списком параметров, переданных функции `create()` и смещенных на один параметр, в котором в функцию `create()` передается ссылка на функцию-конструктор или строковое имя этой функции:

```
var objRef = new classConstructor()
var args = [];
for (var i = 1; i < arguments.length; i++)
  args[i-1] = arguments[i];
objRef.init.apply(objrRef, args)
```

Полученная ссылка на объект возвращается из метода `create()` оператором `return` и содержит ссылку на полноценный инициализированный объект:

```
return objRef;
```

Честно признаюсь, что использование "магических" функций, вроде `init()`, мне не совсем по душе. Поэтому я искал способ вызвать функцию-конструктор со списком параметров переменной длины. Сделать это непосредственно в операторе `new` оказалось (по крайней мере, для меня) невозможным, но, погрузившись в глубины спецификации JavaScript (ECMAcript-262), я все же нашел работающее решение:

```
function Class(){}
Class.nativePrototype = Class.prototype;
function create(className) {
  if (typeof className == "string") {
    load(className);
    var classConstructor = eval(className)
  }else if (typeof className == "function")
    var classConstructor = className;
  Class.prototype = classConstructor.prototype;
  var objRef = new Class()
  Class.prototype = Class.nativePrototype;
  var args = [];
  for (var i = 1; i < arguments.length; i++)
    args[i-1] = arguments[i];
```

```
  classConstructor.apply(objRef, args);
  return objRef;
}
```

Что мы сделали? Мы создали пустую функцию-конструктор `Class()`, которую будем использовать для создания новых экземпляров объектов любого типа. Для того чтобы экземпляр создавался не как экземпляр объекта `Class()`, а как экземпляр объекта нужного нам типа, на момент выполнения оператора `new` у этой функции подменяется свойство `prototype`, и затем возвращается в исходное значение. (Впрочем, возвращать в исходное значение это свойство совсем не обязательно, просто я хочу этим подчеркнуть, что после выполнения оператора `new` это свойство не влияет на работу приложения и может принимать совершенно произвольное значение.)

```
Class.prototype = classConctructor.prototype;
var objRef = new Class()
Class.prototype = Class.nativePrototype;
```

Для удобства, этот фрагмент кода я помещаю в функцию `newInstatce()`:

```
function Class(){}
function newInstance(classConstructor) {
  Class.prototype = classConstructor.prototype;
  var objRef = new Class();
  Class.prototype = Class.nativePrototype;
  return objRef;
};
function create(className) {
  if (typeof className == "string") {
    load(className);
    var classConctructor = eval(className)
  } else if (typeof className == "function")
    var classConstructor = className;
  var args = [];
  for (var i = 1; i < arguments.length; i++)
    args[i-1] = arguments[i];
  classConstructor.prototype.derive = derive;
  var objRef;
  classConstructor.apply(
    objRef = bhv.classes.newInstance(classConstructor), args);
}
```

Я выделил функцию `newInstatce()` с прицелом на использование этой функции для выполнения более сложного кода инициализации объекта в следующем разделе, когда мы будем уже создавать окончательную версию нашей библиотеки classe$.js.

Все, что нам осталось рассмотреть, — это использование функции `derive()`, которая будет организовывать смешивание (mix) или наследование объектов:

```
function derive=function(className){
  if (typeof className == "string") {
    load(className);
    var classConctructor = eval(className)
  }else if (typeof className == "function")
    var classConstructor = className;
  for (var i = 1; i < arguments.length; i++)
    args[i-1] = arguments[i];
  bhv.isa(this.constructor.prototype, classConstructor.prototype)
  classConstructor.call(this, args)
  return this;
};
```

Теперь задать наследование объектов (классов) можно очень просто:

```
function Class5(prop1, prop2, prop3){
  this.derive("Class0", prop1);
  this.derive("Class1", prop1, prop2)
  this.prop3 = prop3
}
object11 = create("Class5", 1, 2, 3);
object12 = create(Class5, 1, 2, 3);
```

Итак, мы уже в общих чертах обрисовали систему, которая будет обеспечивать нам загрузку модулей, создание объектов, реализацию наследования и загрузку кода всех объектов-родителей. Осталось только дополнить нашу систему подробностями, в частности, реализовать собственно код загрузчика модулей. Этим мы и займемся в следующем разделе.

Обратите внимание, что методам `create()` и `derive()` можно передать и строку с именем функции-конструктора, и ссылку на функцию-конструктор. Это не ошибка и не опечатка. Я специально предусмотрел такую возможность, чтобы эти функции можно было использовать с обычными объектами, которые нет необходимости загружать из модулей динамически.

11.4. Реализация загрузчика модулей JavaScript

Основные моменты, связанные с использованием пространств имен, созданием классов, наследованием и загрузкой модулей мы уже разобрали в предыдущем разделе, поэтому перейдем к разбору полного текста модуля classe$.js.

Как вы можете заметить, функции `bhv.create()`, `bhv.derive()` и `bhv.load()` ожидают, что в качестве первого параметра им будет передана строка, определяющая пространство имен, которое, в свою очередь, определяется путем к модулю (файлу с определением объекта или модуля) относительно каталога приложения. Для определения пути файловой системы используется следующее выражение:

```
bhv.getApplicationFolder() + strNameSpace.replace(/\./g,"/") + ".js"
```

Мы договорились, что код с определением объектов будет храниться в файлах, имя которых совпадает с именем объекта, а путь будет совпадать с именем пространства имен.

Пока что мы говорили только об объектах. А как же быть, если вам необходимо загрузить модуль и вы не хотите создавать экземпляр объекта? Для этого достаточно просто воспользоваться функцией `bhv.load()`, а не `bhv.create()`:

```
var module1 = load("bhv.test.module1")
```

Модуль должен иметь примерно такое определение:

```
function module1(){/*имя, к которому привязываются имена модуля*/}
module1.function1 = function(){};
module1.function2 = function(){};
```

После загрузки модуля к его членам можно обращаться либо по имени переменной, которой присвоена ссылка на модуль, либо по полному имени модуля с использованием в качестве разделителя знака $ (а для тех, кто решил использовать не библиотеку classe$.js, а библиотеку classes.js — разделенных точкой):

```
var test1 = load("bhv.test.module1");
test1.function1();
test1.function2();
bhv$test$module1.function1();
bhv$test$module1.function2();
```

Для преобразования точек в знаки $ применяется регулярное выражение:

```
strNameSpace.replace(/\./g,"$");
```

Функция `bhv.load()` может использоваться вами непосредственно для загрузки модулей. Для того чтобы имена модулей не были смешаны с именами объектов (классов), имена модулей следует всегда начинать со строчной буквы. Такое соглашение позволяет сделать код читаемым и избежать недоразумений.

Для определения пути от корневого каталога Web-сервера к каталогу Web-приложения используется функция `bhv.getApplicationFolder()` из файла bhv\util.js:

```
bhv.APPLICATION_FOLDER = null;
bhv.getApplicationFolder = function(){
  if (bhv.APPLICATION_FODER)
    return bhv.APPLICATION_FOLDER;
  var scripts = document.getElementsByTagName("SCRIPT");
  var indexOfRoot = -1;
  for (var i = 0; i < scripts.length; i++) {
    indexOfRoot = String(scripts[i].src).replace(/\\/g,'/')
                        .lastIndexOf('bhv/util.js');
    if (indexOfRoot >= 0){
        bhv.APPLICATION_FOLDER =
           new String(scripts[i].src).substring(0, indexOfRoot)
        return bhv.APPLICATION_FOLDER;
    }
  }
}
```

Работа этой функции основана на том, что при загрузке в HTML-элементе `SCRIPT` будет указан или абсолютный, или относительный путь к библиотеке bhv\util.js. В противном случае библиотека не была бы загружена и функция `bhv.getApplicationFolder()` была бы недоступна. Поэтому из пути к этой библиотеки извлекается часть строки до `bhv/util.js`. В нашем тестовом приложении библиотека util.js расположена по пути \bhvajax\bhv\util.js. Следовательно, функция `bhv.getApplicationFolder()` вернет строковое значение `"/bhvajax/"`. Если же путь к библиотеке в атрибуте `src` HTML-элемента `SCRIPT` задан в относительной форме, например:

```
<SCRIPT src="../bhv/util.js"></SCRIPT>
```

функция `bhv.getApplicationFolder()` вернет значение `"../"`, что также позволит нам правильно адресовать наши модули в выражении:

```
bhv.getApplicationFolder() + strNameSpace.replace(/\./g,"/") + ".js"
```

Замечание

Некоторые Web-браузеры при загрузке документа все относительные адреса, заданные в атрибутах `src` или `href`, преобразуют в абсолютные адреса, включая имя протокола, домена и порт. Это, как правило, не вносит дополнительных сложностей, но может стать неожиданным, когда разработчик начинает анализировать фактическое значения атрибутов `src` и `href`, заданные в HTML-документе относительными путями.

Программный код модуля или скрипт с определением объекта (класса) следует загружать всего один раз. После первой загрузки этот код уже находится в интерпретаторе JavaScript и может быть использован без повторной загрузки. Поэтому код функции `bhv.load()` в самом начале проверяет, не был ли модуль загружен до вызова функции `bhv.load()`:

```
if (bhv.classes.loadedClasses[strNameSpace])
  return bhv.classes.loadedClasses[strNameSpace];
```

Массив `bhv.classes.loadedClasses` заполняется при вызове функции `bhv.loader()`:

```
bhv.classes.loadedClasses[strNameSpace] =
          eval(strNameSpace.replace(/\./g,"$"));
```

Если модуль не был загружен, то функция `bhv.load()` загружает его функцией `bhv.sendRequesr()` в синхронном (не асинхронном) режиме. В качестве обработчика события `onreadystatechange` задается функция `bhv.classes.loader()`. Эта функция носит вспомогательный характер и поэтому объявлена в пространстве имен `bhv.clases`, в отличие от функции `bhv.load()`, которая будет вызываться как из кода функции `bhv.create()`, так и самостоятельно для загрузки модулей, которые не содержат определение объектов.

Код функции `bhv.create()` был практически полностью рассмотрен в предыдущем разделе. Более сложным, в сравнении с вариантом из предыдущего раздела, стал код функции `bhv.classes.newInstance()`:

```
bhv.classes.newInstance = function(classConstructor, className) {
  bhv.classes.Class.prototype = classConstructor.prototype;
  var objRef = new bhv.classes.Class();
  objRef.parentList = {};
  objRef.superClass = {};
  objRef.parentList[""+className] = true;
  bhv.classes.Class.prototype = bhv.classes.Class.nativePrototype;
  return objRef;
};
```

В дополнение к коду функции `newInstance()`, которую мы разобрали в предыдущем разделе, для каждого объекта инициализируются объекты, которые используются для организации наследования (смешивания), объект `objRef.parentList`:

```
objRef.parentList = {};
objRef.parentList[""+className] = true;
```

используется для исключения зацикливания при наследовании от классов, содержащих циклические зависимости, когда объект **А** наследует объект **Б**, и, одновременно, объект **Б** наследует объект **А**, непосредственно или косвенно. Для защиты от зацикливания в процессе наследования в функции `bhv.derive()` производится следующий анализ цепочки наследования:

```
bhv.classes.derive=function(className){
  if (this.parentList[""+className])
    return this;
  this.parentList[""+className] = true;
```

При повторном появлении объекта в списке наследования код функции `bhv.classes.derive()` завершается на первом операторе. Возврат ссылки на текущий объект (`return this`) позволяет выстраивать вызов функции `derive()` в цепочку:

```
function Class6(){
  this.derive(Class0).derive(Class1).derive(Class2);
}
```

вместо вызовов:

```
function Class6(){
  this.derive(Class0);
  this..derive(Class1);
  this..derive(Class2);
}
```

Код библиотеки, которая обеспечивает загрузку модулей и скриптов в процессе создания объектов функцией `bhv.create()`, создан. Я не буду приводить развернутый пример использования кода библиотеки в этой главе. Потому, что в следующей *главе 12* мы используем библиотеку classe$.js для создания Ajax-компонента "Плавающее окно". Приведу, в качестве анонса материала следующей главы, только фрагмент программы, которая создает компоненты "Плавающее окно" при помощи функции `bhv.create()`:

```
var window1 = bhv.create("bhv.widget.BaseWindow",
  "Пустое окно служит заготовкой для потомков объекта BaseWindow")
var window2 = bhv.create("bhv.widget.WindowURL", "Web-ресурс google.com")
```

```
window1.show();
window2.show();
window2.loadIFrame("http://google.com:80");
```

Довольно неплохой результат нашей работы! Особенно, если учесть, что код объекта `BaseWindow` ничем не отличается от кода обычного объекта JavaScript, как он определен в спецификации языка. А код объекта `WindowURL` для наследования свойств и функций-методов объекта `BaseWindow`, а также для загрузки этого объекта, должен всего лишь содержать в конструкторе вызов функции-метода `derive()`, а в остальном остается самым обычным объектом JavaScript:

```
function WindowURL(title){
  this.derive("bhv.widget.BaseWindow", title);
  this.loadURL=loadURL
  this.loadTextURL=loadTextURL
  this.loadIFrame=loadIFrame
  this.loadDiv=loadDiv
};
```

Мой опыт использования динамической загрузки модулей показывает, что приложения, применяющие динамическую загрузку модулей, работают достаточно надежно. Небольшая задержка в работе Web-приложения происходит при первой загрузке модуля. Взамен разработчик получает средства, позволяющие легко отлаживать Web-приложение и использовать все преимущества модульной разработки. Пользователь Web-приложения также остается не в накладе, поскольку разработчик имеет возможность разрабатывать более надежные и удобные для пользователя элементы интерфейса.

Все рассмотренные до этого компоненты Ajax-приложений я в своей повседневной практике использую совместно с динамической загрузкой функциями, аналогичными `bhv.load()` и `bhv.create()`. Но я не мог позволить себе навязывать уважаемому читателю свои методы разработки. И поэтому только в предпоследней главе книги рассказал о методах динамической загрузки модулей и лишь в последней главе буду использовать динамическую загрузку при разработке Ajax-компонентов.

Если вы решите динамически загружать свои собственные модули, то должен вас предупредить об одной особенности работы встроенной функции JavaScript `eval()`. Функция `eval()` выполняет программный код в том контексте, в котором вызывается эта функция. В нашем случае этот вызов происходит в контексте функции `bhv.classes.loader()`:

```
bhv.classes.loader = function (strNameSpace){
  var mod = bhv.classes["package"](strNameSpace);
  eval(this.responseText);
```

При вызове `eval()` в теле произвольной функции все глобальные переменные, определенные с ключевым словом `var`, в загружаемом модуле станут локальными переменными для функции `bhv.classes.loader()` и, следовательно, будут недоступными после выхода из функции. Существует несколько способов определить переменные в модуле так, чтобы они были видны на глобальном уровне. Вы можете выбрать понравившийся вам способ.

Во-первых, можно использовать определение переменных без ключевого слова `var`. Такие переменные приписываются в соответствии со спецификацией JavaScript к глобальному контексту, если не встретятся ни в одном из локальных контекстов. Например:

```
// var globalVar = true;
globalVar = true;
```

Во-вторых, можно для Web-браузера задавать глобальные переменные как свойства объекта `window`, например:

```
window.globalVar = true;
```

В-третьих, можно задавать глобальные значения в качестве свойств глобальных объектов пространства имен, которые определены на глобальном уровне. В наших приложениях это объект `bhv`:

```
bhv.globalVar = true;
```

Наконец, вместо функции JavaScript `eval()` можно воспользоваться функцией `window.eval()` или `execScript()` — в зависимости от типа Web-браузера. Такое решение является не кроссбраузерным, но только оно позволяет выполнить код программы JavaScript гарантированно в глобальном контексте, независимо от точки вызова этой функции.

Надо заметить, что все эти варианты являются недостаточно красивыми с точки зрения культуры программирования. Наиболее культурным следует признать третий вариант с использованием глобального объекта пространства имен:

```
bhv.globalVar = true;
```

Однако и этот вариант не без недостатков, и мы их рассмотрели в тексте главы. Это два недостатка — сложности с созданием объекта пространства имен и дополнительные затраты на обращение к значениям как к свойствам объекта пространства имен.

Первый вариант, с использованием переменных без ключевого слова `var`:

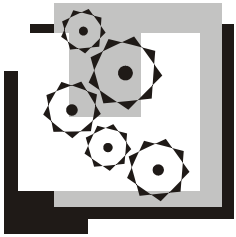
```
// var globalVar = true;
globalVar = true
```

многими признается как нестрогий и даже ошибочный. Тем не менее, этот вариант предусмотрен спецификацией JavaScript (ECMAScript-262) и используется некоторыми библиотеками.

В библиотеке classe$.js используется именно такой способ создания переменных. При этом уникальность значений переменных обеспечивается способом их именования по имени пакета и имени модуля (объекта), разделенных знаком $.

Библиотека classes.js решает проблему с определением глобальных значений при помощи объекта пространства имен `bhv`. Впрочем, если вы решите создать пространство имен, отличное от `bhv`, например `ru.bhv`, библиотека classes.js воспользуется приемом неявного создания переменной в глобальном контексте без ключевого слова `var`.

На этом я завершаю главу о загрузчиках модулей JavaScript, чтобы в следующей, заключительной главе, воспользоваться этим загрузчиком для создания Ajax-компонента "Плавающее окно". Если такие способы загрузки модулей идут слишком уж вразрез с вашими принципами и понятиями о допустимых приемах программирования, надеюсь, что материал, который мы рассмотрели до настоящего времени, окупит те отрицательные эмоции, которые, возможно, вы испытали при прочтении настоящей главы.



Глава 12

Разработка компонента "Плавающее окно"

Вот мы и добрались, уважаемые читатели, до последней главы нашей книги. В этой главе мы разработаем компонент "Плавающее окно". Плавающие окна, реализованные при помощи HTML-элемента `DIV`, в настоящее время используются на Web-сайтах довольно часто. Как правило, такие окна выглядят более привлекательно, чем окна типа `window.alert()`. Следует учитывать, что пользователи настольных (desktop) приложений привыкли к некоторым стандартным возможностям, которые ожидаются от реализации плавающего окна в приложении. В частности, плавающее окно должно перемещаться по экрану методом *drag-and-drop*, должно *распахиваться* на все окно и *сворачиваться*, выводиться на первый план при выборе окна кнопкой мыши, иметь средства для изменения размеров по желанию пользователя. Если окно не реализует таких возможностей, скорее всего, оно будет только раздражать пользователя приложения. Впрочем, может быть это только мое видение вопроса. Меня, например, в детстве раздражали пилястры, которые только при виде издали напоминали колоны, и кармашки-обмашки, в которые ничего нельзя было положить.

При реализации компонента "Плавающее окно" мы используем возможности динамической загрузки модулей и наследования (смешивания) объектов, рассмотренные в *главе 11*. Как вы помните, в *главе 11* мы специально не рассматривали примеров использования библиотек classe$.js и classes.js, чтобы поработать с этими библиотеками не на игрушечных примерах, а посмотреть на их использование при разработке реальных компонентов. Использовать смешивание (наследование) для реализации плавающих окон достаточно удобно. В этом случае, базовый объект плавающего окна должен обеспечивать нам просто прорисовку окна на экране и базовые операции с окном: drag-and-drop, распахивание, сворачивание и т. п. На базе этого объекта можно создавать более специализированные объекты плавающих окон, которые, например, смогут загружать и отображать документы методами Ajax.

Текст примеров главы расположен на прилагаемом к книге компакт-диске в папке \bhvajax\draganddrop. Текст модулей с определением объектов "Плавающее окно" расположен в папке \bhvajax\bhv\widget, что соответствует предложенному в *главе 11* отображению пространств имен объектов и модулей на папки и документы файловой системы.

12.1. Реализация технологии drag-and-drop средствами JavaScript

Реализовать технологию drag-and-drop (перетаскивание) для HTML-документа — задача не из простых. При разработке мы столкнемся с такими сложностями.

- В спецификации HTML не определены события, которые бы обслуживали конкретно технологию drag-and-drop (такие события определены только для некоторых Web-браузеров). Для перетаскивания используются стандартные события мыши. Работа с любыми событиями является наиболее некроссбраузерной частью Web-приложений. Это связано с тем, что в спецификации HTML определены только имена событий, но не указано, как эти события могут назначаться при помощи JavaScript (не в атрибуте HTML-элемента), и какая дополнительная информация, например координаты указателя мыши, должны передаваться в функцию-обработчик события.
- Следующей сложностью является то, что стандартное действие пользователя при перетаскивании, а именно нажатие на копку мыши и перемещение указателя мыши в новую точку без отпускания кнопки, приводит в Web-браузере к выделению текста, как это было бы в большинстве текстовых редакторов. Это связано с тем, что Web-браузер изначально разрабатывался для отображения текста, а не для работы в качестве "богатого" клиента Web-приложений. Запрет на выделение текста в разных версиях Web-браузеров задается по-разному. Но иногда выделения текста все же необходимо, поэтому полностью запрещать эту операцию нельзя. Например, мы часто выделяем текст в HTML-элементе `INPUT`, чтобы новые данные вводились поверх имеющихся данных, как это было реализовано в компоненте Lookup Combobox.

Резюме из сказанного может быть только такое: лучше использовать готовую реализацию технологии drag-and-drop из библиотеки JavaScript стороннего разработчика, чем пытаться это сделать самому. Вопреки этому здравому суждению, мы попытаемся реализовать перетаскивание drag-and-drop самостоятельно, исходя совершенно из других соображений. Во-первых, для того чтобы выбрать из готовых вариантов наиболее подходящий для вашего

приложения, необходимо хорошо представлять, как может быть реализовано перетаскивание и какие проблемы при этом приходится решать. В этом случае, изучив код библиотеки, вы сразу поймете, глубоко ли эта библиотека прорабатывает те или иные моменты в технологии drag-and-drop, или реализация является поверхностной, некроссбраузерной и ненадежной. Вторая причина более прозаична. Все мои примеры предоставлены для вашего удобства на прилагаемом к книге компакт-диске, и они должны работать без необходимости для читателя загружать какие-то дополнительные библиотеки. Кроме того, код этих библиотек сторонних разработчиков может иногда изменяться без поддержки обратной совместимости, что могло бы сделать примеры неработающими.

Рассмотрим вопрос с назначением обработчиков событий HTML-элементам. До сих пор мы назначали события в атрибутах HTML-элемента (в разметке HTML):

```
<div id="div0" onmouseover="this.className='selectedDiv';"> </div>
```

или присваивали свойству HTML-элемента ссылку на функцию-обработчик:

```
var div0 = document.getElementById("div0");
div0.onclick = function(){this.className="buttonPressed";};
```

Существует более гибкий способ, который позволяет назначить одному событию произвольное количество обработчиков. К сожалению, этот способ не кроссбраузерный, поэтому сразу рассмотрим функцию, которая назначает события кроссбраузерно:

```
bhv.addEventListener = function(element, strEvent, callback){
  if (element.addEventListener)
    element.addEventListener(strEvent, callback, false);
  else
    element.attachEvent("on" + strEvent, callback);
}
```

Как вы можете убедиться, для различных версий Web-браузеров способ назначения событий практически одинаковый, но имена событий передаются в первом случае без приставки `on`. Впрочем, более существенная разница возникает не при назначении функции-обработчика, а при вызове самой функции-обработчика различными версиями Web-браузера. Но это отдельная тема для разговора.

В настоящее время назначение обработчиков событий при помощи функций `addEventListener()` и `attachEvent()` все еще не вошло в широкую практику. Более того, с чьей-то легкой руки распространилось мнение, что такой способ из-за некроссбраузерности использовать не следует. Поэтому широко

рекламируются самодельные способы обработки событий, которые реализованы средствами JavaScript по модели "Подписка — Рассылка". Я могу поспорить с таким мнением. Как мы только что убедились, кроссбраузерность назначения функций обеспечивается довольно просто. А некроссбраузерность событийной модели Web-браузера в целом нужно учитывать при любом из вариантов назначения событий. Кроме того, отказ от использования этой сравнительно новой и далеко идущей возможности Web-браузера сковывает движение вперед.

Единственный, действительно существенный, недостаток у рассматриваемого способа назначения функций-обработчиков событий — это невозможность получить список всех назначенных событию функций. Впрочем, никто не запрещает самостоятельно реализовать такую возможность:

```
bhv.addEventListener = function(element, strEvent, callback){
  if (!element.bhv_listeners)
    element.bhv_listeners = {};
  if (!element.bhv_listeners[strEvent])
    element.bhv_listeners[strEvent] = [];
  element.bhv_listeners[strEvent].push(callback);

  if (element.addEventListener)
    element.addEventListener(strEvent, callback, false);
  else
    element.attachEvent("on" + strEvent, callback);
}
```

В файле dnd.js приведена простейшая реализация технологии drag-and-drop с учетом кроссбраузерной работы.

Мы определяем объекты библиотеки dnd.js в пространстве имен `bhv.dnd`. Затем определяется глобальный объект-монитор, который будет содержать ссылку на текущий перетаскиваемый объект:

```
bhv.dnd.draggedElement = null;
```

Значение `null` этого объекта свидетельствует о том, что в настоящее время никакой объект не перетаскивается.

Заканчивается перетаскивание объекта по событию `onmouseup`:

```
bhv.addEventListener(document, "mouseup",
  function(event){
    event = event || window.event
    if ( bhv.dnd.draggedElement)
```

```
      bhv.dnd.draggedElement.endDrag(event);
  }
)
```

Присоединение этого события к объекту `document` гарантирует, что событие `onmouseup` будет обработано в любой точке документа. При наступлении события проверяется, есть ли текущий перетаскиваемый объект, и если такой объект существует, вызывается его метод `endDrag()`:

```
draggedElement.endDrag(event);
```

В настоящее время нас не должно интересовать, как именно реализован этот метод. Просто мы предполагаем, что наш перетаскиваемый объект должен реализовывать некоторый интерфейс: `startDrag()`, `endDrag()` и `destroy()`.

Прежде чем разбирать реализацию объекта `DnD` (drag-and-drop), разберем еще три события, которые позволят нам отменять выделение текста, которое может помешать перетаскиванию:

```
bhv.addEventListener(document, "selectstart",
  function(event){
    if (bhv.dnd.draggedElement)
      return false;
  }
)

bhv.addEventListener(document, "select",
  function(event){
    if (bhv.dnd.draggedElement)
      return false;
  }
)

document.onmousedown = function() {
  if ( bhv.dnd.draggedElement)
    return false;
}
```

Обработчики событий присоединяются к объекту `document`, поэтому действуют для всех элементов, если только явно не отменены. Событие `onselectstart` присутствует не во всех типах Web-браузеров. Если на момент вызова обработчика события пользователь перетаскивает объект:

```
if (bhv.dnd.draggedElement)
```

то функция-обработчик события возвращает логическое значение `false`, что предотвращает стандартное действие Web-браузера для события, и текст не будет выделен.

Событие `onselect` присутствует во всех Web-браузерах. Однако это событие генерируется только при выделении текста в элементах `INPUT`. Обычный текст все же будет выделяться. Поэтому мы дополнительно задаем обработчик `onmousedown`. Вы обратили внимание, что обработчик этого события задан как свойство объекта `document`, а не присоединен функцией `bhv.addEventListener()`? Дело в том, что только такой способ отменяет выделение текста во всех моделях Web-браузеров.

Как видите, для поддержки кроссбраузерности довольно часто приходится идти на применение трюков. Это еще раз показывает, что использовать библиотеки, которые поддерживают разнообразную функциональность с учетом кроссбраузерности, необходимо в любом серьезном приложении, потому что вы можете использовать функциональность этих библиотек, не задумываясь о том, что следующая версия Web-браузера, которая выйдет завтра или через неделю, перечеркнет всю вашу работу по поддержке кроссбраузерности. Вам достаточно будет только взять с сайта очередную версию библиотеки. Или, если таковой не окажется, самостоятельно внести изменения в библиотеку, если это допускается лицензией библиотеки. Программный же код остальной части приложения останется без изменений.

До этого момента мы еще не рассматривали собственно объект `DnD` (drag-and-drop), а только назначали соответствующие обработчики событий к объекту `document`. Вот вам еще одна дополнительная выгода от применения методов `addEventListenet()` или `attachEvent()`, использование которых иногда называют *второй моделью обработки событий*. Вы можете присоединять собственные события к объекту, не беспокоясь, что к этому объекту разработчик прикладного программного обеспечения присоединит еще и свои функции-обработчики. Все функции будут вызваны последовательно и, в том случае, если они написаны достаточно культурно, не будут мешать работе друг друга. Единственное исключение, которые мы себе позволили — присоединили обработчик события `onmousedown` непосредственно к объекту `document`:

```
document.onmousedown = function() {
  if ( bhv.dnd.draggedElement)
    return false;
}
```

Увы, разработчик прикладного программного обеспечения также может присоединить обработчик события к этому свойству, заменив нашу функцию своей функцией. В данном конкретном случае это не приведет к критическим

сбоям в работе приложения. Потому что наша функция просто отключает выделение текста мышью при перетаскивании объекта.

Прейдем к рассмотрению объекта `DnD` (drag-and-drop). Функция-конструктор этого объекта принимает три параметра. В первом параметре передается собственно перетаскиваемый HTML-элемент. Во втором — HTML-элемент, нажатие на который инициирует перетаскивание. Например, в плавающем окне вы, скорее всего, захотите по нажатию кнопки мыши осуществлять действия, отличные от перетаскивания: нажатие кнопок, выбор пунктов выпадающего меню и т. п. Поэтому имеет смысл инициировать перетаскивание только по нажатию на заголовке окна. Разумеется, никто не запрещает назначить в качестве первого и второго параметров один и тот же HTML-элемент. В третьем параметре функции-конструктору передается HTML-элемент, нажатие на который инициирует процесс изменения размера окна.

```
function DnD(element, handle, resize){
  if (! handle)
    handle = element;
  this.element = element;
  this.handle = handle;
  this.resize = resize;
  this.element.dndmodel = this;
  handle.dndmodel = this;
  bhv.addEventListener(handle, "mousedown", this.startDrag)
  if (resize){
    resize.dndmodel = this;
    bhv.addEventListener(resize, "mousedown", this.startDrag)
  }
}
```

Изменение размера HTML-элемента и его перетаскивание (drag-and-drop) — с точки зрения программирования похожие операции. При перетаскивании вы должны определить координаты точки нажатия кнопки и перемещение указателя мыши относительно этой координаты и переместить на вычисленную величину весь HTML-элемент. А при изменении размера на ту же самую величину будет изменяться высота и ширина HTML-элемента.

Наиболее сложна реализации методов `startDrag()` и `endDrag()`. При первом вызове в процессе работы приложения метода `statDrag()` создается рамка красного цвета, которая будет визуализировать текущую позицию перетаскивания:

```
if (! bhv.dnd.border){
  bhv.dnd.border = document.createElement("DIV");
```

```
  bhv.dnd.border.style.borderStyle = "solid";
  bhv.dnd.border.style.borderWidth = "4px";
  bhv.dnd.border.style.borderColor = "red";
  bhv.dnd.border.style.position = "absolute";
  bhv.dnd.border.style.zIndex = "1000";
  document.getElementsByTagName("body")[0].appendChild(bhv.dnd.border);
}
```

Для чего необходима эта рамка? Дело в том, что HTML-элемент может содержать достаточно объемный текст и изображения, возможно, со скроллингом. Теоретически, можно было бы перетаскивать вслед за указателем мыши весь HTML-элемент. Но в зависимости от модели Web-браузера, объема текста, содержащегося в перетаскиваемом HTML-элементе, и технических характеристик компьютера этот процесс может происходить достаточно медленно. В худшем случае, перетаскиваемый HTML-элемент будет, пульсируя, с запаздыванием в добрый десяток секунд, с неэстетичным эффектом "размазывания" скользить по экрану. Поэтому часто перетягивается не сам элемент, а его рамка. Рамка выбрана заметного красного цвета. Разумеется, вы можете изменить ее внешний вид на свой вкус.

Второй вопрос, который может у вас возникнуть: почему эта рамка создается при первом вызове функции `startDrag()`? Дело в том, что в зависимости от места расположения тега `SCRIPT`, который может размещаться в элементах `HEAD` или `BODY` HTML-документа, создание новых HTML-элементов и их присоединение к HTML-элементу `BODY` может быть недоступным. Вероятность того, что разработчик прикладного программного обеспечения присоединит скрипт dnd.js в `HEAD` HTML-документа, очень высока, потому что часто звучит не совсем корректное мнение, что скрипты всегда следует загружать исключительно в элементе `HEAD` HTML-документа. Это не всегда так.

Часто, и вполне обоснованно, инициализацию объектов и создание новых HTML-элементов выносят в обработчик события `onload`. Этот вариант является справедливым, потому что к моменту вызова события `onload` все ресурсы, заданные в заголовке `HEAD` HTML-документа, загружены с Web-сервера в Web-браузер, и тело документа (элемент `BODY`) также загружено в Web-браузер. Разработчик прикладного программного обеспечения, скорее всего, захочет самостоятельно использовать обработчик события `onload`. Поэтому инициализация объекта `bhv.dnd.border` отложена до первого вызова функции `startDrag()`. Как следует из определения функции, этот элемент будет создан одни раз и использоваться для всех объектов `DnD` (drag-and-drop). Перед отображением объект `bhv.dnd.border` будет привязываться к координатам перетаскиваемого элемента:

```
bhv.dnd.border.style.width = element.offsetWidth + "px";
```

```
bhv.dnd.border.style.height = element.offsetHeight + "px";
bhv.dnd.border.style.left = bhv.left(element) + "px";
bhv.dnd.border.style.top = bhv.top(element) + "px";
bhv.dnd.border.bhv_offsetLeft = event.clientX - bhv.left(element);
bhv.dnd.border.bhv_offsetTop = event.clientY - bhv.top(element);
bhv.dnd.border.style.display = "block";
```

Обратите внимание на использование свойств `bhv_offsetLeft` и `bhv_offsetTop`. В них мы запоминаем текущие координаты нажатия мыши относительно левого верхнего угла перетаскиваемого HTML-элемента. Здесь используется соглашение об именовании свойств, которые начинаются префиксом `bhv`. Если не использовать префикс для обеспечения уникальности имени, с большой долей вероятности разработчик прикладного программного обеспечения, который используют библиотеку, может переопределить заданные в библиотеке свойства своими значениями. Но вероятность того, что он использует префикс `bhv`, особенно если в документации библиотеки специально обратить на это внимание, ничтожно мала.

За перемещением или изменением размера HTML-элемента (а фактически — рамки, которая представляет HTML-элемент) следит функция-обработчик события `onmousemove` объекта `document`:

```
bhv.addEventListener(document, "mousemove",
  function(event){
    event = event || window.event
    try{
      if (bhv.dnd.border.style.display == "block") {
        if (! bhv.dnd.isResize){
          bhv.dnd.border.style.top = event.clientY
              - bhv.dnd.border.bhv_offsetTop + "px";
          bhv.dnd.border.style.left = event.clientX
              - bhv.dnd.border.bhv_offsetLeft + "px";
        }else {
          bhv.dnd.border.style.height = event.clientY
              - bhv.dnd.border.offsetTop + "px";
          bhv.dnd.border.style.width = event.clientX
              - bhv.dnd.border.offsetLeft + "px";
        }
      }
    }catch(ex){}
  }
)
```

Это событие обрабатывается на уровне всего документа, а не перетаскиваемого элемента или рамки, которая представляет элемент, потому что указатель мыши может легко обогнать перетаскиваемый элемент. И на этом процесс перетаскивания может остановиться и стать неуправляемым. Обрабатывая событие для всего документа, мы гарантируем, что перемещение указателя мыши будет в большинстве случаев обработано правильно. Да, я не оговорился, именно в большинстве случаев, но не всегда. Дело в том, что код промышленно работающих библиотек, реализующих технологию drag-and-drop, гораздо сложнее и учитывает на порядок больше тонкостей, связанных с кроссбраузерной работой и непредсказуемостью действий пользователя. Кроме того, контекст Web-приложения, в котором вызывается операция перетаскивания, также может быть самым разнообразным и влиять на корректность работы процесса перетаскивания drag-and-drop. Поэтому еще раз агитирую вас использовать профессиональные библиотеки, реализующие, в частности, и кроссбраузерное перетаскивание объектов с очень широким диапазоном возможностей (таких как плавная визуализация перетаскивания, определение объектов, в которые разрешено или запрещено перетаскивать объекты, встраивание перетаскиваемых объектов не только визуально, но и логически, в структуру документа, например, в качестве пунктов списка, строк таблицы и т. п.).

При отпускании кнопки мыши вызывается функция `endDrag()`:

```
bhv.addEventListener(document, "mouseup",
  function(event){
    event = event || window.event
    if (bhv.dnd.draggedElement)
      bhv.dnd.draggedElement.endDrag(event || window.event)
  }
)
```

Эта функция также привязана к объекту `document` и вызывается только в тех случаях, когда глобальный объект `bhv.dnd.draggedElement` определен и имеет значение, отличное от `null`.

Функция `endDrag()` определяет, какая из операций — перетаскивание или изменение размера — осуществляется пользователем, и, в зависимости от этого, выполняет свою часть кода. Например, если пользователь осуществляет изменение размера HTML-элемента, новый размер устанавливается равным текущему размеру красной рамки, которая отвечает за визуализацию процесса изменения размера:

```
bhv.dnd.draggedElement = null;
if (bhv.dnd.isResize){
```

```
  this.element.style.height = bhv.dnd.border.offsetHeight + "px";
  this.element.style.width = bhv.dnd.border.offsetWidth + "px";
  try{
    bhv.dnd.border.style.display= "none";
    this.element.model.validate();
  }catch(ex){};
return;
}
```

Вы можете поставить мне в упрек, что я реализовал и перетаскивание, и изменение размера HTML-элемента одними и теми же функциями. Я принимаю это ваше замечание, но в свое оправдание хочу сказать, что в большей степени углубился в обеспечение кроссбраузерной работы и элементарной надежности не зависимо от действий пользователя. И начинал я с чистого перетаскивания HTML-элемента, а затем при минимальных изменениях в программном коде добавил функциональность изменения размера HTML-элемента. При этом заранее я не знал, какие методы будут применены и какие сложности встретятся на пути. Поэтому получился именно такой программный код, каким вы его видите.

12.2. Реализация базового объекта "Плавающее окно"

В *главе 11* мы рассмотрели приемы, при помощи которых можно создавать собственные модули, в частности, модули, которые сдержат определение объектов. При этом создание объектов методом `bhv.create()` обеспечивает загрузку модуля, содержащего программный код объектов, а также всех зависимых модулей. Такой подход позволяет реализовать смешивание объектов, которое по своим возможностям аналогично множественному наследованию классов в языках программирования, работающих с классами.

Базовый объект "Плавающее окно" будет обладать только базовой функциональностью, такой как отображение окна методом `show()`, сокрытие методом `hide()`, сворачивание, разворачивание, распахивание (разворачивание во весь экран), перетаскивание. Плавающее окно создается при помощи HTML-элементов `DIV`, а некоторые его элементы при помощи HTML-элементов `SPAN`. Код создания окна вы можете изучить по файлу \bhvajax\bhv\widget\BaseWindow.js.

Программный код создания окна по своей логике достаточно прост. Все HTML-элементы `DIV` и `SPAN` создаются методом `document.createElement()`. Внешний вид HTML-элементов определен стилевыми классами:

```
var div = this.div = document.createElement("div");
div.model = this;
```

```
div.toplevel = 51;
div.className = "windowPanel";
div.style.position = "absolute";
```

Как всегда, можно изменять стилевое оформление плавающего окна только за счет изменения стилей в css-файлах, что не потребует изменения программного кода JavaScript.

Разъясню некоторые моменты в программном коде из файла \bhvajax\bhv\widget\BaseWindow.js. Функции сворачивания, разворачивания и распахивания окна вызываются при помощи функции `window.setTimeout()` с задержкой в одну секунду (1000 мс):

```
this.toggleExpand = function() {
  setTimeout(function() { ...
```

Это не является необходимым с точки зрения программного решения. Но, как показывают исследования психологов, лучше, если элементы пользовательского интерфейса реагируют на действия пользователя с небольшим запаздыванием. Вот поэтому я и применил функцию `window.setTimeout()`, которая задает задержку визуализации действий пользователя на небольшой интервал времени.

Перетаскивание и изменение размера плавающего окна обеспечивается вызовом оператора создания объекта `DnD`:

```
new DnD(div, head, footerX);
```

Первый параметр содержит ссылку на охватывающий HTML-элемент `DIV`, который собственно и реализует окно, его самый внешний контур. Второй параметр, `head`, который также реализован HTML-элементом `DIV`, представляет собой заголовок окна. При нажатии кнопки мыши на заголовке окна будет инициирован процесс перетаскивания. И, наконец, третий параметр — это маленький прямоугольник в правом нижнем углу окна, который благодаря стилевому классу `windowFooterInsetBox` выглядит вдавленным в нижнюю полоску (footer) окна, а благодаря текстовому содержанию, выглядит почти как соответствующий элемент в настольных (desktop) приложениях, что является предметом моей особой гордости:

```
var footerX = this.footerX = document.createElement("span");
footerX.appendChild(document.createTextNode(".:i"));
footerX.className = "windowFooterInsetBox";
footerX.model = this;
footerX.onmousedown = function(){this.model.focuseWindow();};
```

Я поместил программный код с определением объекта, реализующего компонент Плавающее окно, в файле bhv\Widget\BaseWindow.js. С учетом того,

что все наше приложение размещено в каталоге \bhvajax, полный путь к файлу будет выглядеть как \bhvajax\bhv\widget\BaseWindow.js. Согласно нашим соглашениям по использованию пространств имен, полное имя объекта `BaseWindow` будет `bhv.widget.BaseWindow`. Для того чтобы загрузить код объекта `bhv.widget.BaseWindow` и, одновременно, в одном операторе создать его, нет необходимости подключать явно файл, содержащий определение объекта при помощи HTML-элемента `SCRIPT`. Достаточно будет подключить наши системные библиотеки util.js и classe$.js (или classes.js). С учетом этого код HTML-документа должен выглядеть так:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <title>Ajax: Плавающее окно</title>
    <link rel="stylesheet" type="text/css" href="window.css">
    <script type="text/javascript" src="../bhv/util.js"></script>
    <script type="text/javascript" src="../bhv/classe$.js"></script>
    <script type="text/javascript" src="dnd.js"></script>
  </head>
  <body>
    <script type="text/javascript">
      var window1 = bhv.create("bhv.widget.BaseWindow", "Title");
      window1.show();
    </script>
  </body>
</html>
```

Файл window.css с определением стилей необходимо подключить явно. На самом деле, точно так, как мы загрузили программный код модуля с определением объекта, можно обеспечить и загрузку css-файлов с определением стилей. Код загрузки будет различаться для разных типов Web-браузеров, и мы его в книге не рассматриваем. Также, кроме библиотек util.js и classe$.js (или classes.js), мы явно загружаем библиотеку dnd.js:

```
<script type="text/javascript" src="dnd.js"></script>
```

Эту библиотек также можно загружать при создании объекта `DnD` посредством метода `bhv.create()`. В данном случае я не использую такую возможность только из тех соображений, чтобы сконцентрировать ваше внимание на загрузке модуля bhv\Widget\BaseWindow.js:

```
var window1 = bhv.create("bhv.widget.BaseWindow", "Title");
```

Рассмотренный в этом разделе объект `BaseWindow` недаром получил название базового, потому что реализует только базовые возможности "Плавающего окна". В следующем разделе мы наполним функциональность "Плавающего окна" новыми полезными возможностями.

12.3. Расширение базового компонента "Плавающее окно" новыми возможностями

Объект `BaseWindow` обеспечивает базовую функциональность компонента "Плавающее окно". Можно было бы продолжить наполнять новой функциональностью объект `BaseWindow`, но это быстро приведет к следующим отрицательным явлениям. Во-первых, программный код объекта `BaseWindow` уже достаточно объемный, и его дополнение ухудшит обозримость программного кода. Во-вторых, программный код объекта `BaseWindow` уже отлажен, и внесение дополнительных изменений повлечет за собой необходимость отладки кода заново. При отсутствии развернутых визуальных сред разработки, поиск и исправление ошибок в программах JavaScript с увеличением размера скрипта может превратиться в достаточно сложную задачу. И, в-третьих, возможности базового компонента "Плавающее окно" (объекта `BaseWindow`) могут быть использованы для создания самых разнообразных окон: для отображения сообщений, диалогов, документов, визуальных компонентов приложений. Все говорит о том, что объект `BaseWindow` лучше оставить в том виде, как он есть, и воспользоваться возможностями нашей библиотеки classe$.js (или clsses.js), которые позволяют смешивать (mix) объекты, реализуя функциональность, похожую на наследование. В принципе, для себя я называю это наследованием, но все же, в угоду строгим ревнителям чистоты JavaScript, я использую более строгий термин. К свойствам и функциям-методам нового объекта (объекта-потомка) присоединяются, или как говорят образно "подмешиваются", свойства другого объекта, который можно назвать базовым объектом или объектом-родителем. Но все же правы те, кто не называет это чистым наследованием, поскольку "подмешиваемый" объект может играть в некоторых случаях вспомогательную роль (а не базовую). Впрочем, все зависит от того, как воспринимать объектно-ориентированный подход — как свойство языка программирования или как видение разработчиком решаемой задачи. Оба подхода имеют право на жизнь.

Мы будем расширять функциональность объекта `BaseWindow` только в одном направлении — для загрузки в него HTML-документов. Вы можете аналогично реализовать любую другую функциональность, которая только вам потребуется. В листинге 12.1 приведен программный код объекта `WindowURL`, который расширяет возможности объекта `BaseWindow`.

Листинг 12.1. Объект `WindowURL` расширяет функциональность объекта `BaseWindow`

```
function WindowURL(title){
  this.derive("bhv.widget.BaseWindow",title);
  this.loadURL=loadURL;
  this.loadTextURL=loadTextURL;
  this.loadIFrame=loadIFrame;
  this.loadDiv=loadDiv;
};

function loadURL(url, param){
  var myself=this;
  bhv.sendRequest("GET", url,param, true,
       handleLoadingURL, null, [myself,url]);
}

function handleLoadingURL(myself, url){
  var strResponse = bhv.relocateSRC(this.responseText, url);
  myself.content.innerHTML = strResponse;
}

function loadTextURL(url, param){
  var myself = this;
  bhv.sendRequest("GET", url, param, true,
       handleLoadingTextURL, null, [myself]);
}

function handleLoadingTextURL(myself){
  var strResponse=new String(this.responseText);
  strResponse=strResponse.replace(/</g,"&lt;");
  strResponse=strResponse.replace(/>/g,"&gt;");
  strResponse=strResponse.replace(/\n/g,"<br>");
  strResponse=strResponse.replace(/ /g," ");
  strResponse=strResponse.replace(/\t/g,"    ");
  myself.content.innerHTML=strResponse;
}

function loadIFrame(url){
```

```
  var iFrame = document.createElement("IFRAME");
  iFrame.src = url;
  iFrame.width="100%"
  iFrame.height ="100%";
  iFrame.scrolling="auto";
  this.content.appendChild(iFrame);
}

function loadDiv(oDiv){
  if (typeof oDiv == "string")
    oDiv=document.getElementById(oDiv);
  oDiv.parentNode.removeChild(oDiv);
  this.content.appendChild(oDiv);
}
```

В функции-конструкторе объекта `WindowURL` вызывается функция `bhv.derive()`, которая при первом вызове загружает программный модуль с определением объекта `BaseWindow` и приписывает новые функции-методы к объекту `WindowURL`. Эти новые функции-методы будут обеспечивать загрузку HTML-документа `loadURL()`, загрузку текстового документа, который не содержит HTML-разметку `loadTextURL()` и загрузку HTML-документа в элемент `IFRAME`:

```
function WindowURL(title){
  this.derive("bhv.widget.BaseWindow",title);
  this.loadURL=loadURL
  this.loadTextURL=loadTextURL
  this.loadIFrame=loadIFrame
  this.loadDiv=loadDiv
};
```

Для упрощения программного кода я привязываю новые функции-методы прямо к неявному параметру `this` внутри функции-конструктора. Можно сделать это более культурно при помощи разработанной в *главе 11* функции `bhv.isa()`, которая переписывает свойства одного объекта в другой объект:

```
bhv.isa(WindowURL.prototype, {
  loadURL: loadURL,
  loadTextURL:loadTextURL,
  loadIFrame:loadIFrame,
  loadDiv:loadDiv
});
```

Вместо такого подхода к определению новых свойств объекта-прототипа иногда используют упрощенный подход (в книге также используется такой код):

```
WindowURL.prototype = {
  loadURL: loadURL,
  loadTextURL:loadTextURL,
  loadIFrame:loadIFrame,
  loadDiv:loadDiv
};
```

Такой упрощенный подход переопределяет заданный интерпретатором JavaScript объект-прототип, что приводит к искажению объектной модели JavaScript. Поэтому такой подход не следует применять, если программный код использует тонкую работу с объектной моделью JavaScript, как, например, смешивание объектов.

Функции, загружающие HTML-документ или текст в плавающее окно, заданы в паре: функция, вызывающая запрос `XMLHttpRequest`, и функция, обрабатывающая ответ Web-сервера:

```
function loadURL(url, param){
  var myself=this;
  bhv.sendRequest("GET", url,param, true,
       handleLoadingURL, null, [myself,url]);
}

function handleLoadingURL(myself, url){
  var strResponse = bhv.relocateSRC(this.responseText, url);
  myself.content.innerHTML = strResponse;
}
```

Функция `bhv.reloacteSRC()` рассмотрена в *главе 3* при разработке компонента "Панель с закладками". Эта функция преобразует URL-адреса загружаемого документа к правильным адресам относительно текущего каталога. Такой подход позволяет правильно отобразить изображения (HTML-элементы `IMG`) и для некоторых типов Web-браузеров даже автоматически загрузить таблицы стилей.

Функции, загружающие текстовый документ, отличаются тем, что преобразуют пробельные символы и угловые скобки в их эквивалент для HTML-документа:

```
function loadTextURL(url, param){
  var myself = this;
  bhv.sendRequest("GET", url, param, true,
```

```
        handleLoadingTextURL, null, [myself]);
}

function handleLoadingTextURL(myself){
  var strResponse=new String(this.responseText);
  strResponse=strResponse.replace(/</g,"&lt;");
  strResponse=strResponse.replace(/>/g,"&gt;");
  strResponse=strResponse.replace(/\n/g,"<br>");
  strResponse=strResponse.replace(/ /g," ");
  strResponse=strResponse.replace(/\t/g,"    ");
  myself.content.innerHTML=strResponse;
}
```

Наконец, реализована функция, загружающая документ в HTML-элемент `IFRAME`. Загрузка документа в "Плавающее окно", которое содержит `IFRAME`, решает сразу несколько задач:

- в такое окно может быть загружен документ с произвольного домена, поскольку защита "песочницы" это допускает;
- будет загружен не только текст документа, но и все связанные ресурсы: изображения, таблицы стилей, скрипты. При этом не требуется приведение адресов к каталогу текущего документа;
- в документе, загруженном в HTML-элемент `IFRAME`, доступна обычная навигация, переход по ссылкам.

На рис. 12.1 вы можете увидеть два плавающих окна, одно из которых представляет собой объект `BaseWindow`, а другое — объект `WindowURL` с содержимым, загруженным в HTML-элемент `IFRAME`:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <title>Плавающие окна</title>
    <link rel="stylesheet" type="text/css"
          href="../bhv/widget/window.css">
    <script type="text/javascript" src="../bhv/util.js"></script>
    <script type="text/javascript" src="../bhv/classe$.js"></script>
    <script type="text/javascript" src="dnd.js"></script>
  </head>
  <body>
    <script>
```

```
      var window1 = bhv.create("bhv.widget.BaseWindow",
                               "Объект BaseWindow");
      var window2 = bhv.create("bhv.widget.WindowURL",
                               "Объект WindowURL");
      window1.show();
      window2.show();
      window2.loadIFrame("http://google.com:80");
    </script>
  </body>
</html>
```

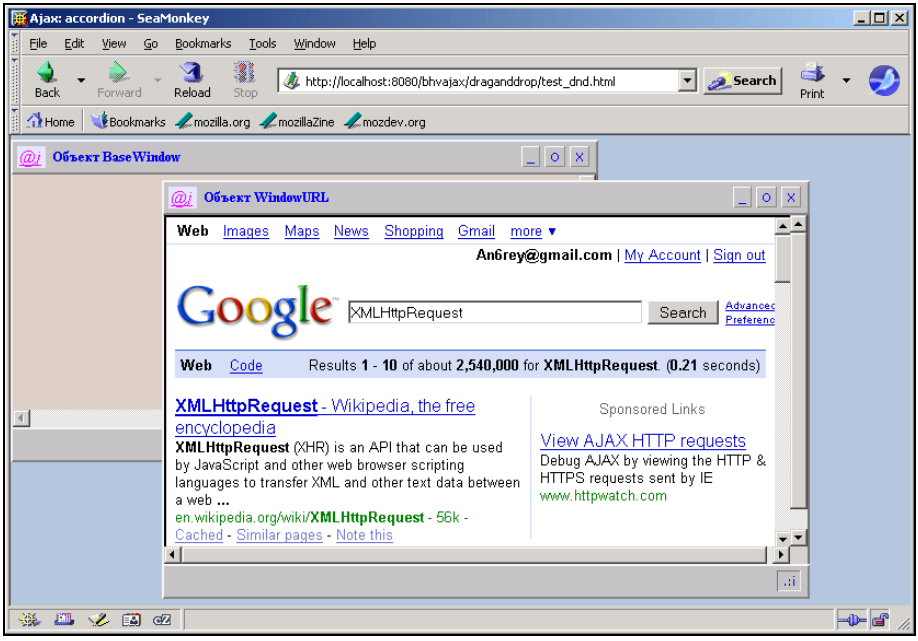
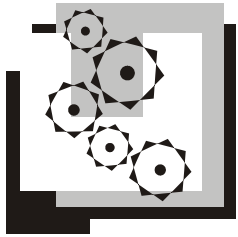


Рис. 12.1. Плавающие окна в окне Web-браузера

Этим примером позвольте закончить мое повествование о бурно развивающейся технологии Ajax.

ПРИЛОЖЕНИЯ



Приложение 1

Применение библиотек JavaScript при разработке Ajax-приложений

В настоящее время имеется возможность использовать готовые библиотеки JavaScript при разработке Ajax-приложений. Как правило, необходимость использования библиотек сторонних разработчиков приходит не сразу. И это нормальное явление. Потому что, прежде чем использовать готовые средства, программист должен получить минимальный опыт разработки, чтобы понимать, для чего ему нужны готовые библиотеки.

Как правило, прежде чем использовать ту или иную функциональность, предоставляемую готовыми библиотеками, разработчик должен изучить код библиотеки. Только в этом случае работа скриптов будет находиться полностью под контролем. Некоторые библиотеки, с которыми мне приходилось работать на практике, я опишу в этом приложении.

П1.1. Библиотека поддержки кроссбраузерности x.js (Coross-Browser.com)

Библиотека x.js является одной из самых полезных библиотек, разработанных на JavaScript, несмотря на то, что она не работает с технологией Ajax. Эта библиотека обеспечивает разработчика уникальной возможностью создавать приложения для достаточно широкого круга Web-браузеров, практически не задумываясь об обеспечении кроссбраузерности. Мы все, так или иначе, пытаемся обеспечить кроссбраузерность работы наших приложений. Но только изучив код библиотеки x.js, понимаешь, сколько тонкостей еще необходимо учесть и как много было упущено. Поэтому библиотеку x.js имеет смысл использовать в любом Ajax-приложении, несмотря на то, что эта библиотека не является Ajax-библиотекой.

Перечень функций, определенных в библиотеке x.js, может на первый взгляд оказаться скупым и даже неполным. Тем не менее, этих функций вполне хватает для разработки сложнейших Web-интерфейсов.

Функции библиотеки x.js начинаются с префикса `x` — `xTop()`, `xLeft()`, `xWidth()` и т. д. В качестве первого параметра этой функции передается ссылка на HTML-элемент или, в некоторых случаях, его идентификатор.

В листинге П1.1 приведен HTML-документ, который использует библиотеку x.js.

Листинг П1.1. Применение библиотеки x.js (Cross-Browser.com)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<style>
html, body {  top: 0px; left: 0px;
  border: 0px; padding: 0px; margin: 0px;
  height: 100%; width: 100%;
}
div {
  position: absolute;
  background: red;
  width: 40px;
  height: 40px;
  border: solid 10px yellow;
  margin: 20px
}
</style>
<script type="text/javascript" src="x.js"></script>
</head>
<body>
<div id="div1" onclick="expand_div1(event);"></div>
<script type="text/javascript">
xAddEventListener(document,"click",move_div1)
function move_div1(event){
  var oEvent = new xEvent(event);
  var oDiv1 = xGetElementById("div1");
 // xMoveTo(oDiv1, oEvent.pageX, oEvent.pageY);
  xSlideTo(oDiv1, oEvent.offsetX - xWidth(oDiv1)/2,
          oEvent.offsetY - xHeight(oDiv1)/2, 1000);
}
```

```
function expand_div1(event){
  xStopPropagation(event)
  var oDiv1 = xGetElementById("div1");
  xTop(oDiv1, 0);
  xLeft(oDiv1,0);

  xWidth(oDiv1, xClientWidth()
    -xGetComputedStyle(oDiv1, "border-left-width", true)
    -xGetComputedStyle(oDiv1, "border-right-width", true)
    -xGetComputedStyle(oDiv1, "margin-left", true)
    -xGetComputedStyle(oDiv1, "margin-right", true));

  xHeight(oDiv1, xClientHeight()
    -xGetComputedStyle(oDiv1, "border-top-width", true)
    -xGetComputedStyle(oDiv1, "border-bottom-width", true)
    -xGetComputedStyle(oDiv1, "margin-top", true)
    -xGetComputedStyle(oDiv1, "margin-bottom", true));
}
</script>
</body>
</html>
```

В приведенном примере HTML-элемент `DIV` можно перемещать в то место на экране, где пользователь щелкнул кнопкой мыши. Если пользователь щелкнул по самому HTML-элементу `DIV`, этот элемент распахивается, занимая пространство по высоте и ширине, равное доступному для рисования пространству в окне Web-браузера, с учетом границ, рамок и отступов. Если вы попробуете организовать подобную функциональность самостоятельно с учетом кроссбраузерности, то очень скоро оцените все преимущества от использования библиотеки x.js. В приведенном примере нет ни одного прямого (т. е. в коде скрипта) обращения к свойствам и методам объектной модели документа (DOM), и вся работа ведется посредством функций, определенных в библиотеке x.js.

Давайте рассмотрим некоторые из них. Вызов функции:

```
xAddEventListener(document,"click",move_div1)
```

кроссбраузерно назначает обработчик событию `onclick` документа. При этом сначала библиотека пытается воспользоваться назначением обработчика событий в новом стиле, используя функции `addEventListener()` или `attachEvent()`. Если это не удается, то используется назначение обработчика

в старом стиле. Если вы изучите код этой функции, то увидите, что помимо той работы, которую мы можем предположить, эта функция выполняет еще целый рад проверок, смысл которых сходу и не разберешь. Поэтому есть масса причин основывать свои разработки с использованием готового кода поддержки кроссбраузерности из библиотеки x.js.

Давайте пойдем дальше вслед за кодом нашего примера:

```
var oEvent = new xEvent(event);
```

Всем известно, что Web-браузеры по-разному передают в функцию-обработчик события объект `event`. Часть Web-браузеров передает этот объект в первом параметре функции-обработчика события, а другая часть позволяет обращаться к объекту как к глобальному объекту `window.event`. Такое поведение имеет давнюю историю. Дело в том, что спецификация HTML определяет перечень событий, который должны поддерживать HTML-элементы, но не определяет, как этим событиям должны назначаться функции-обработчики, и, тем более, не определяет, какие свойства должны содержать объекты `Event`. Поэтому библиотека x.js на базе объекта `event`, переданного в качестве фактического параметра, или события `window.event` создает свой объект `xEvent`. Этот объект имеет строго заданный перечень свойств, из которых достаточно просто извлечь, например, координату указателя мыши с учетом таких тонких и некроссбраузерных условий, как, например, текущее положение скроллинга.

Функции библиотеки x.js могут использоваться как для извлечения свойств HTML-элементов:

```
xTop(oDiv1);
```

так и для изменения свойств HTML-элементов:

```
xTop(oDiv1, 0);
```

При помощи функции:

```
xGetComputedStyle(oDiv1, "margin-top", true)
```

можно узнать текущее значение свойства стиля HTML-элемента.

Библиотека x.js поддерживает некоторые простые визуальные эффекты:

```
xSlideTo(oDiv1, oEvent.offsetX - xWidth(oDiv1)/2,
                oEvent.offsetY - xHeight(oDiv1)/2, 1000);
```

Приведенный код плавно перемещает HTML-элемент, заданный в первом параметре, в координату, заданную вторым и третьим параметрами функции.

Наконец, функция `xGetElementById()` действует аналогично функции с похожим названием из DOM Level 1 `document.getElementById()`. Но в отли-

чие от этой функции, `xGetElementById()` в качестве параметра может быть задана не только строка, но и HTML-элемент, который будет возвращен оператором `return`. Такая функциональность на первый взгляд может показаться лишней, но опыт показывает, что эта функция используется очень часто. Даже в самой библиотеке x.js эта функция вызывается практически во всех функциях, работающих с HTML-элементами. Дело в том, что поиск HTML-элемента по его идентификатору (атрибуту `id`) — одно из самых распространенных действий при работе с DOM. В определенный момент переменных, содержащих ссылки на объекты и имена идентификаторов, становится так много, что наверняка сказать, что содержит переменная — идентификатор HTML-элемента или ссылку на этот элемент — становится сложно. Все сложности устраняет использование функции `xGetElementById()`, которая гарантированно возвращает HTML-элемент, если он существует, независимо от способа его задания в первом параметре функции. Эту функциональность, с рядом усовершенствований, обеспечивают практически все современные Ajax-библиотеки.

П1.2. Библиотека jsolait (JavaScript Object Lait)

Название библиотеки происходит от сокращения JavaScript Object Lait. Мне лично очень нравится эта библиотека, поэтому я поставил ее на второе место, сразу после x.js. Эта библиотека реализует довольно нестандартный набор средств, если ее сравнивать с другими Ajax-библиотеками. Библиотека написана в виде нескольких связанных модулей. Модули библиотеки jsolait создаются с соблюдением некоторых соглашений и могут динамически загружаться при помощи запросов `XMLHttpRequest`. Библиотека jsolait обеспечивает Web-разработчиков следующими средствами: создание и динамическая загрузка пользовательских модулей и классов, работа со строками, кодировками, шифрованием, XML, а также удаленным вызовом процедур XML-RPC и JSON-RPC.

Все возможности библиотеки реализованы на высоком уровне культуры (за исключением использования на глобальном уровне тривиальных имен). Поэтому библиотека легко расширяется. Вы можете использовать встроенные модули и создавать собственные и встраивать их в систему. Можете использовать реализованные в библиотеке кодеки и шифры, а также легко добавлять свои кодеки и шифры.

В качестве примера разберем работу с удаленным вызовом процедур XML-RPC. Программисты обожают удаленный вызов процедур. В PHP 5 предусмотрена возможность работы с XML-RPC, которая на момент написания

книги (июль 2008 года) являлась экспериментальной, была не полностью документирована и может быть изменена в следующих релизах (и, скорее всего, будет изменена). Тем не менее, воспользуемся уже имеющимися возможностями PHP 5:

```
<?php
header('Content-Type: application/xml; charset="utf-8"');

function getValue($a,$b){
  return $b[0] * $b[1];
}

$server = xmlrpc_server_create();
xmlrpc_server_register_method($server, 'test', 'getValue');
$output=xmlrpc_server_call_method($server, $HTTP_RAW_POST_DATA,
  Array(), Array('encoding'=>'utf-8'));
echo $output;
?>
```

В этом фрагменте кода регистрируется сервер XML-RPC, где под именем `test` регистрируется функция `getValue()`. Возможно, такой подход с созданием удаленного сервера, который выполняет процедуры XML-RPC, покажется вам непривычным. Но повторяю, что на момент написания книги поддержка XML-RPC в PHP 5 была еще на уровне экспериментальной. Возможно, вы поделитесь своими соображениями с разработчиками, и они будут учтены в следующих релизах.

XML-документ, который содержит запрос, можно извлечь из предопределенной переменной PHP 5 `$HTTP_RAW_POST_DATA`.

А вот как выглядит удаленный вызов из кода JavaScript с использованием библиотеки jsolait:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <script src="jsolait/jsolait.js"></script>
  </head>
  <body>
    <script>
      var xmlrpc = imprt("xmlrpc");
      var method = new xmlrpc.XMLRPCMethod("xmlrpc.php","test","","")
```

```
      var result = method(2, 6);
      alert(bb);
    </script>
  </body>
</html>
```

Вызов функции:

```
var xmlrpc = imprt("xmlrpc");
```

загружает модуль xmlrpc библиотеки jsolait в интерпретатор JavaScript (если до этого он еще не был загружен), после чего его функции становятся доступными для Web-приложения. Затем создается удаленный метод:

```
var method = new xmlrpc.XMLRPCMethod("xmlrpc.php","test","","")
```

Теперь этот метод может вызвать как самую обычную функцию JavaScript:

```
var result = method(2, 6);
```

Удобно, не правда ли?

П1.3. Библиотека Prototype.js — новый стиль программирования на JavaScript

Библиотека Prototype.js имеет неоспоримый авторитет среди разработчиков Ajax-приложений. Целый ряд профессиональных библиотек выбрал эту библиотеку в качестве основы ("движка") для разработки собственных библиотек.

Библиотека Prototype.js ознаменовала собой новый этап в истории программирования на JavaScript. В первые годы существования JavaScript этот язык программирования ассоциировался у подавляющего большинства средних Web-разработчиков с достаточно примитивными скриптами, не вызывающими большого энтузиазма.

В известном смысле, медвежью услугу сыграл маркетинговый ход, когда совершенно оригинальному языку программирования ECMAScript решили дать маркетинговое название JavaScript.

Попытка программировать на JavaScript в стиле C или своего неполного тезки натыкалась на препятствия, которые поначалу воспринимались как недостатки JavaScript, а потом были использованы разработчиками как его преимущества: нестрогая типизация, прототипы объектов. Но время шло, и начал складываться новый стиль программирования на JavaScript, который открыл самобытность и неповторимость этого языка программирования. В это время и появилась библиотека Prototype.js, которая на 100% была написана в совершенно новом стиле. Web-разработчикам не привыкать к тому, что один

и тот же язык программирования может выглядеть по-разному, поскольку ярчайшим примером такого явления являются диалекты языка программирования Perl. В отличие от Perl, JavaScript, скорее всего, не даст обилия диалектов, но новый стиль программирования на JavaScript — уже реальность, которая закреплена в библиотеке Prototype.js.

Библиотека Prototype.js реализует работу с запросами Ajax, коллекциями объектов, HTML-элементами, а также поддержку кроссбраузерности.

Давайте рассмотрим простейший пример использования возможностей библиотеки:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <script src="prototype.js"></script>
  </head>
  <body>
    <div id="output">Сюда будет загружен текст документа</div>
    <script>
      new Ajax.Request("../accordion/accordion.html",
      {
        onSuccess: function(xhr, json) {
          alert(xhr.responseText);
          $("output").innerHTML = xhr.responseText;
        }
      }
      )
      a = [1,2,3,4,5]
      a.each(function(element){
        alert(element);
      }
      )
      a1= a.grep(/3/)
      a2 = a.map(function(element){
        return element*2
      }
      )
      a3 = a.select(function(element){
        return element%2==0
```

```
      }
      )
    </script>
  </body>
</html>
```

Использование функции `$("output")` — это альтернатива вызову метода `document.getElementById("output")`. Для отправки Ajax-запроса к серверу используется объект `Ajax.Request`. В первом параметре функции-конструктора этого объекта передается URL загружаемого ресурса, а во втором параметре — объект, в свойствах которого задаются параметры запроса:

```
{
  onSuccess: function(xhr, json) {
    alert(xhr.responseText);
    $("output").innerHTML = xhr.responseText;
  }
}
```

В частности, свойство `onSuccess` содержит ссылку на функцию, которая будет вызвана в случае успешного запроса. Эта функция имеет предопределенную сигнатуру: `function(xhr, json)`. В первом параметре этой функции будет передаваться ссылка на объект `XMLHttpRequest`, а во втором параметре — ссылка на объект, полученный с Web-сервера, если пришел ответ в JSON-нотации.

Далее, показаны возможности работы библиотеки Prototype.js с коллекциями. Разработчик библиотеки проявил незаурядную изобретательность для того, чтобы работа с коллекциями на JavaScript приблизилась по своему стилю к работе с итераторами в некоторых языках программирования (таких как CLU), в которых имеется такая реализация. Например, вызов функции:

```
a1 = a.grep(/3/);
```

создает на базе массива `a` массив `a1`, который содержит элементы из массива `a`, соответствующие регулярному выражению (шаблону) `/3/`.

Надо заметить, что функционал библиотеки Prototype.js чаще используют разработчики библиотек JavaScript, чем разработчики Web-приложений. Это связано с тем, что библиотека содержит только базовые средства для разработки, но в ней отсутствуют реализации компонентов пользовательского интерфейса, визуальных эффектов. Эти возможности обеспечивают другие библиотеки, которые используют библиотеку Prototype.js в качестве базовой для своих разработок. И с одной из таких библиотек мы познакомимся прямо сейчас.

П1.4. Применение библиотеки scriptaculous для разработки Ajax-приложений

Библиотека scriptaculous использует в качестве базовой библиотеки Prototype.js, которую мы рассмотрели в предыдущем разделе. Такой подход позволяет библиотеке сосредоточиться на прикладных аспектах. И эффект не заставил себя ожидать. Библиотека реализует ряд интересных визуальных эффектов, полезных компонентов пользовательского интерфейса, таких как Autocompleter, Slider и технологию drag-and-drop.

Кстати, название библиотеки не такое уж ridiculous, как это может показаться на первый взгляд, а очень даже зубастое и происходит от доменного имени разработчика **script.aculo.us**.

Приведу код с использованием библиотеки scriptaculous:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <script src="jsolait/aculo/lib/prototype.js"></script>
  <script src="jsolait/aculo/src/scriptaculous.js"></script>
  <script src="jsolait/aculo/src/effects.js"></script>
  <script src="jsolait/aculo/src/dragdrop.js"></script>
</head>
<body>
  <div id="div1" style="width:40;height:40;background:red">123</div>
  <div id="div2" style="width:40;height:40;background:red">456</div>
  <div id="div3" style="width:40;height:40;background:red">789</div>
  <div id="div4" style="width:40;height:40;background:red">101112</div>
  <script>
    new Draggable ($("div1"))
    Effect.Puff($("div2"))
    Effect.Shrink($("div3"))
    Effect.Highlight($("div4"))
  </script>
</body>
</html>
```

Для работы библиотеки необходимо подключить библиотеки Prototype.js. Задав HTML-элемент в конструкторе объекта `Draggable`, мы тут же получаем возможность перетаскивать (drag-and-drop) этот HTML-элемент. Возможности

drag-and-drop библиотеки scriptaculous очень широкие и поддерживают плавную визуализацию, перенос HTML-элемента из одной коллекции в другую, обработку событий, специфичных для drag-and-drop.

Визуальные эффекты использовать достаточно просто, в чем вы можете убедиться, посмотрев на код примера. Например, визуальный эффект `Effect.Puff($("div2"))` плавно увеличивает в размере и одновременно "растворяет в воздухе" HTML-элемент, заданный в качестве фактического параметра.

П1.5. Богатство и разнообразие библиотек JavaScript

До этого момента я рассмотрел несколько библиотек JavaScript и простые примеры их использования. В настоящее время Web-разработчики могут выбирать из широкого круга библиотек JavaScript то, что им нужно. В этом разделе я назову лишь некоторые из этих библиотек. То, что я не посвящаю каждой из этих великолепных библиотек отдельный раздел, связано только с тем, что я не в состоянии проработать код этих библиотек так же тщательно, как и в предшествующих разделах приложения, и не могу приводить примеры использования этих библиотек.

Библиотека YUI (YHOO User Interface) реализует и базовые возможности, и конкретные компоненты пользовательского интерфейса, такие как меню, таблицы, окна. К положительным особенностям этой библиотеки относятся профессиональный художественный дизайн и глубокая проработка каждого модуля библиотеки. Библиотека построена по модульному принципу. При этом каждый модуль ведет разработчик или группа разработчиков, которые ищут наиболее удачные решения и стараются проработать свой модуль как можно более тщательно. Разработчики модулей ведут блоги, иллюстрируют свои модули интересными примерами, переписываются с Web-программистами, использующими библиотеку YUI. Впрочем, в этой переписке, в основном, превалирует восхищение и благодарность, к которым я с удовольствием присоединяюсь.

Библиотека jQuery реализует базовые средства работы для разработчика Ajax-приложений (подобно Prototype.js). Прикладные разработчики в последнее время предпочитают использовать именно библиотеку jQuery. Это можно объяснить тем, что библиотека написана простым доступным языком, не использует головоломных конструкций и не засоряет глобальное пространство тривиальными именами.

Для работы с XML-документами, включая XSLT-преобразование на стороне Web-браузера, вам может пригодиться библиотека Sarissa.

Особо следует отметить библиотеку dojo. Эта библиотека реализует поддержку модульной разработки, динамическую загрузку модулей, а также оригинальную систему поддержки классов и наследования. Библиотека реализует поддержку самой широкой функциональности, в частности создание компонентов пользовательского интерфейса, которые построены на строгой иерархии классов.

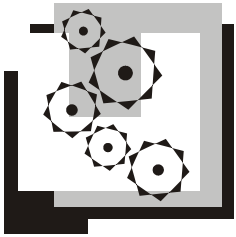
На этом я закончу список обсуждаемых библиотек, потому что привык говорить только о том, с чем работал на практике. На самом деле этот список можно было бы продолжать и продолжать.

Давайте обсудим, на какие моменты следует обратить особое внимание при выборе библиотеки в ваших приложениях.

Библиотеки могут по-разному работать с глобальными именами приложения. Лучше, если библиотека не использует тривиальных глобальных имен и следует соглашениям об именовании переменных по доменному имени разработчика или по имени библиотеки. Кроме того, библиотеки могут дополнять встроенные объекты JavaScript, такие как `Object`, `Array`, `String`, новыми методами. В предельном случае, библиотека может переопределять функции, такие как `window.alert()`, так что некоторые моменты в поведении программного кода могут стать для вас неожиданностью.

Желательно использовать ровно одну библиотеку и хорошо изучить ее по документации и по исходному коду JavaScript. Но часто у разработчика Web-приложения возникает необходимость в дополнительной функциональности, и приходится загружать дополнительную библиотеку. Совместно используемые библиотеки должны быть изучены на предмет совместимости, которая, в первую очередь, касается использования имен глобальных переменных и функций, а также определения дополнительных методов встроенных объектов JavaScript.

Разумеется, разработчик Web-приложения должен с такими требованиями подходить не только к разработчикам библиотек JavaScript, но и к своим собственным разработкам.



Приложение 2

Описание содержимого компакт-диска

Таблица П2.1. *Описание компакт-диска*

Папка, файл	Описание	Глава
\readme.txt	Рекомендации по установке примеров	Главы 1—12
\bhvajax	Главная папка приложения, которая содержит все папки приложения	Главы 1—12
\bhvajax\bhv	Папка содержит базовые утилиты для работы примеров	Главы 2—12
\bhvajax\bhv\util.js	Библиотека функций JavaScript, которая содержит базовые средства, используемые в примерах	Главы 2—12
\bhvajax\accordion	Папка содержит компонент "Аккордеон"	Глава 1
\bhvajax\xmlhttprequest	Папка содержит примеры работы с объектом `XMLHttpRequest`	Глава 2
\bhvajax\tabbedpane	Папка содержит компонент "Панель с закладками"	Глава 3
\bhvajax\text	Папка содержит HTML-документы для тестирования компонента "Панель с закладками"	Глава 3
\bhvajax\xml	Папка содержит примеры работы с документами XML	Глава 4
\bhvajax\menu	Папка содержит компонент "Полоска меню"	Глава 5
\bhvajax\editor	Папка содержит компонент "Редактор кода — отладчик PHP 5"	Глава 6
\bhvajax\mysql	Папка содержит компонент "Консоль базы данных MySQL 5"	Глава 7

Таблица П2.1 *(окончание)*

Папка, файл	Описание	Глава
\bhvajax\profile	Папка содержит реализацию системы управления пользователями Web-приложения	Глава 8
\bhvajax\combobox	Папка содержит компонент Lookup Combobox	Глава 9
\bhvajax\table	Папка содержит компонент "Редактируемые таблицы данных"	Глава 10
\bhvajax\test	Папка содержит HTML-документы для тестирования компонентов Lookup Combobox и "Редактируемые таблицы данных"	Главы 9, 10
\bhvajax\bhv\classe$.js	Библиотека функций JavaScript, которая обеспечивает работу с модулями и объектами	Глава 11
\bhvajax\bhv\classes.js	Вариант библиотеки classe$.js, использующий объекты пространства имен	Глава 11
\bhvajax\bhv\widget	Папка содержит реализацию компонента "Плавающее окно"	Глава 12
\bhvajax\draganddrop	Папка содержит библиотеку поддержки технологии drag-and-drop и HTML-документы для тестирования компонента "Плавающее окно"	Глава 12

Предметный указатель

(окончание рубрики см. на стр. 424)