# Разработка CGI-приложений на Perl

# Writing CGI Applications with Perl

Kevin Meltzer
Brent Michalski

Addison-Wesley

*Boston • San Francisco • New York • Toronto • Montreal*
*London • Munich • Paris • Madrid*
*Capetown • Sydney • Tokyo • Singapore • Mexico City*

# Разработка CGI-приложений на Perl

Кевин Мельтцер
Брент Михальски

Москва • Санкт-Петербург • Киев
2001

ББК 32.973.26-018.2.75
М48
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *А.В. Высоцкого*

По общим вопросам обращайтесь в Издательский дом "Вильямс"
по адресу: info@williamspublishing.com, http://www.williamspublishing.com

М48 **Мельтцер Кевин, Михальски** Брент.

Разработка CGI-приложений на Perl. : Пер. с англ. — М. : Издательский дом "Вильяме", 2001. — 400 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0211-8 (рус.)

Эта книга научит вас применять Perl для решения задач, необходимых в современных сетевых приложениях. Множество примеров и еще более многочисленные упражнения дают не просто полезную информацию, но и готовые программы, которые вы можете сразу поместить в свои приложения. Книга поможет вам овладеть практическими приемами и методами, с которыми вы сможете разрабатывать на Perl любые Web-приложсния на основе протокола CGI.

Основное внимание в книге уделяется важным вопросам разработки Web-приложений, таким как работа с базами данных, обработка форм и файлов, безопасность, электронная почта и работа с графикой. Кроме того, в этой книге подробно рассматриваются некоторые более специальные темы: обработка Web-форм и получение через них данных пользователя, файлы cookie, отслеживание щелчков и счетчики доступа, применение модуля Apache mod_perl, связывание переменных с базами данных, встраивание кода Perl в HTML при помощи модуля HTML:Mason, управление документами через Web, создание динамических изображений, применение XML и его производных — RSS и RDF.

Книга рассчитана на программистов средней и высокой квалификации.

**ББК 32.973.26-018.2.75**

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Addison-Wesley Publishing Company, Inc.

Authorized translation from the English language edition published by Addison-Wesley Publishing Company, Inc Copyright © 2001

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2001

ISBN 5-8459-0211-8 (рус.)
ISBN 0-201-71014-5 (англ.)

О Издательский дом "Вильяме", 2001
© Addison-Wesley Publishing Company, Inc

# Оглавление

# Содержание

*От Брента:*

*Крису, Люку, Рей и Логену. Давайте бороться!*

*От Кевина:*

*Моей дочери Кайле и моей жене Сузи. Только ради вас у меня была причина отрываться от компьютера.*

# Вступление

Вспышка популярности Web в середине 90-х гг., оттеснившая на задний план Gopher, WAIS, Hyper-G и другие конкурирующие технологии, большей частью своего феноменального успеха была обязана той легкости, с которой разработчики могли связывать механизмы обработки данных с привлекательными интерфейсами Web-страниц. Возможности Web уже не были ограничены сложным программным интерфейсом к немногим языкам программирования; напротив, эта технология предложила общий шлюзовый интерфейс (Common Gateway Interface, или CGI).

CGI — это своего рода болт, который скрепляет Internet. Он не зависит ни от языка, ни от платформы и, что лучше всего, им легко овладеть. Скрипт CGI, генерирующий динамическую Web-страницу, состоит лишь из нескольких строк кода. Ненамного большие затраты труда позволяют принимать данные пользователя из заполненной формы, передавать их в прикладную программу или базу данных для обработки и генерировать новую страницу Web с выводом результатов. CGI позволяет придать любому вновь написанному или старому приложению форму, пригодную для работы через Web.

Почти с самого начала CGI был связан с языком программирования Perl. По сути, для многих CG1 и Perl — синонимы. Есть серьезные основания так считать. Если CGI — это крепежная деталь для Internet, то Perl можно уподобить скотчу. Возможности межпроцессной связи Perl наряду с его мощными средствами синтаксического анализа текста создают окружение, которое облегчает объединение разных программных компонентов в единое целое.

Например, типичный сайт электронной коммерции должен "уметь" соединяться с поисковым сервером, отображать страницы каталога с обновленной ценовой информацией из базы данных, управлять "тележкой для покупок", принимать заказы клиентов, проверять кредитные карточки и управлять выполнением заказов. В задачах такого типа Perl превосходит другие языки. Он может посылать запрос на поисковый сервер и преобразовывать результаты его выполнения в Web-страницу с гиперссылками, оперативно генерировать непрерывные страницы каталога исходя из информации базы данных товаров, передавать через сеть информацию кредитной карточки в службу проверки и вводить заказ пользователя в базу данных заказов.

Диапазон передовых приложений, которые уже построены на базе Perl/CGI, по меньшей мере удивителен. Туристические агентства используют Perl/CGI для создания интерактивных карт городов и местностей. В медицинских школах Perl/CGI служит для интерактивного моделирования физиологии человека. В проекте генома человека при помощи Perl/CGI среди участников биологических исследований распро-

страняются генетические карты и последовательности. Perl/CGI также применяется для подключения к Web самых разнообразных устройств, начиная с подводных видеокамер и кончая манипуляторами роботов.

За эти годы технология Web стала намного сложнее. Уже недостаточно только принимать данные пользователя из заполненной формы и генерировать в ответ страницу HTML. Web-сайты должны уметь анализировать XML, генерировать XHTML и DHTML, управлять данными cookie и взаимодействовать со все более и более распределенными конечными системами. Требования пользователей к Web-сайту также возросли. Сегодня пользователи хотят, чтобы Web-сайт запоминал их с прошлого посещения, и чтобы они могли настраивать сайт в соответствии со своими специфическими потребностями.

К счастью, с развитием Web Perl и CGI также достаточно "выросли", чтобы удовлетворять этим требованиям. Эта книга демонстрирует, каким мощным и жизнеспособным остается союз Perl и CGI. На ее страницах вы научитесь, как соединять Web-страницы с базами данных, отслеживать действия пользователей, обмениваться информацией с другими сайтами с помощью XML и "на ходу" генерировать графику и мультипликацию. Кроме того, в этой книге равное внимание уделяется трем другим вопросам, которые часто игнорируются при поспешной разработке Web: безопасности, надежности и масштабируемости.

Я знаю, что комбинация Perl и CGI покажется вам как мошной, так и изящной — какой уже много лет видится она и мне.

Линкольн Штайн

Стонибрук, шт. Нью-Йорк

8 декабря 2000 г.

# Предисловие

## Цель этой книги

Популярность Perl как языка скриптов CGI стремительно возрастает. Однако в настоящий момент лишь немногие книги охватывают эту тему во всей ее глубине, включая широкий диапазон концепций. Мы хотели, чтобы наша книга помогла читателям научиться применять Perl и убедила их, что это наилучший выбор для создания Web-приложений. Цель этой книги состоит не в обучении языку Perl, хотя она позволяет изучить некоторые приемы и особенности, но в том, чтобы показать, как Perl может выполнять задачи, необходимые во многих нынешних сетевых приложениях. Короче говоря, мы хотели предложить книгу, в которой не только преподаются новые возможности применения Perl, но также читателям предлагаются упражнения с применением этих концепций. Также приводится стандартная документация по Perl. Книга охватывает широкий диапазон концепций и их применение. Пользуясь этими методами, вы сможете написать почти любое приложение Perl/CGI.

Эта книга отличается от другой литературы по Perl и CGI. В ней принят интегральный подход и приводятся приложения, в которых применяются изученные в предшествующих главах понятия. Каждая глава содержит по крайней мере одно специфическое Web-приложение и объяснение его кода строка за строкой (или блок за блоком), так что читатель узнает не только то, что делает скрипт, но и то, как он это делает. Также, чтобы стимулировать самообучение и самостоятельное формирование приложений, каждое приложение сделано работоспособным, но неполным. Мы приводим каркасные приложения, которые могут работать сами по себе, но не вводим в них некоторые функции, которые может добавить читатель (и мы предлагаем ему сделать это в упражнениях) на основе материала текущей и предыдущих глав. Наша цель — не дать вам программы, которые можно просто переписать, а скорее показать, как создавать такие программы самостоятельно.

## Краткое содержание глав

*Глава 1. "Perl, CGI и эта книга".* Эта глава объясняет, что такое Perl и CGI. В ней также приводятся дополнительные сведения об этой книге и объясняется, как использовать модуль CPAN.

*Глава 2. "Что вы должны знать".* Мы не ожидаем, что читатель будет знать все, но мы надеемся, по крайней мере, на базовый уровень знаний. В этой главе намечено, что вы уже должны знать, и объяснены некоторые вещи, которых вы можете не знать, но должны знать, чтобы получить максимум пользы от этой книги. Это, например, загрязнение данных, вопросы безопасности и устранение неисправностей.

*Глава 3. "Использование окружения".* В приложениях CGI часто требуется получать от клиента информацию, такую как IP-адрес или данные броузера. В этой главе рассказано, как получить доступ к переменным окружения Web-сервера и что они означают.

*Глава 4. "Введение в Web-формы".* Немного найдется сетевых приложений, в которых не используются Web-формы какого-то рода. Эти формы позволяют конечному пользователю вводить информацию. В главе 4 рассматриваются элементы HTML Web-форм и способы получения данных от пользователя.

*Глава 5. "Работа с cookie".* Применение объектов cookie для хранения данных у клиента Web может быть очень полезным для фиксации предпочтений пользователей и данных о них. В этой главе показано, как создавать, получать и использовать cookie.

*Глава 6. "Счетчики обращений".* Многие люди хотят знать, сколько пользователей посещают их Web-сайт. В примерах этой главы показано, как это реализовать.

*Глава 7. "Загрузка файлов на базе Web",* Здесь вы узнаете, как, не подвергая сервер опасности, позволить конечным пользователям загружать на него файлы с локальных дисков.

*Глава 8. "Отслеживание щелчков".* Иногда бывает полезно знать, по каким ссылкам на Web-сайте переходит пользователь и откуда. Примеры в этой главе показывают, как проследить эти действия пользователя.

*Глава 9. "Использование mod_perl".* Популярный модуль Apache `mod_perl` может быть чрезвычайно полезен, когда он соответствует приложению. Здесь показано, как сконфигурировать `mod_perl`, а также как использовать и создавать модули raod_perl на Perl.

*Глава 10. "Электронная понта на базе Web".* В примерах этой главы демонстрируется соединение с сервером POP3, просмотр электронной почты и вложений и отправка сообщений через Web.

*Глава 11. "Введение в DBI и базы данных в Web".* В примерах глав 1 — 10 были представлены основные области применения баз данных. В этой главе Perl DBI рассматривается более подробно.

*Глава 12. "Связанные переменные".* В этой главе описаны волшебные возможности связывания структур данных с переменными и показано, как это сделать, когда структура представляет собой базу данных.

*Глава 13. "Внедрение Perl в HTML с помощью Mason".* Ознакомление с этим популярным инструментом и его анализ. В этой главе показано, как встроить Perl в HTML и ускорить разработку при помощи модуля HTML: `:Mason`.

*Глава 14. "Управление документами через Web".* Теперь вы уже знаете, как загружать файлы на сервер. В этой главе рассматривается дистанционное управление этими файлами через Web.

*Глава 15. "Динамическая обработка изображений".* Создание диаграмм, графиков, эскизов и галерей и изменение изображений "на ходу" — все эти концепции могут быть полезны в приложениях CGI. Здесь показано, как включить эти функции в программу.

*Глава 16. "RSS и XML".* XML — еще одна технология с растущей популярностью. В этой главе рассматривается применение XML и его производных — RSS и RDF.

*Приложение А. "Коды сервера".* Рассматривается значение кодов, возвращаемых Web-сервером.

*Приложение Б. "Переменные окружения".* Это список наиболее общих переменных окружения Web-сервера.

*Приложение В. "Форматы POSIX::strftime()".* В этой книге несколько раз применяется модуль POSIX для форматирования строк даты. Здесь приведен список форматов этого модуля и их действие.

*Приложение Г. "Общедоступная лицензия".* Если вы не прочли копию этого документа, которая поставляется вместе с Perl, это можно сделать здесь.

*Приложение Д. "Творческая лицензия".* Это другая лицензия, по которой распространяется Perl.

*Приложение И. "Документация к Perl".* Список документации, которая поставляется вместе с Perl. Документация из этого списка полезна для интерактивного изучения наряду с этой книгой.

*Приложение Ж. "Коды ASCII".* Список символов ASCII и их шестнадцатеричных и десятичных кодов.

*Приложение З. "Специальные символы HTML".* Список специальных символов, таких как Ł, ® и ¶. Хотя они не определены в Perl, вам, вероятно, рано или поздно понадобятся некоторые из них при создании HTML.

## Другие источники

Один из лучших источников по Perl — это документация к Perl и к его различным модулям. В главе 1 показано, как читать эту документацию с помощью команды perldoc. Начальная страница Perl, http://www.perl.com, чрезвычайно полезна своими статьями, советами, документацией, ссылками на другие ресурсы и новостями из мира Perl. Начальная страница Perl Mongers на http://www.perl.org дает хорошую информацию о мире Perl и о преимуществах этого языка. Новости Perl на http://news.perl.org будут сообщать вам о самых последних событиях и выпусках новых модулей. Web-сайт Use Perl на http://use.perl.org — страница сообщества Perl, на которой распространяется и обсуждается информация о нем. Web-сайт документации Perl на http://www.perldoc.com — • очень полезный сайт, содержащий новейшую документацию по Perl. Наконец, Perl Monks на http://www.perlmonks.com — еще одно сообщество, участники которого могут задавать вопросы, отвечать на них, вести беседу и делиться знаниями.

Также существуют полезные группы новостей по Perl: comp.lang.perl.announce содержит объявления в области Perl; comp.lang.perl.misc — популярная группа для вопросов, имеющих отношение к Perl, а comp.lang.perl.modules — для представления и обсуждения модулей Peri. Не связанная непосредственно с Perl группа новостей, посвященная CG1 — comp.infosystems.www.authoring.cgi. Здесь вы можете обсуждать все вопросы CG1.

## Как связаться с нами

Мы будем рады получить известие от вас. Информацию об этой книге и об опечатках МОЖНО найти на http://www.perlcgi-book.com и http://www.awl.com.

## От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать электронное письмо или просто посетить наш Web-сервер, оставив свои замечания, — одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более подходящими для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш e-mail. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: info@williamspublishing.com

WWW: http://www.williamspublishing.com

# Благодарности

## От Кевина

Создание книги — это целый процесс! Я мог бы назвать многих людей, которые непосредственно, и еще больше тех, которые косвенно помогали мне своей поддержкой, техническими советами и ответами на мои вопросы, когда я проверял свои идеи, выражения и код. В первую очередь я хотел бы поблагодарить мою жену Сузи за терпение, которое она проявляла весь прошлый год, пока я работал над этим проектом. Огромное спасибо моим матери и отцу за всю их поддержку и любовь! Я также хотел бы поблагодарить Элайн Аштон, Криса и Дэйла из OWLS, завсегдатаев конференции EFNET #perl, Скотта Фарлея. "Гладденз", Джона "Гуся" Госселина, Джозефа Холла, группу Hartford Perl Mongers, Клинтона Пирса, Джима Прайка из Netinstitute.com, Питера Скотта, Рэндала Шварца, Денниса Тейлора, Джима Вудгейта, Эда Райта и Иосефа Мендельсона. Я благодарен Марианн Курафас и Мэри Т. О'Брайен из издательства "Эддисон-Уэсли" за то, что они дали Бренту и мне возможность сделать это; Ларри Уоллу, без которого эта книга никогда не была бы напИсана; и, наконец, моей дочери Кайле, которой я посвящаю эту книгу. Я ожидаю того дня, когда и она начнет работать в Perl.

## От Брента

Я хочу начать с благодарности моей жене Крис, которая поддерживала меня все время, пока я писал эту книгу. Я очень восхищен ею. Она много вынесла в течение всего этого процесса и все еще остается со мной. Спасибо, бэби. Моим детям, Люку, Рей и Логену. Вы — мой мир, и я благодарю вас за терпение. Моим родителям — спасибо за всю вашу поддержку! Без вас я не был бы тем, кем я есть сейчас — в буквальном смысле. Особая благодарность Биллу Игеру. Он был моим командиром, когда я служил в BBC, и именно благодаря ему я смог покинуть службу, когда получил "настоящую работу". Билл верил в меня и помог мне осуществить мою мечту. Такими людьми, как он, военные могут гордиться. Да, еще участники конференции #perl, которые не раз выручали меня! Они были достаточно тактичны, чтобы не смеяться и не перебивать меня, когда я задавал глупые вопросы. Я должен поблагодарить Ларри Уолла, который стоял у истоков этого замечательного языка. Я уверен, что, когда он разрабатывал Perl, ему вовсе не думалось, что это затронет стольких людей. В заключение, я искренне благодарю Эла Гора за изобретение Internet. А теперь, парни, поборемся!

Глава

# Perl, CGI и эта книга

## Что такое Perl

В 1986 г., когда Ларри Уолл (Larry Wall) работал в Лаборатории реактивного движения NASA, перед ним стояла задача синхронизации обмена данными между компьютерами в Санта-Монике, Калифорния, и Паоли, Пенсильвания. Этот проект включал также задачу создания отчетов на основании обмена данными. В то время Ларри чувствовал, что никакой из доступных программных инструментов не подойдет для этой работы. Работая над этим проектом, он обнаружил определенные особенности доступных ему инструментальных средств. Язык C имел то преимущество, что мог управлять сложными конструкциями, а также обладал способностью проникать во внутренность различных процессов. Это свойство Ларри назвал "мультиплексностью". С другой стороны, языки командных оболочек позволяли быстро написать программу без особых затрат времени. (Ларри назвал это "тяпляпиальностью".) Perl появился от недостатка как мультиплексности, так и тяпляпиальности.

18 декабря 1987 г. Ларри представил версию 1.000 Perl в конференции Usenet comp. sources. Именно в ЭТОТ момент "практический язык извлечения данных и создания отчетов" (Practical Extraction and Report Language — Perl) был официально представлен миру. Perl смог заполнить нишу, не занятую никаким другим доступным языком, и поэтому начал привлекать разработчиков. Ниже приведена выдержка из руководства по Perl версии 1.000, которая прекрасно резюмирует этот новый язык.

```
НАЗВАНИЕ[1]
perl I Практический язык извлечения данных и создания отчетов
СИНТАКСИС
perl [опции] имя_файла параметры
ОПИСАНИЕ
Perl - интерпретируемый язык, оптимизированный для
просмотра произвольных текстовых файлов, извлечения
информации из этих текстовых файлов и печати отчетов,
основанных на этой информации. Это также хороший язык
для многих задач управления системой. Язык perl
разработан скорее как практичный (удобный, эффективный,
законченный), чем как красивый (компактный, изящный,
минимальный). Он сочетает в себе (по крайней мере,
по мнению автора) некоторые из лучших свойств C, sed,
awk и sh, так что те, кто знаком с этими языками, не
должны иметь особых трудностей с perl. {Историки языков
также обратят внимание на некоторые рудименты csh,
Pascal и даже BASIC|PLUS). Синтаксис выражений
весьма близко соответствует синтаксису C. Если у вас
есть задача, которая обычно требует использования sed,
awk или sh, но превосходит их возможности или должна решаться немного
быстрее, и вы не хотите писать такую
мелочь на C, то perl может вам подойти. Есть также
трансляторы, которые переводят скрипты с sed и awk на
язык perl. Ну все, хватит рекламы.
(18 декабря)
```

За следующие насколько лет Perl значительно развился; в июне 1988 г. появилась версия 2.000, а в октябре 1989 г. — версия 3.000[2]. В марте 1991 г. была выпущена версия 4.000[3] Perl. В это время, как и в 1992 г., Perl зарекомендовал себя как очень надежный и стал широко использоваться в системах Unix. Но теперь, когда он был настолько широко распространен, стали проявляться некоторые ограничения Perl. Perl был превосходен, когда для решения задач достаточно было коротких программ, но оказался громоздким, когда потребовалось перейти к созданию больших приложений. Чтобы исправить положение, группа во главе с Ларри, занимающаяся разработкой языка Perl, стала устранять проблемы, которые потенциально могли бы склонить пользователей к выбору другого языка. В результате появилась версия Perl 5.000 (конец 1994 г.).

В версии 5.000 Perl пришел к своего рода зрелости и стал в большей степени универсальным языком программирования, а не только "примочкой" для системных администраторов. В этой версии появились объекты, POD, perldoc, лексическая область видимости "my", ключевое слово "use" и другие усовершенствования.

Тогда, как и теперь, Perl использовался в качестве "клея" между различными форматами данных и приложениями. Иногда Perl нежно называют "Универсальной Бензопилой", так как он имеет множество применений и может быть полезен почти в любой ситуации. Когда Ларри Уолл создавал Perl для себя и своих сотрудников, он скомбинировал лучшие свойства разнообразных языков, и это наследие приносит пользу и всем остальным.

Сейчас Perl поддерживается группой так называемых "портеров" (perl5-porters, или p5p) во главе с Ларри Уоллом. Эта группа создает новые функции, устраняет ошибки, обновляет документацию и координирует новые выпуски Perl. Каждый может видеть,

---

[1] *Здесь и далее описания, комментарии и строковые константы, не влияющие на работу скриптов, даны в листингах в переводе на русский язык — Прим. ред.*

[2] *Первая версия, подпадающая под действие Общедоступной лицензии GNU.*

[3] *Кроме GPL, появилась Творческая лицензия.*

как происходит разработка Perl, и даже получать патчи для Perl и документацию, читая список рассылки perl5-porters.[4]. Архив этого списка можно найти в Web по адресу http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/.

# Что такое CGI

Общий шлюзовый интерфейс (Common Gateway Interface, или CGI) — это стандарт, который позволяет внешним программам взаимодействовать с серверами, например, серверами HTTP, баз данных, электронной почты и т.д. Когда Web-сервер получает документ HTML, этот документ статичен. Это означает, что HTML находится в постоянном, неизменном состоянии. С другой стороны, программа CGI выполняется в реальном времени и поэтому может выводить динамическую информацию. Способность создавать документы в реальном времени позволяет получать информацию из нескольких источников, обрабатывать ее и показывать конечному пользователю.

Слово "общий" в термине CGI относится к возможности создавать скрипты для этого интерфейса на разнообразных языках и для работы в разнообразных системах. "Шлюзовый" означает, что скрипт CGI выполняет функцию шлюза между различными приложениями для передачи информации в реальном времени. "Интерфейс" означает способность соединяться по протоколу CGI.

Сам по себе CG1 не является таинственным или неосязаемым. Это протокол, который позволяет программистам использовать различные языки для сообщения с разными системами в реальном времени с целью создания динамического содержимого для Web. Допустим, что на нашем сервере баз данных находится база данных контактов. Мы хотим иметь возможность обращаться к этой информации через Web при помощи Web-сервера. Очевидно, статичный документ HTML не может выполнить этой задачи. Для этого нужен скрипт CG1, который можно будет вызывать через Web. Такой скрипт будет запускаться Web-сервером и устанавливать подключение к серверу базы данных. После чего скрипт должен отыскать нужные данные и возвратить результаты Web-клиенту. Данный скрипт будет играть роль шлюза между Web-сервером и сервером базы данных.

Это простой пример использования CGI. По сути, нет никакого предела в применении скриптов CGI. CGI служит для сопряжения с базами данных, графическими генераторами, патентованными программами, программами с открытым кодом, функциями операционной системы и т.д.

## Чем Perl хорош для CGI

Сегодня в мире наблюдается возрастающая тенденция к преобразованию старых приложений в Web-версии и созданию новых приложений на базе Web, в противоположность прежним программам, которые для применения должны были устанавливаться на компьютере пользователя. Возможность иметь доступ к приложению через Web — простой способ гарантировать, что заказчики и служащие могут обращаться к нему отовсюду, где только есть подключение к Internet и Web-броузер. Это желание иметь Web-доступ к приложению вызывает необходимость CG1. Принять решение о языке, на котором будут написаны эти приложения CGI, иногда бывает трудно.

Существует общепринятое неправильное представление, что Perl и CGI — это одно и то же. Perl — не CGI, а популярный и простой способ написания скриптов CGI. Perl ни в коем случае не является "первым и единственным" языком для создания скриптов CGI, но это жизненно важный инструмент, который необходимо иметь в запасе, когда встает

[4] *Перед этим полезно, но не обязательно прочесть архив списка рассылки, а также страницы руководства* `perlguts` *список часто задаваемых вопросов p5p.*

задача написания скриптов. Другие языки, такие как PHP, ASP, Tcl и Cold Fusion, также могут применяться для создания скриптов CGI. Однако у Perl есть преимущества над ними, когда речь идет об использовании как для CGI, так и для общих задач.

Во-первых, Perl поддерживается на разных платформах. Можно писать скрипты на Perl для Win32, нескольких платформ Unix, MacOS, BeOS, VMS и различных других платформ[5]. Поскольку программы на Perl не компилируются, нет необходимости заново компилировать их для каждой платформы, на которой они должны работать.

Perl, в отличие от некоторых других языков, обладает такими возможностями, как внутренний отладчик, средства для работы с базами данных, сетевые функции и, возможно, лучший механизм обработки регулярных выражений. Эти и другие особенности делают Perl оптимальным языком для разработки скриптов CGI.

Срок разработки при использовании Perl очень уменьшается. В некоторых языках требуется множество строк кода, чтобы выполнить то, для чего Perl достаточно лишь нескольких строк. Это означает увеличение доступного времени для разработки новых скриптов и уменьшение объема кода, который придется сопровождать и отлаживать.

Perl хорош и тогда, когда проект включает не только скрипты CGI, но и отдельные программы обработки данных. Это относится к наиболее крупномасштабным проектам. Предположим, что в проекте требуется обновлять через регулярные интервалы определенное количество баз данных, выкладывать информацию из баз данных в Web и создавать настраиваемые пользователем отчеты на основе этих данных. Perl может решить все эти задачи, тогда как, например, PHP может обработать лишь ту часть, в которой данные отображаются в Web-клиенте. Это означает, что для выполнения проекта потребуется несколько языков. Если использовать для всех задач один язык, разработчики смогут работать совместно, время разработки уменьшится, и каждый, кто знает Perl, сможет сопровождать любую часть кода (при условии, что код хорошо написан).

Существуют и другие причины, по которым Perl является наилучшим выбором как для CGI, так и для общего применения, но Perl — лишь одно из многих инструментальных средств. Главная цель проекта заключается в том, чтобы выполнить его качественно и с учетом ограничений бюджета или времени. Perl не может быть лучшим вариантом для всех проектов, но это достаточно ценный инструмент, чтобы иметь его в запасе. Даже если вы не используете Perl для всех проектов, он всегда будет реальным вариантом, на который можно рассчитывать при планировании проекта.

# Об этой книге

Назначение этой книги — показать читателю, как использовать Perl для приложений CGI. Она демонстрирует широкий диапазон областей применения Perl в CGI, например, для работы с базами данных, обработки форм, XML, манипулирования изображениями, работы с POP и SMTP, `mod_perl` и других функций, которые необходимы в большинстве Web-приложений. В результате рассмотрения всех этих концепций читатель получает массу вариантов применения, что позволяет написать почти любое приложение CG1 с помощью приведенных средств и методов.

Примеры в этой книге представляют собой каркасы приложений CGI, написанных на Perl. Хотя вы можете непосредственно использовать их код, во многих примерах сознательно исключены некоторые функции. В конце каждой главы помещены упражнения, в которых читателю предлагается добавить эти функции самостоятельно. Мы не даем "ответов" на упражнения, поскольку все, что нужно для их выполнения, можно найти в этой книге и иногда в документации по Perl или некоторым используемым модулям. Наша цель — создать среду для интерактивного обучения с исполь-

[5] *Список платформ и советы по переносу приложений между ними можно найти в документе perlport.*

зованием этой книги, документации по Perl и модулям и ресурсов из Web. Наша задача — не просто дать вам полнофункциональные приложения, которые вы могли бы переписать на свой Web-сервер, а скорее способствовать тому, чтобы вы научились использовать Perl для CGI, и показать, какие доступные ресурсы помогут вам в этом.

Как можно видеть на обложке, у этой книги два автора. Поэтому вы можете заметить некоторые различия в стиле и методике. Так как отличительная особенность Perl — возможность сделать одно и то же несколькими способами, представление разных авторских методов, а также сравнение с вашими собственными должно постоянно приводить вас к открытиям: "Я никогда не думал, что это можно сделать и *так*!". Мы постоянно пытались использовать в коде общепринятые выражения и (конечно) не делать ошибок. Код в некоторой степени ориентирован на Unix и предназначен для работы на Web-сервере Apache. Изменения, которые следует внести в примеры, чтобы они работали в системах Win32, отмечаются, однако во всех случаях рекомендуется прочесть страницу руководства perlport, которая поставляется вместе с Perl. Поскольку сам Perl, вероятно, будет применяться в вашей системе, большая часть приложений должна работать на Web-серверах, отличных от Apache. Исключения составляют примеры, в которых сервером приложений служит `mod_perl`. Но если ваш Web-сервер должным образом сконфигурирован для выполнения скриптов CGI (обратитесь к документации по серверу), примеры должны прекрасно работать в вашей системе. Все примеры были проверены на операционных системах FreeBSD 4.0 и Red Hat Linux 6.x, а также Apache версий 1.2.x и 1.3.x с использованием версий Perl 5.005 и 5.6 (наиболее новых на момент написания этой книги). Все модули, используемые в примерах — самые последние доступные на CPAN в это время. Мы настоятельно рекомендуем вам ознакомиться с документацией по этим модулям, чтобы узнать о новых методах, а также о возможных различных способах использования старых методов.

## Для кого предназначена эта книга

Эта книга — для тех, кто еще не начал или только что начал писать приложения CGI на Perl. Даже если у вас есть опыт создания приложений CGI на Perl, эта книга будет полезной для вас, так как в ней представлены методы и инструментальные средства, позволяющие быстро и легко выполнять проекты. Книга охватывает широкий диапазон вопросов, и, возможно, с отдельными из них вы еще не сталкивались. Если же у вас минимальный или нулевой опыт программирования CGI на Perl, она научит вас многим вещам.

Эта книга также предназначена для людей, которые изучают Perl на практике, используя его для CGI. Действительно, после вспышки популярности Web и при большой потребности в приложениях CGI многие изучают Perl путем применения его для CGI, а не на приложениях системного администрирования. Книга — не учебник по Perl, но она расскажет вам многое о Perl и о том, насколько этот язык полезен для CGI. Эта книга предназначена и для тех, которые желают убедиться, насколько прост Perl в написании приложений CGI.

Мы уже рекомендовали вам ознакомиться с другими формами документации надеемся, что у вас есть документация как по Perl, так и по модулям, — это поможет вам понять все концепции этой книги и полные функциональные возможности модулей. Но не волнуйтесь. Вам не придется все время держать эту книгу на коленях, сидя перед компьютером. Мы лишь предлагаем, чтобы вы отметили страницы, относящиеся к разделам, которыми вы бы хотели более подробно заняться позже.

Наконец, эта книга предназначена для тех, кто по-настоящему хочет учиться. Вовсе не для того, чтобы переписать полнофункциональные приложения на свой сервер и использовать их, не вникая. Мы написали эту книгу для тех читателей, которые действительно хотят стать программистами на Perl и научиться создавать правильные, безопасные, качественные Web-приложения.

# Соглашения, используемые в этой книге

В этой книге применяются некоторые соглашения, призванные помочь вам определять назначение текста с одного взгляда. Во-первых, все строки кода примеров нумеруются и оформляются примерно так.

```
01:  my $foo = "bar";
02:  my $bar = 2;
```

После кода приводится объяснение каждой нумерованной строки. Например, после вышеприведенного блока помешается объяснение в такой форме.

Строки 1 и 2 инициализируют две переменные. **В строке 1** переменной **$foo** присваивается значение "bar", а в **строке** 2 значение 2 сохраняется в скаляре `$baz`.

Имена всех методов, подпрограмм и функций набраны *курсивом* и сопровождаются круглыми скобками. Вот несколько примеров.

Мы хотим использовать функцию *split (),* чтобы получить значения ...

Мы хотим применить *use()* к модулю `Fcntl`, чтобы импортировать константы ...

Метод *Foo::bar ()* чрезвычайно полезен для ...

Все скрипты в этой книге написаны с использованием прагмы `strict`, а также со включенными предупреждениями. В большей части скриптов применяется также переключатель -т, который включает загрязнение (более подробно об этом в главе 2). Эта книга была написана во время выхода версии Perl 5.6, и мы понимаем, что многие разработчики в течение некоторого времени будут использовать Perl 5.005 перед переходом к версии Perl 5.6. По этой причине предупреждения включаются с использованием переключателя -w, а не прагмы `warnings`, как в Perl 5.6. Мы не включили в эту книгу ничего, что бы являлось специфичным для версии 5.6, кроме нескольких замечаний о мелких различиях между версиями 5.005 и 5.6 там, где это необходимо. Итак, если ваш интерпретатор Perl имеет версию 5.005 или 5.6, вы можете использовать эти скрипты.

# Использование perldoc

Как мы уже говорили, по ходу чтения рекомендуется ознакомиться с документацией, которая поставляется вместе с Perl. В ее состав входит описание функций Perl, а также разные списки часто задаваемых вопросов и учебники. Кроме того, после установки какого-либо модуля (как и модулей, которые устанавливаются вместе с самим Perl) вы получаете доступ к его документации. Один из самых простых способов обращения к этой документации предоставляет программа `perldoc`, поставляемая вместе с Perl. Если вы еще не знакомы с `perldoc`, ниже приведено небольшое пособие для начинающих.

Один хороший способ ознакомления с программой `perldoc` состоит в том, чтобы использовать ее для просмотра собственной документации. Для этого надо ввести в командной строке следующее;

```
$ perldoc perldoc
```

Допустим, вам встретилась в этой книге незнакомая функция Perl, например, *map ()*. В этот момент полезно вызвать `perldoc`, чтобы выяснить, что сказано в документации Perl о ней. Чтобы выяснять, какая документация касается данной функции Perl, задайте после `perldoc` переключатель -f (f— function), сопровождаемый именем функции. Результат выполнения этой команды для функции *map ()* показан в листинге 1.1.

**Листинг 1.1, Вывод по команде perldoc -f map**

```
# perldoc -f map
=item map BLOCK LIST
=item map EXPR,LIST
Определяет BLOCK или EXPR для каждого элемента LIST (с
локальной установкой C<$_> для каждого элемента) и
возвращает значение списка, составленное из результатов
каждого такого определения. Определяет BLOCK или EXPR
в контексте списка, так что каждый элемент LIST может
вызвать один, несколько или ни одного элемента в
возвращаемом значении. В контексте скаляра возвращает
общее количество сгенерированных таким образом
элементов.
  @chars = map (chr, @nums);
Переводит список чисел в соответствующие символы.
  %hash = map { getkey($_) => $_ } @array;
- более экономичный способ записи следующего кода
  %hash = ();
  foreach $_ (@array) {
    $hash{getkey($_)} = $_;
  }
Заметьте, что, поскольку C<$_> - ссылка на значение
списка, эту функцию можно использовать для изменения
элементов массива. Хотя это полезно и поддерживается,
но может вызвать странные результаты, если LIST - не
именованный массив.
Применение для этой цели обычного цикла C<foreach> в
большинстве случаев дает более ясный код. См. также
L</grep> для массива, составленного из тех
элементов исходного списка, для которых BLOCK или EXPR
определено как истинное значение.
```

Затем предположим, что вам необходимо больше узнать о прагме `strict`, об использовании которой мы постоянно будем вам сообщать. Чтобы просмотреть документацию по любой прагме или установленному модулю, просто введите его имя после команды `perldoc` команды, как показано в листинге 1.2.

**Листинг 1.2. Вывод по команде perldoc strict**

```
# perldoc strict
strict(3) Документация Perl strict (3)
НАЗВАНИЕ
strict - прагма Perl для ограничения опасных конструкций
СИНТАКСИС
use strict;
use strict "vars";
use strict "refs";
use strict "subs";
use strict;
no strict "vars";
ОПИСАНИЕ
Если не имеется списка импорта, принимаются все
возможные ограничения. (Это самый безопасный
режим для постоянной работы, но иногда слишком строгий
для нерегулярного программирования.) В настоящее время
```

```
существует три возможных области ограничения: "subs",
"vars" и "refs".
И т.д. ...
```

Также, вероятно, будут возникать ситуации, в которых надо будет просмотреть исходный код модуля. Было бы неудобно вручную искать исходный код в системе и затем просматривать его. Программа perldoc позволяет сделать это автоматически с помощью переключателя -m (m — module), после которого указывается имя модуля. Например, чтобы узнать, как реализован модуль CGI::Carp, можно ввести следующую команду.

```
$ perldoc -m CGI::Carp
```

## Примечание

Если модуль использует отдельный файл .pod, переключатель -ra может не вывести исходный код.

Также могут встретиться ситуации, когда вам надо будет больше узнать о какой-либо концепции, а не о модуле или функции Perl, и выяснить, освещается ли она в каком-нибудь из списков часто задаваемых вопросов (FAQ), которые поставляются вместе с Perl. Предположим, например, что вас интересует, что в FAQ говорится об окружении. Переключатель -q (q — question), сопровождаемый ключевым словом[6], заставляет perldoc искать в списках FAQ вопросы, соответствующие этому ключевому слову. Если такое соответствие (или несколько соответствий) найдено, программа показывает, в каких конкретных вопросах оно встречается, каков источник этого списка FAQ, а также ответы. Для изучения Perl это поистине огромное подспорье. Результаты поиска по ключевому слову environment (окружение) показаны в листинге 1.3.

**ЛИСТИНГ 1.3. ВЫВОД ПО команде `perldoc -q environment`**

```
# perldoc -q environment
=head1 Найдено в /usr/lib/perl5/5.00503/pod/perlfaq8.pod
=head2 Я {изменил каталог, изменил окружение} в скрипте
perl. Почему это изменение исчезло, когда я вышел из
скрипта? Как мне сделать так, чтобы изменения были видимы?
=over 4
=item Unix
В самом строгом смысле этого сделать нельзя - скрипт
и оболочка, из которой он был запущен, выполняются как
разные процессы. Изменения в процессе не отражаются в
его родительском процессе, а лишь в дочерних, созданных
после изменения. Существует способ обойти этот закон,
применив в оболочке к выводу скрипта функцию eval();
подробнее см. в FAQ comp.unix.questions.
=back
```

В состав поставки Perl также входят другие документы, такие как perlrun, perlfunc и perlop (полный список приведен в приложении E), которые могут быть чрезвычайно полезны для изучения и решения задач. Эти документы можно прочитать с помощью программы perldoc. Для этого надо указать после команды perldoc название документа.

```
$ perldoc perlre
```

[6] *Параметр может быть регулярным выражением. Более подробную информацию можно получить по команде* `perldoc perldoc`.

Данная команда выводит страницу руководства perlre, которая дает возможность больше узнать о регулярных выражениях в Perl. Команда perldoc имеет гораздо больше возможностей, но те из них, которые мы только что продемонстрировали, используются наиболее часто. В приложении E перечислена основная документация, поставляемая с Perl[7].

# Использование CPAN

В этой книге используется много различных модулей. Некоторые из них, как например, CGI.pm и различные прагмы, входят в состав поставки Perl, однако многие, такие как Untaint.pm и GD.PM, отсутствуют. Получить модуль Perl можно в CPAN[8] (Comprehensive Perl Archive Network — Общая сеть архивов Perl). CPAN — это архив модулей Perl, написанных для множества целей энтузиастами Perl со всего мира. Вообще, это самый первый источник, куда можно обратиться, если вам нужно "что-то, что делает X" или "API для Y".

Как вы будете использовать Web-интерфейс CPAN — ваше дело. Вы можете работать с каталоговой структурой главного сервера CPAN или предпочесть более дружественный интерфейс на cpan.org. Какой бы интерфейс вы ни выбрали, CPAN — это невероятно полезное средство ознакомиться с тем, что уже существует, прежде чем изобретать велосипед. Оно удобно и в том случае, когда вы должны написать какой-нибудь скрипт и помните, что в CPAN есть модуль, который выполняет эту задачу. Web-интерфейс CPAN позволяет как загружать файлы, так и просматривать авторскую документацию по модулям. Еще раз скажем, что это очень полезный способ узнать, какими программными средствами вы можете располагать.

Другой способ работы с CPAN — применение модуля CPAN.pm, который распространяется вместе с Perl. В частности, этот модуль позволяет легко устанавливать модули, работая из оболочки[9]. Скажем, нам нужно установить модуль URI::Escape. Для этого можно, конечно, зайти на ближайший Web-сайт CPAN и загрузить модуль оттуда. После чего остается распаковать его и установить с помощью make. Но можно и вызвать модуль CPAN.pm, который выполнит всю работу по одной команде. Для этого достаточно ввести следующую строку.

```
$ perl -MCPAN -e 'install URI::Escape'
```

Эта команда загружает модуль CPAN.pm с переключателем -M и вызывает метод *install ()*, в параметре которого указан устанавливаемый модуль — в данном случае URI::Escape. Если вы применяете модуль CPAN.pm впервые, вам будут заданы некоторые вопросы, в частности о размере кэша, расположении программы tar, нужен ли прокси-сервер, и другие данные конфигурации, которые должен иметь CPAN.pm для извлечения и установки модулей. В основном подходят параметры по умолчанию, но не мешает обращать внимание на содержание вопросов, чтобы удостовериться, что все правильно. Если модуль уже был сконфигурирован, после ввода этой команды немедленно начинается загрузка модуля URI::Escape из Internet и установка.

При установке URI::Escape можно обратить внимание на один факт: этот модуль имеет модуль зависимости MIME-Base64. В большинстве случаев в ходе установки обнаруживаются такие зависимости, из-за чего процесс приходится прерывать, устанавливать требуемый модуль и начинать установку первого модуля заново. При ис-

---

[7] *Perl 5.6.*

[8] *На http://www.perl.com/CPAN-local/SITES.html можно найти глобальный список зеркальных серверов, а на http://www.cpan.org — дружественный поисковый интерфейс.*

[9] *Пользователи Win32 могут работать с CPAN.pm, но могут предпочесть диспетчер пакетов Perl (Perl Package Manager — PPM), который выполняет ту же задачу установки модулей для Win32.*

пользовании CPAN.pm этот модуль ищет в устанавливаемом модуле зависимости, автоматически устанавливает их и завершает работу без перерыва. Так, при установке URI::Escape модуль CPAN найдет зависимость от MIME::Base64, проверит, имеется ли этот модуль и его требуемая версия, и установит его в случае отсутствия.

Часто задают вопрос: "Как можно узнать, какие модули уже установлены?" Конечно, ответ можно получить, написав небольшой скрипт, который будет проверять каталоги в @INC И искать файлы с расширением .pm. Однако модуль CPAN.pm также предоставляет возможность сделать это. Приведенная ниже команда создает файл со списком всех установленных модулей и номерами их версий[10].

```
$ perl -MCPAN -e awutobundle
```

Эта команда производит два действия. Сначала она отображает список установленных модулей с номерами установленных версий, а также последних версий, которые доступны в CPAN, и где в CPAN их можно найти. По умолчанию эти данные не сохраняются в файле[11]. В файл записываются только имена установленных в данный момент модулей и номера их версий. Такая информация удобна, когда требуется сравнить имеющиеся модули с тем, что можно получить в CPAN. Файл списка модулей сохраняется не в текущем рабочем каталоге, а в подкаталоге Bundle основного каталога CPAN (по умолчанию $HOME/.cpan).

Модуль CPAN может применяться и в режиме оболочки. Такой режим пригоден для установки модулей, но лучше всего искать в нем информацию о модулях, авторах и пучках. Это может быть полезно при попытке найти данные об авторе модуля, а также о том, какие доступны модули и пучки, соответствующие заданным критериям. Мы покажем примеры каждой из этих областей применения, но сначала вы должны научиться входить в оболочку CPAN.

```
S perl -MCPAN -e shell
```

Эта команда переводит CPAN в режим оболочки. Сейчас мы попробуем найти данные об авторе. В режиме оболочки можно производить поиск авторов, модулей и пучков. Область поиска обозначается ключевым словом (a — авторы, m — модули, b — пучки), после которого указывается критерий поиска.

Критерий поиска может быть регулярным словом или регулярным выражением. Если указано регулярное слово, будут найдены данные, содержащие это слово. Поиск по регулярному выражению происходит иначе. Это лучше всего объяснить на примере (см. листинг 1.4).

### Листинг 1.4. Поиск автора с помощью идентификатора CPAN

```
cpan> a KMELTZ
Author id = KMELTZ
  EMAIL perlguy@perlguy.com
  FULLNAME Kevin Meltzer
```

Как можно видеть из этого листинга, поиск по указанному идентификатору CPAN выдает имя и адрес электронной почты соответствующего автора. Это хорошо, если вам известны идентификаторы CPAN авторов, а если нет? Казалось бы, листинг 1.5 выдаст похожие результаты, но это не так.

### Листинг 1.5. Поиск автора с помощью частично заданного идентификатора CPAN

```
cpan> a KM
Для параметра KM не найдено объектов типа Author
```

[10] *Если в модуле нет переменной $VERSION, вместо версии будет выдано 'undef'.*

[11] *Можно сохранить их, если переадресовать стандартный вывод команды в файл.*

Очевидно, от этого способа мало толку. Поэтому следует применить регулярное выражение. Результат показан в листинге 1.6,

### Листинг 1.6. Поиск с помощью частично заданного идентификатора CPAN в регулярном выражении

```
cpan> /KM/
Author  KMACLEOD  (Ken MacLeod)
Author  KMELTZ  (Kevin Meltzer)
Author  MARKM  (Mark Mielke)
Author  STAS  (Stas Bekman)
Author  TKML  (The Tk Perl Mailing list)
```

Как можно видеть, в листинге 1.6 выдаются все авторы, идентификатор CPAN которых содержит текст "KM". Теперь посмотрим, как искать модули по этому способу. При поиске модуля без регулярного выражения CPAN возвращает информацию об этом конкретном модуле, как показано в листинге 1.7

### Листинг 1.7. Поиск информации о модуле

```
cpan> m URI::Escape
Module id = URI::Escape
  DESCRIPTION
  General URI escaping/unescaping functions
  CPAN_USERID  GAAS (Gisle Aas <gisle@aas. no>)
  CPAN_VERSION  3.13
  CPAN_FILE    G/GA/GAAS/URI-1.07.tar.gz
  DSLI_STATUS  Rmpf  (released,mailing-list,perl,functons)
  MANPAGE
  URI::Escape - Escape and unescape unsafe characters
  INST_FILE  /usr/lib/perl5/site_perl/5.005/URI/Escape.pm
  INST_VERSION 3.13
```

На этот раз мы используем опцию m, чтобы искать модули, а не опцию a, применяемую для поиска авторов. В результате выводится информация о модуле, включая описание, автора, номер последней доступной на CPAN версии, где располагается файл модуля, если он установлен, и номер установленной версии. Это еще один удобный способ выяснить, можно ли обновить установленный модуль на более новую версию. Теперь допустим, что нам надо найти в CPAN все модули по определенной теме. Это может быть CGI, Apache, XML или что угодно. Например, для проекта, в котором используется XSLT, надо узнать, какие модули, содержащие слово XSLT, имеются в CPAN. Как это сделать, показано в листинге 1.8.

### Листинг 1.8. Поиск всех модулей, содержащих "XSLT"

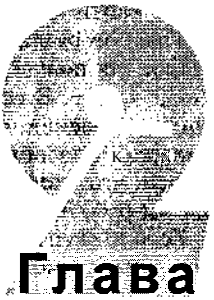
```
cpan> ra /XSLT/
Module  Apache::AxKit::Language::NotXSLT  (M/MS/MSERGEANT/AxKit-
0.67.tar.gz)
Module  Apache::AxKit::Language::XSLT  (M/MS/MSERGEANT/AxKit-
0.95.tar.gz)
Module  XML::EP::Processor::XSLT  (J/JW/JWIED/XML-EP-0.01.tar.gz)
Module  XML::EP::Processor::XSLTParser  (J/JW/JWIED/XML-
EP-0.01.tar.gz)
Module  XML::XSLT   (B/BR/BRONG/XML-XSLT-0.24.tar.gz)
Module  XSLT       (J/JO/JOSTEN/XML-XSLT-0.20.tar.gz)
Module  XSLTParser   (J/JO/JOSTEN/xslt-parser-0.13.tar.gz)
```

Мы получили список доступных модулей, которые можно использовать для обработки XSLT. Последний способ поиска — по пучкам. *Пучок* (bundle) — это группа модулей, распространяемых вместе. Например, существует много модулей для работы с LDAP, и эти модули объединяются в пучки. Чтобы найти доступные модули Perl для LDAP, можно задать следующую команду поиска пучков.

**Листинг 1.9. Поиск пучка для LDAP**

```
cpan > b /ldap/
Bundle Bundle::NetLDAP (G/GB/GBARR/perl-ldap-0.19.tar.gz)
Bundle Bundle::Wizard::LDAP (J/JW/JWIED/Wizard-LDAP-
0.1008.tar.gz)
```

Мы посчитали, что будет важно показать вам некоторые сферы применения модуля `CPAN.pm` и самого CPAN в начале этой книги. Это невероятно полезный ресурс, который может облегчить установку модулей для выполнения примеров этой книги.




# Что вы должны знать

## Предпосылки

Мы, как авторы, предъявляем к читателям некоторые требования. Эти требования и ожидания нельзя назвать странными или необоснованными; они охватывают много базовых тем, но затрагивают их лишь кратко. Эта книга научит вас создавать на Perl работоспособные и эффективные приложения CGI, используя как те знания, которые уже есть у вас, так и те, которые вы получите здесь. В отличие от некоторых других книг, наша цель состоит не в том, чтобы показать читателю законченные полнофункциональные приложения CG1 и коротко объяснить, как их использовать. Напротив, в нашей книге представлены рабочие "каркасы" приложений, пригодные для использования, безопасные и демонстрирующие хорошие приемы программирования. (ЭМСНС — это можно сделать несколькими способами.)

Хотя, чтобы работать с данной книгой, вы не обязаны быть профессионалом в Perl или даже иметь в нем глубокие знания, вы должны иметь некоторое представление о Perl. Фактически, чтобы применять Perl, хорошо знать его не обязательно. Эта книга охватывает программирование на Perl в несколько расширенном объеме, но она не ориентирована исключительно на опытного программиста. В ней не излагаются основные концепции Perl, такие как структуры и типы данных, вывод на печать или открытие файла. В то же время здесь рассматриваются такие вопросы, как `map()`, DBI, system `()` и другая информация, с которой, возможно, некоторым программистам уже приходилось сталкиваться. Поэтому мы считаем, что у читателя есть базовые знания языка Perl или, по крайней мере, простых программных конструкций.

Вы также должны иметь Web-сервер, чтобы работать на нем. Все скрипты и приложения в этой книге были написаны и проверены на Web-сервере Apache. Однако, большая часть действий выполняется точно также и на многих других Web-серверах,

например, IIS и Netscape FastTrack[1]. На протяжении этой книги мы будем указывать на различия между исходным Perl и Perl для Win32, Самую новую информацию о различиях можно найти на странице руководства perlport и примечаниях к дистрибутивам Perl. Главное различие, о котором следует помнить, касается системных вызовов. Например, для получения списка каталога в UNIX надо вызвать функцию ls, а в системе Win32 — dir. В соответствующих местах мы будем обращать ваше внимание на необходимые изменения в примерах, вызванные этими различиями операционных систем, а также на способы повышения переносимости.

Мы ожидаем, что у вас уже установлен Perl, желательно на той же машине, что и Web-сервер. Если вы работаете в системе UNIX или подобной, весьма вероятно, что Perl уже установлен. Если вы используете Windows 9x/NT/2000, вам придется сделать это самостоятельно. Чтобы узнать, имеется ли в системе Perl, после приглашения оболочки введите:

```
perl -v
```

Если Perl установлен в системе, вы должны получить примерно следующий вывод.

**Листинг 2.1. Вывод версии Perl**

```
This is perl, version 5.005_04 built for i386-freebsd
Copyright 1987-1999, Larry Wall
Perl может быть скопирован только согласно условиям
Творческой лицензии либо Общедоступной лицензии GNU,
которую можно найти в наборе исходных текстов Perl 5.0.
Полную документацию по Perl, включая списки FAQ, можно
получить в этой системе, введя 'man perl' или 'perldoc
perl'. Если вы имеете доступ к Internet, направьте свой
броузер на начальную страницу Perl по адресу
http://www.perl.com/.
```

Если такое сообщение не появится, вы, возможно, должны будете установить Perl самостоятельно. Все скрипты в этой книге написаны и проверены с использованием, по крайней мере, версии 5.005_03. Если в вашей системе установлена более старая версия Perl, вам понадобится обновить ее. Самые последние версии Perl можно получить на http://www.perl.com или на http://www.activestate.com соответственно для стандартного Perl или Perl для Win32. Если вы должны или хотите обновить вашу версию Perl, загрузите самый новый дистрибутив для вашего типа операционной системы и следуйте инструкции по установке. Мы не приводим здесь команды установки Perl, так как существуют различные варианты установки, и будет лучше, если вы прочтете документацию и установите Perl именно там и так, как пожелаете.

Если вы не создаете приложения CGI непосредственно на Web-сервере, у васдополнительно должна быть какая-нибудь хорошо знкомая вам программа FTP. В этом случае необходимо знать, как запустить команду chmod (при использовании сервера типа UNIX), чтобы должным образом изменять разрешения для скриптов CGI. Следует также удостовериться, что при загрузке скриптов Perl на сервер установлен режим ASCII, а не двоичный. Загрузка скрипта как двоичного файла вызовет ошибку при попытке его выполнения.

Данная книга написана в предположении, что вы уже прочли значительную часть документации по Perl и используемому вами Web-серверу. Но если это не так, не волнуйтесь. Не стоит немедленно закрывать книгу и штудировать всю документацию по Perl и Web-серверу. Однако нужно знать, где такая документация находится, чтобы при

[1] *Серверы Netscape можно найти на http://www.netscape.com, а Web-серверы Microsoft — на http://www.microsoft.com. Web-сервер Apache для систем Win32 и UNIX доступен на http//www. apache.org.*

необходимости справиться в ней. Например, если на вашем Web-сервере нет каталога с поддержкой CGI, вы должны обратиться к документации по нему, чтобы научиться включать поддержку CGI для каталога. На сервере Apache подкаталог cgi-bin корневого каталога имеет поддержку CGI по умолчанию. Включить поддержку CGI для других каталогов можно, добавив в соответствующее место файла httpd.conf строку

```
Options +ExecCGI
```

Для повторения примеров этой книга можно создать в корневом каталоге Web-сервера каталог под названием cgibook и добавить в файл httpd. conf следующие строки.

Листинг 2.2. Пример включения поддержки CGI в h t t p d . c o n f

```
<Directory "/usr/local/apache/htdocs/cgibook">
  Options Indexes FollowSymLinks ExecCGI Includes
  AllowOverride All
  Order allow,deny
  Allow from all
</Directory>
```

Этот элемент конфигурации позволяет каталогу cgibook запускать скрипты CGI и вставки на стороны сервера (Server Side Includes). Он также показывает содержимое каталога и предусматривает следование по символьным связям. Возможно, вам понадобится изменить путь к каталогу, показанный в листинге 2.2, если корневой каталог вашего сервера — не /usr/local/apache/htdocs/cgibook. Вам также следует убедиться, что конфигурация корневого каталога сервера разрешает изменения (AllowOverride). В конфигурации Apache по умолчанию установлены максимальные ограничения, так что в httpd.conf будет установлено следующее:

```
AllowOverrides None
```

Это означает, что подкаталоги сервера не могут изменять конфигурацию параметров, установленную в корневом каталоге. Если корневой каталог не позволяет подкаталогам использовать необходимые параметры (чего он не делает по умолчанию), эту директиву следует заменить на следующую.

```
AllowOverride All
```

После внесения этих изменений в файл httpd.conf сервер следует перезапустить, чтобы они вступили в силу. Эти директивы также можно поместить в файл .htaccess в каталоге cgibook.

# Редакторы

В основном вы можете использовать свой любимый текстовый редактор (нет, Microsoft® Word™ —- это *не* текстовый редактор!). В системах типа UNIX имеется несколько популярных редакторов — vi, vim, emacs, pico и ed. В системах Win32 эту задачу прекрасно выполняет WordPad. Он может работать и с файлами, полученными из UN!X и содержащими разделители строк стандарта UNIX. Многие используют Notepad, но этот редактор не читает разделители строк UNIX и отображает их как "кракозябры". При работе в среде нескольких платформ целесообразно применять редактор, который может "понимать" форматы других операционных систем. Существуют также иные редакторы для Win32, перенесенные из UNIX, как, например, vim и emacs, а также различные интегрированные среды разработки Perl. Используйте любой текстовый редактор, который удобен для вас, и даже испытайте один-два новых!

# Разрешения файлов

В системах Win32 необходимые для скриптов CGI разрешения можно установить, сделав эти файлы выполняемыми. Лучше всего выяснить, как это можно сделать в Win32, обратившись к документации по Web-серверу. Но в системах типа UNIX более важно иметь представление о разрешениях на доступ к файлам UNIX. Для скриптов CGI применяются стандартные разрешения 755. Это означает, что файл могут читать и выполнять все, а также то, что владельцу предоставлено разрешение на чтение/запись. Для тех, кто незнаком с разрешениями файлов UNIX, ниже приведен краткий обзор этого вопроса.

Для каждого файла можно установить три бита разрешения: бит чтения, бит записи и бит выполнения. Существует также три вида пользователей, для которых устанавливаются разрешения: владелец, группа и все. Первые три бита разрешения устанавливаются для разрешений владельца, следующие три — для группы и последние три — для всех. Просмотрев содержимое каталога, в котором имеется скрипт Perl/CGI, можно заметить строку наподобие следующей:

```
- rwxr-xr-x 1 kevin users 1736 Oct 29 10:29 my.cgi
```

Здесь можно видеть, что файл my.cgi имеет разрешения 755. Как узнать, что `rwxr-wr-w` — это 755? Каждый из трех битов разрешения имеет числовой эквивалент. Эти эквиваленты показаны в листинге 2.3

**Листинг 2.3. Разрешения файлов UNIX**

```
r  4
w  2
x  1
```

Отсюда ясно, что первая тройка разрешений rwx эквивалентна 7, вторая, `r-x`, — 5 и третья, `r-x`, — также 5. Разрешения файлов имеют значение не только для самого скрипта CGI, но и для всех файлов, в которые скрипт может производить запись. Наиболее простой пример — текстовый файл, в который пишет скрипт CGI. Учитывая, что скрипт должен будет как читать, так и записывать в этот файл, его разрешения должны быть `-rw-rw-rw`, или 666. Иногда бывает, что программисты дают скриптам CGI разрешения на запись в файлы, имеющие разрешения 777, думая, что это делать безопасно, так как в файле нет никакого исполняемого кода. Это неправильное представление, которое может вызвать опасные последствия. Если этот файл окажется в каталоге, в котором Web-серверы разрешают выполнение скриптов, он может быть запущен на выполнение. В большинстве случаев выполнение файла, содержащего случайные данные, не грозит никакой опасностью. Однако, если кто-то запишет в этот файл скрипт на командном языке оболочки, возникнет серьезная неприятность. Мораль: следует быть осторожным с разрешениями файлов и всегда назначать минимальное количество необходимых разрешений.

Мы рассказали об основах разрешений файлов UNIX и их отношении к CGI. Более подробную информацию о разрешениях файлов и команде chrr.od можно получить на странице руководства UNIX по `chmod`.

# Основные принципы безопасности

В отношении CGI существуют некоторые проблемы безопасности, которые касаются не столько CGI, сколько безопасности Web-сервера или сети (например, пароли, разрешения, брандмауэры и т.д.). Немногие связаны со скриптами CGI. По сути, если сервер обслуживает опытный системный администратор, найдется немного причин

опасаться скриптов CGI. С точки зрения программиста, одна из главных проблем заключается в использовании непроверенных данных потенциально опасным способом (например, помещении их в файл с разрешениями 777). Что такое непроверенные данные? Просто любые данные, исходящие извне вашей программы. Это могут быть параметры, переданные в командной строке или из других скриптов, а также, что вызывает наибольшие опасения, — данные, введенные в скрипт пользователем, например, через Web-форму. Если данные переданы в скрипт CG1 из неизвестного или непроверенного источника, они считаются "загрязненными". Пользователь всегда считается непроверенным источником. Большей частью, если только переменной не присваивается явно указанное в скрипте значение или другая переменная, значение которой взято из скрипта, данные считаются загрязненными. Этот принцип очень полезен, так как он предупреждает программиста о данных, использование которых определенными способами может быть опасно. Однако информация о том, что загрязнено и что безопасно, по умолчанию не дана.

# Использование переключателя -T

Переключатель -T заставляет интерпретатор Perl выполнять "проверку данных на загрязнение", переводя его в режим проверки загрязнения. Обычно это производится по умолчанию, если скрипты запускаются в режиме setuid или setgid. В противном случае нужно явно указать этот переключатель. В режиме -т интерпретатор Perl проверяет, не пытается ли скрипт использовать данные, полученные из внешнего источника, чтобы повлиять еще на что-то вне скрипта. Он проверяет, например, не вызывает ли скрипт системную команду с данными, представленными пользователем через Web-форму, если только данные не были очищены. Все данные, полученные через параметры командной строки, из файлов, из различных системных функций и переменных окружения, считаются загрязненными. Эти загрязненные данные не могут применяться в командах, изменяющих файлы, каталоги, процессы, и командах подоболочки. Единственное исключение — когда список параметров передается в функции *exec()* или *system()*, элементы этого списка не проверяются на загрязнение по отдельности. Если один элемент в списке загрязнен, весь список считается загрязненным — так сказать, за соучастие. Кроме того, переменная считается загрязненной, если присвоить ей значение другого загрязненного выражения. Рассмотрим некоторые примеры загрязненных данных в листинге 2.4.

### Листинг 2.4. Примеры загрязненных данных

| | |
|---|---|
| `$foo = $ARGV[0];` | `$foo` загрязнено. |
| `Sbar = Sfoo;` | `$bar` загрязнено, так как получает значение от загрязненного `Sfoo`. |
| `$file = <FOO>;` | Загрязнено. Все, полученное через оператор угловых скобок (`<>`), `<STDIN>`, `<FILEHANDLE>` и т.д. считается загрязненным. |
| `Sfoo = "Hello";` | `Sfoo` уже не загрязнено. |
| `Spath = $ENV{'PATH'};` | `$path` загрязнено. |
| `SENV{'PATH'} = '/bin:/usr/bin'; Spath = $ENV{'PATH'};` | `Spath` не загрязнено, так как значение переменной окружения PATH устанавливается внутри скрипта. |
| `$param = param('Name');` | Sparam загрязнено. Все данные, возвращаемые методом `CGI.pm param()`, считаются загрязненными. |

Согласно этим примерам, переменные Sfoo, $file и $param загрязняются. Если попытаться использовать эти загрязненные переменные опасным способом, будет получена ошибка "Insecure dependency" (Опасная зависимость) или "Insecure $ENV{PATH}". Рассмотрим несколько примеров безопасного использования этих загрязненных данных и попробуем использовать их так, что это вызовет ошибку при заданном переключателе -т.

### Листинг 2.5. Примеры использования загрязненных данных

```
unlink $foo;              Опасно и не будет выполнено.
open(FOO, "$foo");        Все в порядке, так как файл открывается только для чтения.
open (FOO, ">$foo");      Опасно, так как файл открывается для записи.
exec "cat $foo";          Опасно, так как используется подоболочка.
exec "cat", Sfoo;         Безопасно, так как подоболочка не используется.
```

Проанализируем этот листинг, особенно в отношении последних двух строк. Почему одна из них безопасна, а другая считается опасной? В конце концов, обе они — вызовы *exec()*! Когда вызывается функция *exec()* или system *()*, она проверяет количество своих параметров. Если передается один скалярный параметр или массив с единственным элементом (как в exec "cat $foo";), функция передает команду для анализа в системную оболочку. В системе UNIX это обычно /bin/sh. Если в функцию не были переданы метасимволы оболочки, она разбивает параметр на слова и выполняет их буквально. Функция Perl *system()* действует таким же образом, с тем исключением, что она возвращает управление обратно в скрипт.

Чтобы можно было использовать данные опасным способом, они должны быть предварительно очищены. Если Perl находится в режиме проверки на загрязнение, у нас нет другого выбора, и это должно войти у программиста в привычку, особенно при работе с CGI. Во всех примерах этой книги производится проверка на загрязнение независимо от того, нужно это или нет. Мы надеемся, что, изучив и повторив эти скрипты, вы сами приобретете эту хорошую привычку и всегда будете писать безопасные скрипты Perl/CGI.

# Проверка на загрязнение и очистка данных

При желании вы можете загрузить с любого узла CPAN разработанный Томом Финиксом (Phoenix) модуль Taint.pm, который выполняет проверку на загрязнение, но эту задачу легко решить и самостоятельно. Рассмотрим метод в листинге 2.6, который возвращает 1 (истину), если какой-то из его параметров загрязнен.

### Листинг 2.6. Подпрограмма **is_tainted**()

```
01:  sub is_tainted {
02:    my $check = shift;
03:    return !eval { $check++, kill 0; 1; };
04:  }
```

**Строка 1** начинает подпрограмму is_tainted.

**Строка** 2 извлекает параметр, переданный в подпрограмму, и присваивает его значение скаляру $check. Как мы только что говорили, после присвоения загрязненных данных новой переменной эта переменная загрязняется. Поэтому, если в подпрограмму передан загрязненный скаляр, $check также будет загрязнен.

**Строка 3** возвращает инвертированное значение *eval ()*. Функция eval *()* используется потому, что таким путем скрипт не "убивается" системой, когда с загрязненными данными производятся какие-либо опасные действия. Мы можем безопасно получить возвращаемое значение и продолжать работу над ним. Эта строка проверяет данные на загрязнение довольно хитрым способом, но при этом не наносится никакого вреда. Если приращение $check выполняется в отдельной строке, это безопасно даже с загрязненными данными. Однако, когда мы добавляем команду *kil()*, это может вызвать ошибку, так как в этой же строке используются загрязненные данные. Зачем мы делаем приращение загрязненной переменной? Это способ выполнить действие с загрязненной переменной, которое Perl будет считать опасным. В данном случае приращение считается опасным, так как оно объединено с оператором *kil()*. Лишь в немногих операционных системах допускается идентификатор процесса 0 (к ним не относится большинство систем Linux и BSD), но и в них "kill 0" считается идиомой и не приводит к вызову системной функции *kill ()*. В остальных системах не бывает процессов с идентификатором 0, так что выполнение kill 0 фактически безопасно, ибо при этом не будет уничтожен ни один процесс[2]. Если данные не загрязнены, *eval()* возвращает значение 1, которое инвертируется в 0. Если *eval()* дает сбой, возвращаемый 0 инвертируется в 1. Пример использования этого метода приведен в листинге 2.7.

**Листинг 2.7. Пример проверки данных на загрязнение**

```
01: #!/usr/bin/perl -wT
02: use strict;
03: use CGI qw(:standard);
04: print header();
05: my $param = param('Name');
06: print "$param загрязнен" if is_tainted($param) ;
07: sub is_tainted {
08: my @check = @_;
09: return !eval j join('',@check), kill 0; 1; };
10: }
```

Данный пример несколько тривиален, так как мы знаем, что $param будет загрязнен, потому что его значение передается через форму, однако это хороший способ показать, как производится проверка. Теперь мы знаем, как перехватить загрязненные данные, прежде чем с ними будет выполнена опасная операция. Но как очистить данные, чтобы использовать их было безопасно? Есть только один способ очистки данных: присвоить загрязненной переменной значение подстроки регулярного выражения. Это означает присваивать значения специальным переменным $1, $2, $3 и т.д., исходя из сопоставления исходной переменной с определенным образцом. Ведь переменную нельзя очистить просто так, не проверив предварительно правильность значения. Например, мы ожидаем, что через форму будет введен какой-то телефонный номер. Он не может содержать букв, хотя допускаются тире. Поэтому мы должны проверить соответствие введенных данных одному из трех образцов.

Пример 1. 555-555-5555

Пример 2. 5555555555

Пример 3. 555 555-5555

Если в Web-форме имеются данные наподобие номеров телефонов, рекомендуется сделать так, чтобы они могли иметь только один вид, например, 555-555-5555. Это помогает удостовериться, что данные будут вводиться в одной и той же форме, пригодной для дальнейшего использования. Но для этого общего примера мы разрешим

[2] *Некоторые операционные системы не имеют команды kill.*

все формы данных. Процесс проверки на загрязнение и попытки очистить загрязненную переменную мы пока проиллюстрируем на примере скрипта командной строки. Позднее вы увидите, как это делается в скриптах CGI.

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

**Строка 1** — путь к Perl. Обратите внимание, что здесь включены предупреждения и проверка на загрязнение. Разумеется, если отключить эту проверку, от нашего примера будет мало толку!

```
02:  use strict;
```

**Строка 2** несколько увеличивает ограничения времени компиляции за счет использования модуля `strict`. Рекомендуется всегда делать это.

```
03:  my $foo = join('',@argv);
```

**Строка 3** получает параметры командной строки. Поскольку это скрипт командной строки, параметры передаются в `@ARGV`. ЭТИ параметры, в данном случае части телефонного номера, мы объединяем в строку.

```
04:  if (is_tainted($foo)) {
```

**В строке 4** начинается блок, если подпрограмма *is_tainted()* возвращает значение **истины**.

```
05:      print "\$foo загрязнен. Пытаюсь очистить\n";
```

**Строка 5** выводит сообщение, что переменная действительно загрязнена,

```
06:      my $pattern = qr (^\d{3}(-|\s+)?\d{3}(-|\s+)?\d{4}$);
```

**Строка 6** представляет образец, которому должны соответствовать данные, чтобы они были успешно очищены. Образец создается оператором *qr()*. Этот оператор, представляющий "кавычки как регулярное выражение", впервые появился в Perl версии 5.005. Одно из преимуществ использования *qr()* — некоторая оптимизация, так как этот оператор компилируется тогда, когда происходит его присвоение. Это регулярное выражение ищет строку, которая соответствует одному из указанных выше форматов телефонного номера.

```
07:      $foo = untaint($foo, $pattern);
```

**В строке 7** вызывается подпрограмма *untaint()*, которую передаются загрязненная переменная и образец, которому должна соответствовать эта переменная, для очистки. Эта подпрограмма очень полезна, так как образец, передаваемый в нее, можно использовать многократно. Значение, возвращаемое *untaint()*, присваивается переменной $foo. Если очистка проходит успешно, новое значение $foo будет чисто и не загрязнено. Если очистка дает сбой, новое значение остается загрязненным.

```
08:  } else {
09:    print "\$foo не загрязнен!!\n";
10: }
```

**Строки 8, 9 и 10** начинают другую ветвь условного выражения, если исходные данные не загрязнены. **Строка 9** выводит сообщение, что переменная не загрязнена.

```
11:  sub is_tainted {
12:    my $check = shift;
13:    return !eval {$check ++, kill 0;1;};
14: }
```

**Строки 11—14—** подпрограмма *is_tainted()*. Она повторяет подпрограмму из предыдущего примера.

```
15: sub untaint {
```

Строка 15 начинает подпрограмму *untaint ()*.

```
16:     my ($foo, $pattern) = @_;
```

**Строка** 16 присваивает входящие параметры — загрязненные данные и образец для них — соответственно переменным $foo и $pattern.

```
17:     if ($foo = ~ /($pattern)/ {
```

**Строка 17** представляет регулярное выражение с использованием полученного образца. За счет того, что образец для проверки определенной переменной перед ее очисткой передается в параметре, эта подпрограмма становится более полезной, так как ее можно использовать многократно. Если затем потребуется проверить, состоят ли данные (например, имя) только из букв, надо будет просто передать в *untaint ()* образец ^\w+$. Это хороший пример динамического программирования. Заметьте, что образец заключен в круглые скобки. Это необходимо для того, чтобы гарантировать, что в случае соответствия образцу его значение присваивается $1, что, как было сказано выше, необходимо для очистки переменной.

```
18:         $foo = $1;
```

**Строка 18** производит повторное присваивание переменной, если регулярное выражение из строки 17 соответствует ей. Здесь переменной $foo присваивается значение совпадающей подстроки, которое перед этим было сохранено в $1. Теперь данные очищены!

```
19:         print "\$foo очищен!!\n";
```

**Строка** 19 сообщает, что очистка прошла успешно.

```
20:         return $foo;
```

**Строка 20** возвращает уже не загрязненный скаляр **$foo.**

```
21:     } else {
22:       print "Нельзя очистить \$foo\n";
23:       return $foo;
24:   }
25: }
```

**Строки 21—25** выполняются, если попытка очистки терпит неудачу. Если данные не соответствуют образцу и не могут быть очишены, этот блок выводит сообщение об этом и возвращает данные без очистки.

Теперь мы проверим работу этого скрипта. Присвоим ему имя taint_ex.pl и попробуем запустить с правильными параметрами и без них. Мы получим результаты наподобие приведенных в листинге 2.8.

Листинг 2.8. Примеры использования **taint_ex.pl**

```
% ./taint_ex.pl 5555555555
$foo загрязнен. Пытаюсь очистить
$foo очищен!!
% ./taint_ex.pl 555-555-5555
$foo загрязнен. Пытаюсь очистить
$foo очищен!!
% ./taint_ex.pl Это не телефонный номер
$foo загрязнен. Пытаюсь очистить
Нельзя очистить Sfoo
% ./taint_ex.pl 555 KL5-5555
$foo загрязнен. Пытаюсь очистить
Нельзя очистить $foo
```

```
% ./taint_ex.pl 555 555-5555
$foo загрязнен. Пытаюсь очистить
$foo очищен!!
```

Как можно видеть, телефонные номера разрешенных форматов пропускаются через "химчистку" и успешно преобразуются в безопасные переменные. Те параметры, которые не соответствуют нашему образцу, не очищаются и остаются загрязненными. Возникает вопрос, надо ли всегда очищать входящие данные? И да и нет. Данные всегда будут загрязнены, но следует очищать лишь те переменные, способы применения которых могут повлиять на систему. Эти способы мы перечислили выше, но для лучшего запоминания не мешает их повторить: загрязненные данные не могут использоваться в командах, которые изменяют файлы, каталоги, процессы, или в командах подоболочки. Исключение — если список параметров передается в функции *exec()* или *system ()*, его элементы не проверяются на загрязнение. Включение проверки на зафязнение дает некоторые побочные эффекты. Один из них заключается в том, что Perl игнорирует переменные окружения PERL5OPT И PERL5LIB. Из-за этого текущий рабочий каталог (".") не включается в @INC при запуске скрипта. Это не означает, что мы не сможем применять *use()* или *require()* для модулей, находящихся в каталогах, перечисленных в @INC. Однако если какие-то модули хранятся в подкаталогах текущего рабочего каталога, придется внести изменения в @INC. Везде в этой книге @INC изменяется при помощи прагмы времени компиляции lib. Также изменения могут производиться в блоке BEGIN, чтобы продемонстрировать принцип ЭМСНС. Все скрипты CGI в этой книге составлены с проверкой на зафязнение, даже когда для этого, как кажется, нет оснований. Это сделано с целью показать преимущества ее использования и заставить вас привыкнуть к ней.

Примечание для профаммистов на C и XS: вы должны явно установить режим проверки на загрязнение переменных в суррогатах XS. Переключатель -т не влияет на XS. Запомните это, когда будете писать суррогаты XS для скриптов CGI.

# Переменная PATH и переключатель -т

В некоторых ситуациях в режиме проверки на зафязнение может быть получена ошибка "Insecure $ENV{'PATH'}". Это происходит при попытке работать с данными, даже с **незагрязненными,** с привлечением оболочки. Давайте еще раз обратимся к скрипту taint_ex.pl. Вставьте между **строками 10** и **11** следующую строку.

```
system("echo $foo");
```

Затем попробуйте снова запустить скрипт с параметрами, которые в прошлый раз были очищены успешно. Вы получите примерно следующий результат.

**Листинг 2.9. Пример очистки данных с помощью taint_ex.pl**

```
% ./taint_ex.pl 5555555555
$foo загрязнен. Пытаюсь очистить
$foo очищен!!
Insecure $ENV{PATH} while running with -T switch at
./taint_ex.pl line 15.
```

Что произошло? Ведь данные были очищены перед тем, как использовать их в *system ()*! Что не в порядке с $foo? Ответ — ничего. В данном случае дело не в переменной $foo, а в $ENV{PATH}. Как можно вспомнить, функция Perl *system()*, если вызывается таким способом, использует подоболочку. Для применения подоболочки надо

знать значение PATH — в данном случае для того, чтобы найти команду echo. Поскольку подоболочка получает значение $ENV{PATH} ИЗ окружения пользователя, эта переменная будет внешней для скрипта и режим проверки на загрязнение не будет доверять ей. Выход— присвоить $ENV{PATH} какое-нибудь известное и доверяемое значение. Но прежде чем мы устраним эту проблему, давайте исследуем один способ, который, похоже, решает дело без обращения к $ENV{PATH}. Кажется целесообразным указать для команды полный путь, в результате чего предыдущая строка примет такой вид.

```
system("/bin/echo $foo");
```

Но, если попробовать запустить скрипт, появится та же ошибка. Почему? Perl не знает, вызывает ли эта команда другую команду, которая зависит от PATH. Поэтому он не позволяет вызвать команду, которая может вызвать другую, которая, в свою очередь, может сделать что-то рискованное. Следовательно, **Perl** требует задать в скрипте переменную окружения $ENV{PATH}, чтобы удостовериться, что она имеет известное и доверяемое значение. Снова вернемся к примеру и изменим строку таким образом.

```
system "/bin/echo", $foo;
```

Теперь команда echo сработает.

```
% ./taint_ex.pl 5555555555
$foo загрязнен. Пытаюсь очистить
$foo очищен!!
5555555555
```

Ручаемся, что сейчас вы скажете: "Почему вы сразу не объяснили все о $ENV{PATH} и не решили проблему так, как она была поставлена в самом начале?" Мы поступили так по двум причинам. Во-первых, продемонстрировали безопасный способ использования *system ()*. Во всех случаях более безопасно передавать в функцию типа system *()* и *exec()* список параметров, а не скаляр. Во вторых, присвоение значения $ENV{PATH} исправляет первоначальную ошибку, как мы только что видели, но не дает желаемых и, возможно, ожидаемых результатов. Вставьте перед исходным обращением к system *()*, вызвавшим ошибку, следующую строку.

```
$ENV{PATH} = '/bin:/usr/bin';
```

Учтите, что, если программа echo не находится в каталоге /bin или /usr/bin вашей системы, этот путь надо будет соответственно изменить. Но и в этом случае можно получить другую ошибку, связанную с $ENV{ENV} или какой-то другой переменной окружения. Это вызвано тем, что, когда **Perl** запускает подпроцесс, он проверяет другие переменные окружения, такие как IFS, CDPATH, ENV И BASH_ENV ИЗ оболочки, чтобы узнать, не пусты ли они или загрязнены. Не все оболочки используют эти переменные. Решить проблему можно, вставив в начало скрипта следующую строку.

```
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

Это возможное решение, но не всегда приемлемое. Представьте группу программистов, которые начинают все свои программы с этой строки! Более канонический способ состоит в том, чтобы вызывать *system()* и *exec()* так, чтобы оболочка не смогла раскрывать символы. Лучше всего это сделать, вообще устранив оболочку. Это достигается, как было сказано выше, если параметры передаются в эти функции в виде списка, а не в виде строки.

Всегда пытайтесь добиться при написании скриптов максимально возможной безопасности. Представьте себя на месте скрипта и не доверяйте ничему, что дают вам или для чего пытаются применить вашу работу. Чтобы немного облегчить очистку переменных, на протяжении все этой книги мы будем использовать модуль **Perl** Untaint.pm. Этот модуль можно получить по адресу http://search.cpan.org/search?mode=module&uery=Untaint.

# Установка скрипта

Теперь, когда мы рассмотрели основы, необходимые для работы с остальной частью книги, разберем полный процесс загрузки скрипта CGI. Во-первых, нам нужен сам скрипт CGI. Так как мы еще не приводили его, используем традиционный пример "Hello World". Эта программа прекрасно подходит для любой цели. Она легка в написании, проста и иллюстрирует основные принципы скриптов CGI на Perl. Взгляните на листинг 2.10.

### Листинг 2.10. Пример Hello World

```
01:  #!/usr/bin/perl -wT
02:  print "Content-type: text/html\n\n";
03:  print "<CENTER><H2>Hello World</H2></CENTER>";
```

Строка 1 должна быть вам знакома, если раньше вы имели дело с Perl (или Python, Tcl и т.д.). Одно из отличий скриптов Perl/CGI от скриптов не для CGI заключается в том, что эта строка всегда должна присутствовать. Благодаря ей оболочка узнает, какой интерпретатор использовать. В CGI нет синтаксиса "perl script.pl", как в оболочке.

Строка 2 выводит клиенту заголовок "Content-type". В этом случае (и, вероятно, в большинстве случаев) тип содержимого MIME указывается text/html, что означает, что выводимые данные имеют формат HTML. Броузер должен знать, документ какого типа передается ему, чтобы соответственно обрабатывать поступающие данные.

Строка 3 выводит в броузер клиента текст "Hello World".

Теперь у нас есть простой образец скрипта. Следующий этап — передача этого скрипта на сервер. Вы можете работать непосредственно на Web-сервере — тогда передавать файл нет необходимости. Но Web-сервер может находиться на удаленной машине, поэтому будет полезно проанализировать процесс передачи. Вся передача через FTP в этой книге выполняется старым добрым способом — в командной строке, без GUI. Почему? Потому что мы не знаем, какой FTP-клиент вы используете. Вам придется справиться в документации клиента о том, как использовать команду site и как обеспечить передачу файлов в формате ASCII. Повторяем: *файлы должны загружаться в формате ASCII.* Если файлы будут загружены в двоичном или каком-либо другом неправильном формате, непонятном для интерпретатора Perl, при попытке выполнить скрипт появятся некоторые странные ошибки.

```
01:  % ftp 127.0.0.1
02:  Connected to 127.0.0.1.
03:  220 phish FTP server (Version 6.00) ready.
04:  Name (127.0.0.1:kevin):
05:  331 Password required for kevin.
06:  Password:
07:  230 User kevin logged in.
08:  Remote system type is UNIX.
09:  Using binary mode to transfer files.
```

Строки 1–9 отображают процесс входа на сервер.

```
10:  ftp> ascii
11:  200 Type set to A.
```

В строках 10 и 11 устанавливается режим передачи ASCII (A), так что файл будет загружен правильно.

```
12:  ftp> cd /usr/local/apache/htdocs/book/c2
13:  250 CWD command successful.
```

В строках **12 и** 13 устанавливается каталог, в который будет помещен скрипт CGI.

```
14:  ftp> put hello.pl
15:  local: hello.pl remote: hello.pl
16:  200 PORT command successful.
17:  150 Opening ASCII mode data connection for 'hello.pl'.
18:  100%    81       --:-- ETA
19:  226 Transfer complete.
20:  81 bytes sent in 0.00 seconds (42.97 KB/s)
```

**В строках 14-20** с помощью команды FTP put файл передается на сервер.

```
21:  ftp> ls
22:  200 PORT command successful.
23:  150 Opening ASCII mode data connection for '/bin/ls'.
24:  total 1
25:  -rw-r--r--  1 kevin wheel    78 Dec 27 16:52 hello.pl
26:  226 Transfer complete.
```

**В строках 21—26** выводится содержимое каталога, чтобы показать, что файл передан, а также показать разрешения файла (т.е., его возможные применения).

```
27:  ftp> site chmod 755 hello.pl
28:  200 CHMOD command successful.
```

**Строки 27 и 28** изменяют разрешения файла с 644 на 755 при помощи команды chmod, передаваемой через команду FTP site.

```
29:  ftp> ls
30:  200 PORT command successful.
31:  150 Opening ASCII mode data connection for '/bin/ls'.
32:  total 1
33:  -rwxr-xr-x   1 kevin wheel    78 Dec 27 16:52 hello.pl
34:  226 Transfer complete.
```

**В строках 29—34** содержимое каталога выводится еще раз, чтобы убедиться, что разрешения файла изменились.

```
35:  ftp> bye
36:  221 Goodbye.
```

**Строки 35 и 36** завершают сеанс.

Затем мы... стоп, стоп! Уже все! Итак, затем мы посмотрим на результат в Web-броузере (рис. 2.1).


*Рис.2.1. Экран примера "Hello World"*

# Устранение неисправностей

Такова жизнь программиста: когда вы разрабатываете программу, вы попутно делаете ошибки. При отладке скриптов Perl, которые запускаются из командной строки, реакция наступает немедленно. Если скрипт содержит ошибку, это сразу же отображается на консоли. Но если ошибка генерируется в скрипте Perl/CGI, выполняемом через запрос HTTP, по умолчанию сообщение об ошибке не выводится в броузере клиента. Мало того, ошибки могут происходить фактически на двух уровнях — в скрипте Perl и на Web-сервера.

Одно из главных выработанных на практике правил при работе с CGI (и Perl вообще) — запускать Perl с включенными предупреждениями (переключатель -w). Благодаря этому сообщения об ошибках направляются в журнал ошибок Web-сервера. Для примера вернемся к предыдущему скрипту "Hello World" и внесем в него небольшую синтаксическую ошибку.

**Листинг 2.11. Hello World с синтаксической ошибкой**

```
01:  #!/usr/bin/perl -wT
02:  print "Content-type: text/html\n\n";
03:  print "<CENTERXH2>Hello World</H2></CENTER>;
```

В этом листинге мы удалили из третьей строки закрывающую кавычку. Синтаксическая ошибка этого типа в скрипте CGI вызывает (на Apache) сообщение, показанное на рис. 2.2. Это сообщение очень неопределенно и не может помочь в отладке. Какая именно "внутренняя ошибка сервера" произошла? Действительно ли сервер неправильно настроен, как об этом говорится в сообщении, или сбой произошел в скрипте Perl? Выяснить это поможет только журнал ошибок Web-сервера. В листинге 2.12 показаны записи журнала, соответствующие этой ошибке.

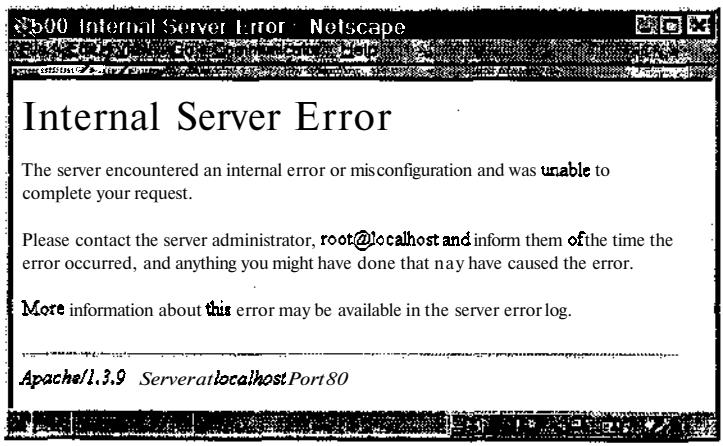



*Рис. 2.2. Экран неопределенной ошибки сервера*

**Листинг 2.12. Синтаксическая ошибка Hello World в журнале ошибок**

```
01: Can't find string terminator '"' anywhere before EOF
    at /usr/locall/apache/htdocs/cgi-bi'n/hello_world.pl line 4.
02:  [Wed Feb 16 22:07:08 2000] [error] [client 127.0.0.1]
    Premature end of script headers:
    /usr/local/apache/htdocs/cgi-bin/hello_world.pl
```

Сообщение в журнале ошибок может дать немного больше информации. Это обычное сообщение о синтаксической ошибке будет появляться в журнале независимо от того, включены ли предупреждения или нет. Как можно видеть, в строке 1 показано сообщение об ошибке, сгенерированное интерпретатором Perl, а в строке 2 — сообщение, сгенерированное Web-сервером. Если в журнале ошибок стоит сообщение Perl (которое всегда находится перед сообщением сервера), то причина ошибки — скрипт Perl, а не неправильная настройка сервера.

Конечно, журнал ошибок полезен, но лучше бы было, чтобы сообщение об ошибке отображалось в броузере, чтобы не нужно было обращаться к журналу. Каждый раз искать ошибки в журнале довольно утомительно, особенно при выполнении большой отладки. Существует способ выводить сообщения о фатальных ошибках непосредственно в броузер. Для этого предназначен модуль `CGI::Carp` (устанавливаемый при установке `CGI.pm`) разработки Линкольна Штайна. Этот модуль содержит метод *fatalsToBrowser ()*, который, если он импортирован, дублирует в броузер все сообщения о фатальных ошибках, выводимые в журнал ошибок Web-сервера.

Теперь снова обратимся к примеру "Hello World", на этот раз с использованием CGI: `:Carp` (листинг 2.13). В этом случае отладка намного упрощается, так как вы можете частично получить эту немедленную реакцию, такую желанную для программиста! На рис. 2.3 показано сообщение, которое на этот раз отображается в броузере.

### Листинг 2.13. Hello World с синтаксической ошибкой и CGI::carp

```
01: #!/usr/bin/perl -wT
02: use CGI::Carp qw(fatalsToBrowser);
03: print "Content-type: text/html\n\n";
04: print "<CENTER><H2>Hello World</H2></CENTER>;
```

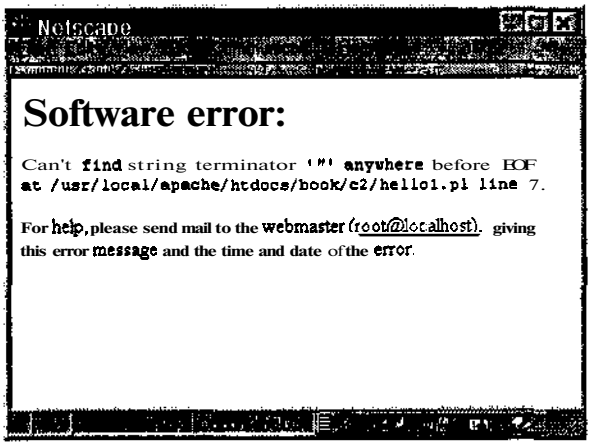



*Рис.2.3. Экран ошибки CGI::Carp*

Теперь дело пошло немного скорее! Мы можем немедленно увидеть ошибку без необходимости просматривать журнал ошибок. Но, *если* взглянуть на этот журнал, можно заметить некоторые изменения. Уже не появится сообщение об ошибке, сгенерированное Web-сервером. Это вызвано тем, что *fatalsToBrowser ()* направляет в броузер минимальное количество заголовков HTTP, так что при ошибке сервер не прерывает процесс. Это нужно и для того, чтобы подавить обычно генерируемое сообщение сервера. Но запомните, что этот метод делает только то, что подразумевает его имя, и не более. Он направляет в броузер сообщения лишь о фатальных ошибках,

хотя это тоже помогает выяснить, что могло вызвать ошибку — Perl или Web-сервер. CGI::Carp мы будем применять на протяжении всей книги, потому что это превосходный способ отладки и получения других полезных сообщений. Но, кроме своей необычайной полезности для отладки, этот метод может раскрыть массу подробной информации, которую не всегда следовало бы обнародовать. Поэтому в реальном применении его не мешает отключить.

В списках рассылки CG1 неоднократно встречается вопрос наподобие "Я получаю ошибку при выполнении скрипта CGI, но из командной строки он работает прекрасно. В чем тут дело?" Такое иногда бывает, когда скрипт запускается из командной строки, а не как CGI. Важно помнить, что, когда Perl выполняется как CG1, он работает в окружении, отличном от того случая, когда он запускается из командной строки с окружением обычной учетной записи пользователя. Однако дело не в окружении — по крайней мере в большинстве случаев, которые мы наблюдали. Скорее, проблема заключается в неправильном отображении заголовков HTTP. Вновь вернемся к нашему примеру. В листинге 2.14 приведен скрипт hello_world.pl с небольшой ошибкой. Это не синтаксическая ошибка Perl и не ошибка в использовании функции Perl. По сути, это вообще не ошибка Perl. Однако она вызывает ошибку сервера. Сначала сравним этот скрипт с листингом 2.13.

### Листинг 2.14. Hello World с ошибкой сервера

```
01:  #!/usr/bin/perl -wT
02:  use CGI::Carp qw(fatalsToBrowser);
03:  print "Content-type: text/html\n";
04:  print "<CENTERXH2>Hello World</H2></CENTER>;
```

Вы нашли ошибку в листинге 2.14? В **строке** 3 имеется только одно \п. Это вызвало бы ошибку сервера, но из командной строки этот скрипт выполняется прекрасно. Всякий раз при отображении HTML в броузере следует указывать после типа MIME (Content-type) два (2) символа новой строки (\п). Необходимость этого разъясняется в документе RFC2616, который описывает HTTP/1.1.

Один из способов проверки скриптов CGI — запуск их из командной строки. После некоторого опыта программирования на Perl поступать так кажется почти естественным. Такой способ отладки очень практичен, но не всегда оптимален. В зависимости от того, что делает скрипт CGI, вы можете не получить ожидаемых результатов. Так в основном и бывает, когда в скрипте используются какие-либо объекты из окружения Web-сервера (см. главу 4), которые не существуют в вашем окружении. Переменные окружения подробно рассматриваются в главе 4.

Во всей этой книге для операций наподобие получения вводимых данных из Web-форм применяется модуль CGI.pm. Более детально об использовании CGI.pm и форм мы расскажем в главе 5; для данного примера нужно только знать, что метод CGl.pm *param()* извлекает введенные данные из формы..

### Листинг 2.15. Hello World со вводом из формы

```
01:  #!/usr/bin/perl -wT
02:  use CGI qw(:cgi);
03:  use CGI::Carp qw(fatalsToBrowser);
04:  my $input = param('input');
05:  print "Content-type: text/html\n\n";
06:  print "<CENTER><H2>$input</H2></CENTER>";
```

Листинг 2.15 — **еще** одна версия нашего простого примера "Hello World". В этом случае ожидается ввод из формы, а именно из поля формы под названием "input".

Если запустить этот скрипт из командной строки с целью проверки или отладки, CGI.pm переведет его в "автономный режим", что означает, что скрипт будет работать не как CGI. В результате мы получаем примерно следующий текст.

### Листинг 2.16. Запуск скрипта CGI с модулем CGI.pm в автономном режиме

```
% ./hello_world.pl
 (offline mode: enter name=value pairs on standard input)
input=hello
<EOF>
Content-type: text/html
```

Введите пару "имя-значение", как здесь предлагается, в форме имя=значение, нажмите <Enter> и введите символ конца файла (EOF) для вашей платформы. Это ^D (<Ctrl-D>) в системах UNIX и ^Z (<Ctrl-Z>) в системах Win32. Как это сделать, показано в листинге 2.17 на примере скрипта "Hello World" из листинга 2.15.

### Листинг 2.17. Полный пример Hello World с модулем CGI.pm в автономном режиме

```
% ./hello_world.pl
Content-type: text/html
 (offline mode: enter name=value pairs on standard input)
input="Hello World"
^D
<CENTERXH2>Hello World</H2></CENTER>
```

При проверке скриптов CGI в командной строке следует помнить еще одно: не надо указывать никаких переключателей времени выполнения интерпретатора, а только запустить сам скрипт. Это вызвано тем, что в CGI переключатели интерпретатора уже заданы в первой строке скрипта. Практически переключатели интерпретатора указывать можно, но если в первой строке скрипта будет включен режим проверки на загрязнение (что следует сделать), а в командной строке не будет указан переключатель -т, произойдет ошибка. Причина в том, что, если запустить скрипт Perl из командной строки таким образом:

```
% perl -<переключатель> script.pl
```

интерпретатор Perl проверит строку скрипта #! и применит переключатели, указанные в ней. Поэтому, если строка #! выглядит так:

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

а Perl запускается так:

```
% perl script.pl
```

то интерпретатор включит предупреждения, поскольку соответствующий переключатель указан в строке #!. Но если строка #! выглядит так:

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

а командная строка — так:

```
% perl script.pl
```

то интерпретатор Perl "видит", что скрипт должен быть выполнен в режиме проверки на загрязнения. Но, поскольку выполнение скрипта уже началось без переключателя -т, устанавливать этот режим слишком поздно, и выполнение скрипта прерывается. Особенность -T заключается в том, что он фактически предназначен для защиты программ setuid, а не CGI. Поэтому многие элементы безопасности имеют смысл только в контексте программы setuid.

Другой хороший способ проверки скриптов CGI (и, по сути, всех скриптов Perl) в командной строке — применение переключателя интерпретатора -c. Этот переключатель вызывает проверку синтаксиса скрипта без его фактического выполнения. В результате мы можем узнать, содержит ли скрипт синтаксическую ошибку, и, если да, то какую. Имейте в виду, что, если скрипт содержит блоки BEGIN ИЛИ END, ОНИ будут выполнены, так как Perl считает, что они не входят в состав собственно скрипта. Это происходит в версиях Perl до 5.6, но начиная с этой версии с переключателем -c выполняется только блок BEGIN. Ниже приведен текст, который выводится при проверке скрипта с правильным синтаксисом. В случае скрипта с неверным синтаксисом будет показана синтаксическая ошибка.

```
% perl -cwT script.cgi
script.cgi syntax OK
```

Еше одна рекомендация — использовать прагму *diagnostics,* которая позволяет видеть более подробные предупреждающие сообщения, что особенно полезно, если вы не очень хорошо знакомы с Perl. По умолчанию эта прагма включает режим -w и отображает подробные сообщения на основе предупреждений, которые генерируются в этом режиме. Их текст можно найти на странице руководства perldiag. Рассмотрим пример отладки следующего простого скрипта из командной строки с использованием прагмы *diagnostics.*

```
#!/usr/bin/perl -wT
print "hello
```

Если считать, что этот скрипт называется script. cgi, мы получим сообщение.

```
% perl -Mdiagnostics -T script.cgi
Нельзя найти признак конца строки '"' вплоть до конца
файла в script.cgi, строка 3 (#1)
 ( F) Строки Perl могут охватывать несколько строк
текста. Это сообщение означает, что заключительный
разделитель был опущен. Поскольку кавычки в скобках
влияют на уровень вложенности, в следующем примере
отсутствует заключительная скобка:
print q(Символ '{' начинает комментарий.);
Если вы получили эту ошибку в данном документе,
вероятно, перед или после закрывающего символа
был вставлен невидимый пробел. Найти этот символ
можно при помощи хорошего редактора программных текстов.
Неперехваченное исключение от кода пользователя:
Нельзя найти признак конца строки '"' вплоть до конца
файла в script.cgi, строка 3 (#1)
```

# Кэширование

Web-кэширование — один из способов уменьшить сетевой трафик через Web и помочь пользователям Web быстрее получать различные документы. Сегодня, когда в Internet так много пользователей, каждому из них доступна лишь определенная пропускная способность. Это утверждение особенно справедливо для локальных сетей, которые соединяются с Internet через прокси-сервер. При таком количестве отправителей и потребителей информации и ограниченной пропускной способности работа в Internet может значительно замедляться. Поэтому WWW часто расшифровывают как Worldwide Waiting — всемирное ожидание. Кэширование — способ избежать этого замедления. Кэш — как кэш броузера, так и кэш сервера — "сохраняет" копии документов, затребованных из Internet. Впоследствии конечному пользователю по его запросу можно предоставлять локальную кэшированную копию, а не вытягивать весь документ заново из сети. В этом разделе мы рассмотрим кэши сервера и броузера и покажем, как сделать так, чтобы документы не кэшировались.

Все это имеет большое значение при создании приложений CGI, так как в некоторых ситуациях, когда страница перезагружается, на ней не отображаются ожидаемые изменения. Например в главе 7 мы рассмотрим различные способы подсчета посещений Web-страницы. Может случиться так, что вы загрузите страницу еще раз, а счетчик посещений не изменится. Не волнуйтесь, с ним все в порядке! Причина может быть в том, что вы получили страницу из кэша, а не непосредственно с Web-сервера. В конце этого раздела мы покажем, как обеспечить, чтобы результаты скриптов CGI не кэшировались. Но сначала рассмотрим различия между кэшем броузера и кэшем сервера.

Кэширование в броузере происходит, когда сам Web-броузер сохраняет документы, которые конечный пользователь просматривает в Internet. Все основные Web-броузеры применяют кэширование как на диске, так и в памяти, и дают конечному пользователю возможность выделять ресурсы для каждого типа кэширования. Кэширование на диске имеет место, когда броузер сохраняет файлы (текст и изображения) в каталоге на жестком диске. При кэшировании в памяти, напротив, информация сохраняется в ОЗУ. Кэширование в броузере очень полезно для конечного пользователя. Например, если вы посещаете сайт — скажем, Web-сайт новостей — несколько раз в день, было бы лишней тратой времени и пропускной способности каждый раз загружать данные с этого сайта при обновлении страницы (если срок действия данных не истек)[3]. При кэшировании броузер знает, что он должен получить обновленную информацию с удаленного Web-сервера, только если данные устарели[4].

Кэширование на сервере основано на тех же принципах, что и кэширование в броузере, с тем исключением, что кэш сервера предназначен для использования многими броузерами, а не только одним. Часто компании устанавливают кэш на своих прокси-серверах Internet. Такой кэш сохраняет документы, которые запрашивают пользователи, и может выдавать их конечным пользователям точно также, как это делает кэш броузера. Это значительно сокращает сетевой трафик и снижает время ожидания. Прокси-кэши — это кэши совместного доступа, так как их используют все пользователи, проходящие через данный прокси-сервер. Узнать, как сконфигурировать кэширование в броузере или на прокси-сервере, лучше всего из документации по ним.

Теперь, когда у вас есть общее представление о кэшировании, попробуем выяснить, как запретить прокси-серверам и броузерам кэшировать определенные документы. Для чего это нужно? Если наша страница (или страницы) претерпевает частые изменения, надо гарантировать, чтобы посетители всегда видели самые последние модификации. Или же, если на Web-странице часто меняются изображения, следует сделать так, чтобы они загружались каждый раз, когда конечный пользователь захочет просмотреть эту страницу[5]. Один из способов, который часто считается достаточным для запрещения кэширования страницы — дескриптор HTML META типа "Pragma: no-cache". Однако этот способ неэффективен, так как прокси-кэши не будут видеть этот код HTML. Прокси-сервер не читает HTML, так что результат не будет достигнут. Однако на кэш броузера это, скорее всего, подействует, поскольку броузер читает HTML. Но это лишь половинный эффект. Если вы хотите, чтобы ваши документы не кэшировались, надо сделать так, чтобы это было обязательно и для прокси, и для броузеров. Лучший способ добиться такого результата дают заголовки HTTP. В этом заголовке можно задать два важных параметра: Expires и Cache-Control[6].

---

[3] *Данные с истекшим сроком действия считаются "устаревшими", а не истекшим — "свежими".*

[4] *Посмотрите параметры настройки вашего Web-броузера, управляющие тем, когда он должен искать обновленную информацию, и должен ли вообще кэшировать ее.*

[5] *Справьтесь в документации Web-сервера о том, как установить кэширование отдельных объектов, таких как изображение и текст.*

[6] *Если вы не можете управлять Web-сервером, а те, кто им управляет, не дают вам возможности устанавливать заголовки HTTP, попытайтесь убедить их разрешить вам это.*

Но, прежде чем перейти к ним, мы скажем несколько слов о кэшировании безопасных (SSL) и защищенных документов. Безопасные документы (с использованием SSL) не кэшируются и не расшифровываются прокси-кэшами. Web-страницы с доступом через проверку подлинности HTTP, как правило, помечаются как частные и поэтому не кэшируются кэшами совместного доступа (прокси-кэшами). Но существует способ разрешить кэширование безопасных документов при помощи параметра заголовка Cache-Control, который мы сейчас рассмотрим. В общем, документы SSL не расшифровываются и не кэшируются, а безопасные документы по умолчанию не кэшируются, но могут кэшироваться.

## Заголовок HTTP Expires

Заголовок HTTP Expires служит для того, чтобы сообщить кэшу, когда документ должен считаться устаревшим. Благодаря ему броузер знает, когда надо обратиться на исходный сервер, чтобы увидеть, был ли документ изменен. Значением этого раздела заголовка является дата. Если требуется гарантировать, что документ не будет кэшироваться, надо установить уже прошедшую дату. Таким образом, когда броузер или прокси-сервер прочтет заголовок, он увидит, что документ не новый, и не будет кэшировать его. В этом случае параметр Expires может выглядеть примерно так.

```
Expires: Fri, 31 Dec 199.0 23:59:59 GMT
```

Как можно заметить, время указано по Гринвичу (GMT), а не по локальному часовому поясу. Если требуется, чтобы документ кэшировался до наступления определенной даты, следует установить будущую дату. Это очень полезный способ избежать или, напротив, обеспечить кэширование документов.

## Заголовок HTTP Cache-Control

С появлением HTTP 1.1 в обращение вошел новый заголовок: **Cache-Control**. Этот полезный заголовок дает разработчикам возможность не только разрешать или запрещать кэширование, но и указывать, как должен быть обработан документ[7]. Так как эта директива очень подробно описана в спецификации HTTP 1.1, мы рассмотрим лишь некоторые параметры Cache-Control.

```
max-age=[секунды]
s-maxage=[секунды]
```

Параметр *шах-аде* устанавливает время в секундах, на протяжении которого данный документ считается свежим, с момента первоначального запроса. Это напоминает действие заголовка Expires, с тем исключением, что здесь указывается промежуток времени в секундах с момента запроса, а не определенная дата. Параметр *s-maxage* действует так же, но применяется только к кэшам совместного доступа.

```
no-cache
```

Эта директива запрещает кэшировать данный документ, и запрашиваемый документ должен *каждый раз* проверяться с исходного сервера. Впрочем, мы еще не говорили о проверке документов. Web-кэш (на данный момент) проверяет документ путем сверки времени его последнего изменения (Last-Modified) со временем документа на исходном сервере. Если документ на сервере новее, чем в кэше, загружается более свежая версия; если нет, отображается документ из кэша. Это гарантирует, что посетители вашего сайта всегда будут видеть самые последние материалы. Учтите, что это не отменяет физического кэширования документа, но кэш должен сначала обратиться к исходному серверу за более свежей версией документа, прежде чем удалить его из кэша.

```
public
```

[7] *Директива Cache-Control очень подробно рассматривается в спецификации HTTP 1.1 в RFC2626, раздел 14.9.*

Эта директива указывает, что документ может кэшироваться даже тогда, когда это обычно не делается, например, в случае безопасного документа. Сочетание этой директивы с *no-cache* позволяет кэшировать безопасные документы, но заставляет клиента каждый раз повторно подтверждать свою подлинность при сверке времени последнего изменения на исходном сервере. При этом выполняется запрос/ответ опознавания, и клиент, не прошедший проверки подлинности, не получает кэшированного документа.

Теперь, когда вам известно, что должно быть в заголовке документа, чтобы он не кэшировался (или, наоборот, обязательно кэшировался), мы рассмотрим простой заголовок HTTP.

### Листинг 2.18. Пример заголовка HTTP

```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Server: Apache/1.3.11 (Unix)
Cache-Control: no-cache, public
Expires: Wed, 31 Dec 1980 23:59:59 GMT
Last-Modified: Fri, 31 Dec 1999 23:59:59 GMT
Content-Length: 1040
Content-Type: text/html
```

Как можно видеть из этого заголовка, мы отправляем документ HTML и запрещаем кэширование. В листинге 2.10 было показано, как вывести часть заголовка. Этот метод можно применить и теперь или же обратиться к модулю CGI.pm, как рассказано далее в этой книге.

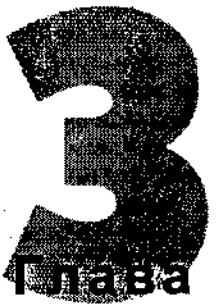
# Листинги

### Листинг 2.19. Пример очистки в командной строке taint_ex.pl

```
01:  #!/usr/bin/perl -wT
02:  use strict;
03:  my $foo = join('',@argv);
04:  if (is_tainted($foo)) {
05:    print "\$foo загрязнен. Пытаюсь очистить\n";
06:    my $pattern = qr (^\d{3}(-|\s+)?\d{3}(-|\s+)?\d{4}$);
07:    $foo = untaint($foo, $pattern);
08:  } else {
09:    print "\$foo не загрязнен!!\n";
10:  }
11:  sub is_tainted {
12:    my $check = shift;
13:     return !eval {$check ++, kill 0;1;};
14:  }
15:  sub untaint {
16:    my ($foo, $pattern) = @_;
17:    if ($foo = - /($pattern)/ {
18:      $foo = $1;
19:      print "\$foo очищен!!\n";
20:       return $foo;
21:    } else {
22:       print "Нельзя очистить \$foo\n";
23:       return $foo;
24:    }
25:  }
```

**Листинг 2.20. Пример сеанса FTP**

```
01:  % ftp 127.0.0.1
02:  Connected to 127.0.0.1.
03:  220 phish FTP server (Version 6.00) ready.
04: Name (127.0.0.1:kevin) :
05:  331 Password required for kevin.
06:  Password:
07:  230 User kevin logged in.
08:  Remote system type is UNIX.
09:  Using binary mode to transfer files.
10:  ftp> ascii
11:  200 Type set to A.
12:  ftp> cd /usr/local/apache/htdocs/book/c2
13:  250 CWD command successful.
14:  ftp> put hello.pl
15:  local: hello.pl remote: hello.pl
16:  200 PORT command successful.
17:  150 Opening ASCII mode data connection for 'hello.pl'.
18:   100%    81       --:-- ETA
19:  226 Transfer complete.
20:  81 bytes sent in 0.00 seconds (42.97 KB/s)
21:  ftp> ls
22:  200 PORT command successful.
23:  150 Opening ASCII mode data connection for '/bin/ls'.
24:  total 1
25:   -rw-r--r--   1 kevin wheel    78 Dec 27 16:52 hello.pl
26:  226 Transfer complete.
27:  ftp> site chmod 755 hello.pl
28:  200 CHMOD command successful.
29:  ftp> ls
30:  200 PORT command successful.
31:  150 Opening ASCII mode data connection for '/bin/Is'.
32:  total 1
33:  -rwxr-xr-x   1 kevin wheel    78 Dec 27 16:52 hello.pl
34:  22 6 Transfer complete.
35:  ftp> bye
36:  221 Goodbye.
```




# Использование окружения

## Введение в %ENV

Переменные окружения — это скрытые значения, к которым Web-сервер позволяет обращаться скриптам CGI. Web-сервер работает в своем собственном окружении, которое включает и создает эти переменные. Это означает, что Web-сервер имеет данные о выполняемых процессах и может применять их для обслуживания запросов. Список некоторых переменных окружения сервера приведен в приложении Б, но мы также рекомендуем ознакомиться с документацией по Web-серверу, чтобы узнать, какие переменные окружения в нем существуют по умолчанию. Web-клиент также передает Web-серверу некоторые данные, которые сохраняются в переменных окружения. Выяснять, что именно сервер делает с каждой из этих переменных — не наша задача. Эта книга лишь объясняет, как получать, использовать и задавать эти переменные в скриптах CGI. Все эти переменные позволяют получить информацию о посетителе (Web-клиенте), произведенном запросе и самом Web-сервере.

Что такое "окружение" в точном смысле слова? Вам может быть знакомо окружение оболочки, которое содержит такую информацию, как пути к программам, домашний каталог, редактор и т.д. и создает среду, в которой вы работаете. Можно прибегнуть к другой, менее специальной аналогии. Представьте, что окружение — это ваше рабочее место. Мой рабочий кабинет — это "оболочка". В "процессе" работы мое окружение составляют такие веши, как телефон, почтовый адрес, стол, освещение, компьютер, стул и различные принадлежности и материалы. Все это можно рассматривать как переменные окружения. Некоторые из них, например, номер телефона, компьютер и стол, не будут изменяться. Другие, такие как используемые материалы, освещение или стул, при необходимости могут быть изменены. Это очень похоже на окружение Web-сервера и его переменные. Окружение содержит известный набор переменных, которые могут (как освещение) или не могут (как номер телефона) изменяться. Некоторые **использу-**

ются больше, чем другие (я работаю с компьютером больше, чем с карандашом), а некоторые используются почти все время (как мой стул).

Определенная информация в окружении, как, например, SERVER_NAME, устанавливается при запуске Web-сервера и не изменяется в ходе его работы. Некоторые переменные окружения, как имя сервера и программное обеспечение, задаются самим Web-сервером. Некоторые переменные задаются пользователем, под учетной записью которого выполняется процесс сервера (обычно пользователем *nobody).* К ним относится, например, PATH. Другие переменные определяются информацией, передаваемой Web-клиентом на сервер — например, агент пользователя и удаленный IP. Переменные окружения, которые создаются данными, передаваемыми клиентом на Web-сервер, всегда представляют информацию о самом клиенте. Имейте в виду, что Web-клиенты могут направлять неверную информацию. Они не могут делать это преднамеренно, но некоторые пользователи создают собственные специальные клиенты, чтобы изменить эту информацию. Это не обязательно бывает повседневной проблемой для Web-сайта, но не мешает помнить, что эта информация *может* быть неверной. Независимо от того, кем установлены переменные окружения, вы можете использовать их и изменять. Окружение также позволяет добавлять при необходимости новые переменные, как делает оболочка.

Насколько полезны переменные окружения, если вы ничего не знаете о них? Полезны, но не особенно. Поэтому сейчас мы напишем скрипт, отображающий переменные окружения Web-сервера в броузере. В этом скрипте появится хэш %ENV. Это хэш (ассоциативный массив), в котором Perl хранит все доступные ему переменные окружения. Ключи %ENV — это имена переменных окружения, а соответствующие им значения — текущие значения этих переменных. Заметьте, что %ENV всегда содержит текущие значения. Мы употребляем слово "текущие", так как эти значения можно изменять на время выполнения скрипта. Все изменения в %ENV будут доступны каждому дочернему процессу, созданному как часть соответствующего окружения, но изменения в дочернем окружении не будут влиять на родительский процесс. Если вы имеете опыт в программировании Perl в оболочке, вы уже должны быть до некоторой степени знакомы с %ENV. Чтобы просмотреть переменные нашего окружения, введем после приглашения командной строки следующий текст.

```
% perl -e 'print qq($_ - $ENV{$_}\n) foreach sort keys %ENV'
```

А теперь попробуем преобразовать этот однострочный пример в форму для CGI. Скрипт в листинге 3.1 отображает в Web-клиенте переменные окружения Web-сервера.

### Листинг 3.1. Скрипт CGI, отображающий переменные окружения

```
01: #!/usr/bin/perl -wT
02: $|=1;
03: print "Content-type: text/html\n\n";
04: print <<HTML;
05: <HTML>
06: <HEAD>
07: <TITLE>Окружение сервера</TITLE>
08: </HEAD>
09: <CENTER><H2>Окружение сервера</H2></CENTER>
10: <BODY>
11: HTML
12: print qq($_ - $ENV{$_}<BR>) for sort keys %ENV;
13: print qq(
14: </BODY>
15: </HTML>
16: );
```

Строка 1 — наша обычная строка с указанием пути к интерпретатору и переключателей. Здесь мы включаем предупреждения переключателем -w. Хотя переключатель -w для этого простого скрипта не очень нужен, его использование — просто хорошая традиция. По этой же традиции включена проверка на загрязнение (переключатель -T). Гораздо лучше и безопаснее установить в самом начале максимум зашиты и предупреждений, а затем, после проверки, ослабить защиту, где это возможно. Вообще, в CGI рекомендуется устанавливать столько мер безопасности, сколько возможно.

Строка 2 производит автосброс буфера, так что мы можем увидеть результаты сразу после получения.

**Строка** 3 выводит в Web-клиент информацию заголовка. Благодаря этому клиент будет ожидать файл, содержащий текст в формате HTML.

Строка 4 начинает включенный документ с разделителя HTML.

Строки 5—10 — строки HTML, выводимые во включенном документе.

Строка И — признак конца включенного документа. Запомните, что он должен стоять в начале строки и в отдельной строке.

Строка 12 — сам механизм скрипта. Здесь выводятся все пары ключей и значений, содержащиеся в %ENV. Эти пары перечисляются в алфавитном порядке для удобства чтения.

Строки 13—16— оператор *print),* который закрывает дескрипторы HTML <BODY> и <HTML>.

Результат выполнения этого скрипта CGI в броузере показан на рис. 3.1.

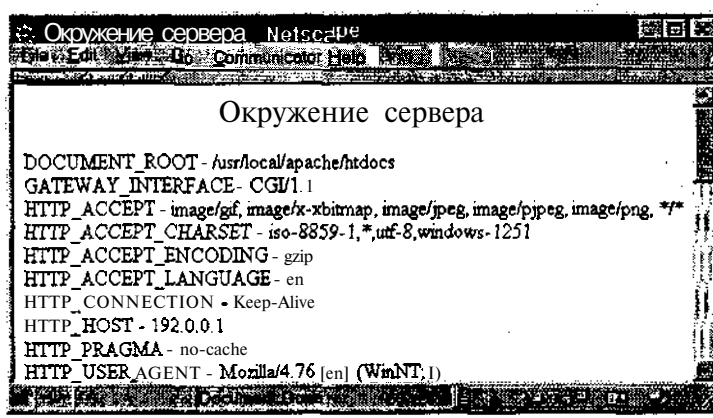



*Рис. 3.1. Экран переменных окружения*

# Добавление в %ENV

Иногда возникает необходимость добавить данные в окружение или создать новые переменные окружения. Это особенно полезно, когда какая-то информация должна совместно использоваться несколькими скриптами или всеми скриптами в системе. Реализовать это можно двумя способами. Во-первых, создать переменную окружения в файле конфигурации Web-сервера. В случае Apache для этого служит директива SetENV в файле `httpd.conf`.

В текстовом редакторе, который вы предпочитаете, откройте файл `httpd.conf` и добавьте директиву следующего синтаксиса.

SetENV *переменная* значение

После этого следует перезапустить Web-сервер, чтобы новая переменная произвела эффект. Запомните, что всякий раз после внесения изменений в файлы конфигурации Apache сервер надо перезапускать, чтобы изменения вступили в силу.

Реальным примером использования заданной таким образом переменной окружения может быть ситуация, когда во многих скриптах используется одна база данных или, наоборот, когда баз данных несколько. Допустим, что с сервером работает группа программистов, и вы хотите избавить их от необходимости вспоминать имена и расположения определенных баз данных. Или, возможно, вы единственный разработчик и не хотите запоминать эту постоянную информацию. Этого можно очень просто добиться, задав в конфигурации Web-сервера соответствующую переменную окружения. Часто встречается ситуация, когда, например, база данных с информацией по управлению находится на сервере A, а другая база данных с информацией по продажам — на сервере B. База данных продаж может называться *sales-info* и располагаться на удаленном сервере *sales_server,* а база данных управления *manager_info* — на удаленном сервере *manager-server*. Имея эту информацию, мы можем создать для ее хранения следующие переменные окружения.

```
SetENV SALESDB sales_info@sales-server
SetENV MANDB manager_info@manager-server
```

После перезапуска сервера эти две новые пары имя-значение появятся в `%ENV` И могут быть использованы в скриптах. Продолжая предыдущий пример, мы предположим, что эти базы данных имеют формат MySQL и для подключения к ним применяется модуль `DBI.pm`. Чтобы облегчить жизнь программиста, сейчас самое время создать модуль подключения к базе данных с помощью DBI. В этой главе речь идет не о DBI, но важно, чтобы вы увидели один из реальных примеров использования окружения. Работа с Perl DBI подробно рассматривается в одной из следующих глав этой книги.

```
01: package MyConnect;
```

Строка 1 объявляет имя пакета для модуля — в данном случае MyConnect. Поэтому файл будет называться `MyPackage.pm`. Всегда при подобном объявлении имени пакета имя файла состоит из имени главного класса с расширением `.pm`.

```
02: use DBI;
03: use strict;
```

**Строки 2 и 3** задают прагму `strict`, которая включает ограничения компилятора, и импортируют модуль **DBI,** который будет устанавливать соединение и выполнять всю рутинную работу по взаимодействию между скриптом Perl и базой данных.

```
04: sub salesDB {
```

**Строка 4** начинает метод salesDB.

```
05:     my ($sales_dbname, $sales_dbhost) =
          split(/\@/,$ENV{'SALESDB'});
```

**В строке** 5 создаются две переменные, `$sales_dbname` и $sales_dbhost, для хранения соответственно имен базы данных и ее сервера. С этого момента начинает действовать новая переменная окружения. Значение SALESDB из `%ENV` может быть получено тем же путем, что и любое значение хэша. Это значение разбивается на имя базы данных и имя сервера; знаком разделения служит символ `@`.

```
06:     my $sales_dsn = "DBI:mysql:database=
          $sales_dbname;host=$sales_dbhost";
```

**Строка 6** создает имя источника данных (data source name — DSN), формат которого определен в модуле **DBI,** и сохраняет его в переменной `$sales_dsn`. DSN имеет следующий синтаксис:

```
DBI:<имя_драйвера>:database=<имя_базы_данных>;
host=<хост_базы_данных>
```

По этой информации модуль DBI узнает, на какой машине находится база данных, каковы ее имя и тип. Но здесь указывается не сам тип базы данных, а только драйвер, используемый для подключения к ней — в данном случае драйвер для MySQL. На Perl написаны драйверы для большинства реляционных СУБД (эти драйверы можно найти в CPAN, в пространстве имен DBD). DBI — очень полезный модуль, так как при его использовании нужно лишь установить драйвер и указать этот драйвер в определении DSN. Все остальное выполняется скрыто от программиста. Например, мы работаем с MySQL, но можно использовать и Oracle (или Informix). Для этого надо просто установить драйвер `DBD::Oracle` (который сам по себе является модулем Perl), внести в DSN указание на драйвер Oracle или Informix вместо драйвера MySQL, и дело сделано. Пользу от модуля DBI трудно переоценить, когда речь идет о работе с базой данных с использованием Perl. К этому модулю прилагается обширная документация, с которой мы рекомендуем вам ознакомиться. Часть его функциональных возможностей не используется в этой книге, но может оказаться полезной для вас.

```
07:    my $sales_dbh=DBI->connect($sales_dsn,
         "username","password");
```

**В строке 7** производится подключение к базе данных, и дескриптор базы данных (database handle — DBH) сохраняется в скаляре $sales_dbh. Этот дескриптор — указатель, или курсор, на открытое подключение к базе данных. Метод *connect ()* можно представить как дверь в базу данных. Переменная $sales_dbh — способ пройти через эту дверь.

```
08:    if (!defined($sales_dbh)) {
09:    print "\nerror: Проблема с подключением к базе
         данных MySQL:\n";
10:    print DBI->errstr;
11:    print "-" x 25 . "\n";
12:    return;
13: }
```

**В строках 8—13** производится небольшой контроль ошибок. **Строка 8** проверяет, удачно ли прошло подключение, и, если нет, выводится сообщение о том, какая ошибка подключения произошла.

```
14:    return $sales_dbh;
```

**Строка 14** возвращает переменную, в которой "хранится" подключение к базе данных. Это позволяет внешнему скрипту, который вызовет метод *SalesDB()*,присвоить какой-либо переменной значение возвращенного **DBH** — в данном случае $sales_dbh.

```
15: }
```

**Строка 15** закрывает метод *SalesDB()*.

```
16:    1;
```

**Строка 16** возвращает значение истины в скрипт, который импортирует данный модуль с помощью `use()`. Это необходимо, и если бы скрипт не возвращал это значение, произошла бы ошибка.

Ниже приведен пример использования этого модуля и создаваемого им подключения к базе данных в другом скрипте `sales.cgi`, показанном в листинге 3.2.

## Листинг 3.2. Пример использования модуля подключения к базе данных

```
01: #!/usr/bin/perl -wT
02: # sales.cgi
03: $|=1;
```

```
04:  use strict;
05:  use lib qw(.);
06:  use MyConnect;
07:  my $sales_dbh = MyConnect->salesDB;
08:  my $sales_sth = $sales_dbh->prepare("SELECT * FROM
      table where ID='5' ORDER BY date");
09:  $sales_sth->execute;
10: и т.д.
```

**Строка 1** — традиционная первая строка с указанием пути к интерпретатору Perl. Также здесь включаются предупреждения и проверка на загрязнение. Хотя переключатель -w не особенно нужен для этого простого скрипта, лучше, чтобы это вошло у вас в привычку, когда вы перейдете к реальной работе.

**Строка** 2 — имя этого скрипта.

**Строка** 3 производит автосброс буфера, так что мы можем увидеть результаты сразу после получения.

**Строка 4** включает ограничения компилятора с помощью прагмы `strict`.

**Строка** 5 требует особого внимания. Один из побочных эффектов режима проверки на загрязнение, о котором говорилось в главе 2, — удаление текущего рабочего каталога (.) из массива `@INC`. `lib.pm`— это прагма времени компиляции, во многом похожая на `strict.pm`. Когда она импортируется, она добавляет указанный в параметре список каталогов в `@INC`. Это необходимо, так как `MyConnect.pm` сейчас расположен в текущем рабочем каталоге, и надо сделать так, чтобы Perl смог его увидеть. Применение `lib` для изменения `@INC`, как в этом сценарии, рекомендуется при включении проверки на загрязнения в CGI.

В **строке 6** импортируется модуль MyConnect, который мы только что создали. Теперь мы можем вызвать его метод подключения.

**Строка 7** вызывает метод *salesDB()*,который мы создали в MyConnect.pm, и присваивает возвращаемое значение переменной Ssalesdbh. Теперь мы можем использовать это подключение в главном скрипте, поскольку метод возвратил его дескриптор, или DBH.

**Строка** 8 — пример подготовки запроса с помощью метода `prepare` *()* и сохранение его в новой переменной $sales_sth. Такой способ обычно используется для обозначения дескриптора команды (statement handle — STH). Метод *prepare ()* готовит запрос к выполнению.

**Строка 9** выполняет запрос к базе данных `sales_info` на сервере `sales-server`.

**Строки 10 и далее** — остальная часть вашего кода.

Мы рекомендуем читателю самостоятельно создать в модуле MyConnect второй метод, устанавливающий подключение к базе данных управления.

# Основы ввода через форму

В следующей главе мы подробно рассмотрим обработку форм в CGI и расскажем, как направлять их в броузер и управлять информацией, которую они возвращают. Однако наше знакомство с переменными окружения было бы неполным, если бы мы не показали один из наиболее обычных примеров их использования. Это получение ввода из формы с помощью переменной окружения `QUERY_STRING`. Эта переменная содержит запрос, который направляется в CGI. Рассмотрим, например, следующую форму.

**Листинг 3.3. Форма HTML**

```
<FORM ACTION="/cgi-bin/query_string.cgi" METHOD="GET">
Имя: <INPUT TYPE="text" NAME="fname" SI2E=20xP>
Фамилия: <INPUT TYPE="text" NAME="lname" SIZE=20><P>
<INPUT TYPE="submit" VALUE="Submit">
```

При отправке эта форма будет направлена на URL

```
http://your.domain.com/cgi-bin/query_string.cgi?fname=
  Joseph&lname=Hall
```

Конечно, здесь будут те значения, которые вы ввели, не обязательно Joseph и Hall. После отправки сервер извлекает из URL часть запроса, стоящую после знака ?, и помещает ее в $ENV{ 'QUERY_STRING'}. Эта часть будет выглядеть так.

```
fname=Joseph&lname=Hall
```

Методы GET и POST мы рассмотрим в следующей главе, так как они работают несколько по-разному. Но уже сейчас вы можете создавать формы и видеть, как читаются и отображаются данные, полученные методом **GET.** В следующем скрипте дан пример разбора QUERY_STRING на соответствующие друг другу пары имен и значений. Листинг 3.4 — это скрипт query_string.cgi, который работает с приведенной выше формой **HTML.**

**Листинг 3.4. Пример разбора QUERY_STRING вручную**

```
01:  #!/usr/bin/perl -wT
02:  use strict;
03:  print "Content-type:text/html\n\n";
04:  my %FORM;
05:  my @pairs = split(/&/, $ENV{'QUERY_STRING'});
06:  foreach (@pairs) {
07:    my ($name, Svalue) = split(/=/, $_) ;
08:    $value =~ tr/+/ /;
09:    $value=~s/% ([a-fA-F0-9][a-fA-F0-9])/pack("C",
         hex($1) )/eg;
10:    $FORM{$name} = Svalue;
11:  }
12:  print «HTML;
13:  <HTML>
14:  <HEAD><TITLE>Вывод формы</TITLE></HEAD>
15:  <BODY  BGCOLOR="#ffffff">
16:  <CENTER><H3>Вывод формы</H3></CENTER>
17:  <P>
18:  HTML
19:  foreach (keys %FORM) {
20:  print "$_ = $FORM{$_}<BR>";
21:  }
22:  print «HTML;
23:    </BODY>
24:    </HTML>
25:  HTML
```

Большая часть этого листинга, кроме **строк 5–10,** должна быть уже знакома вам, так что мы сосредоточимся именно на этих строках.

**В строке 5** выполняется разбиение значения QUERY_STRING, которое имеет вид имя=значение&имя2=значение2&имя3=значение3 и т.д. Оператор split разбивает эту строку на пары имя-значение, разделяя ее по символам & и записывая каждую пару как элемент в массив @pairs. Каждый элемент этого массива имеет вид имя=значение.

**В строке 6** начинается цикл обработки каждой из пар в `@pairs`.

В **строке** 7 каждый элемент разделяется по знаку = и создаются две переменные, содержащие соответственно имя и значение.

**В строке 8** производится декодирование URI. Когда информация передается через форму, она кодируется в формат URI. Этот вопрос мы подробно рассмотрим в следующей главе, но в двух словах можно сказать, что это заключается в преобразовании некоторых символов в шестнадцатеричные эквиваленты, которые Web-сервер и направляет в скрипт CGI. Список этих символов приводится в приложении находится в приложении Г., "Общедоступная лицензия". Здесь все знаки + транслитерируются в пробелы.

**В строке 9** продолжается декодирование URI, но на этот раз с применением регулярного выражения. Здесь все шестнадцатеричные значения преобразуются обратно в соответствующие значения ASCII с помощью функции *`pack()`*, которая возвращает строку в двоичной структуре. После этого переменная `$value` приобретает в точности то значение, которое было введено пользователем.

**Строка** 10 добавляет пару имя-значение в хэш `%FORM`. С этой парой можно сделать все что угодно (например, поместить ее в массив или другую структуру данных), но хранение в хэше упрощает последующее получение значений.

**Строка 11** завершает цикл.

Этот пример демонстрирует разбор поступающих данных формы "вручную". Конечно, в **Perl** это можно сделать несколькими способами (ЭМСНС), и один из этих способов, использование `CGI.pm`, мы подробно опишем в следующих главах. Но уже теперь мы можем оценить разницу между предыдущим примером и листингом 3.5, в котором для решения той же задачи применяется `CGI.pm`.

**ЛИСТИНГ 3.5. Пример получения ввода из формы с помощью `CGI.pm`**

```
01:  #!/usr/bin/perl -wT
02:  use strict;
03:  use CGI qw(:standard);
04:  my $first_name = param('fname');
05:  my $last_name = param('lname');
06:  print header,
07:    start_html('Вывод формы'),
08:    h3({-align=>'center'},  'Вывод формы'),
09:    p;
10:  print $_ . " = ". param($_)  . br foreach param;
11:  print end_html;
```

Какое отличие! Казалось бы, зачем разбирать `QUERY_STRING` вручную, если можно предоставить эту работу CGI.pm. Действительно, если такие причины и есть, они вряд ли будут достаточно вескими, чтобы делать разбор вручную. Так почему же мы сначала занимались именно этим? Мы считаем, что важно понять, как это делается, а не просто принять к сведению, что модуль сделает эту работу за вас. Другая причина в том, что это разбор переменной окружения, а эта глава посвящена переменным окружения! По ходу этой книги вы увидите множество других примеров использования `CGI.pm` и получите более глубокие объяснения, как он выполняет ту или иную задачу и как это можно использовать.

# Пример скрипта: журнал посетителей

Очень часто CGI применяют для создания специальной системы регистрации для Web-сайта. Конечно, очень немногие Web-серверы обращаются к журналам, создаваемым такими системами, но информация из них может быть полезна для какой-то

специальной базы данных. Представьте ситуацию, когда требуется создавать отчеты с данными о посетителях Web-сайта. Для таких отчетов нужно знать доменные имена посетителей, их Web-клиенты, посещенные страницы и дату посещения. Эти немногочисленные, но важные сведения позволяют генерировать отчеты самых разнообразных типов. Например, отчеты могут показывать, какие Web-клиенты и в каком количестве применяются для просмотра сайта, часы наибольшей активности, какие посетители приходят на сайт и откуда.

В этом примере объединяется все, о чем мы рассказали в этой главе. Сюда входит создание новых переменных окружения в файле конфигурации Web-сервера, подключение к базе данных с помощью библиотечного модуля и использование некоторых стандартных переменных окружения, которые содержат информацию о посетителе.

Работу над примером мы начнем с создания базы данных. В этой базе будут храниться шесть элементов информации, поэтому мы создадим таблицу с четырьмя колонками. Таблица будет называться access_log и входить в базу данных visitors на сервере raysqlhost. Она создается запросом SQL из листинга 3.6.

### Листинг 3.6. Команда SQL create

```
create table access_log {
  IP CHAR(15),
  DOMAIN CHAR(255),
  CLIENT CHAR(75),
  PAGE_TO CHAR(255),
  PAGE_FROM CHAR(255),
  TIME_STAMP CHAR(20)
)
```

### Примечание

Проверьте в документации по применяемой базе данных, каковы ограничения на длину типов CHAR и VARCHAR.

В зависимости от того, какие отчеты генерируются и какие запросы делаются наиболее часто, было бы целесообразно включить в таблицу соответствующие индексы. Затем мы добавим в файл httpd.conf специальную переменную окружения. Эта переменная будет содержать расположение базы данных на случай, если эта информация понадобится для будущих скриптов.

```
SetENV VISLOGDB visitors@mysqlhost
```

Затем мы напишем небольшой модуль создания подключения к базе данных. Рекомендуется создавать код для многократного использования, помещая в отдельные модули методы для выполнения действий, необходимых во многих скриптах. В предыдущем примере мы показали, как выполнить эту задачу путем создания "библиотечного" скрипта для установки подключения. Это другой, и, вероятно, более эффективный способ. Продемонстрируем оба способа, чтобы дать читателю максимум информации.

Так как подключение к базе данных посетителей, скорее всего, будет выполняться многими скриптами и приложениями, следующий модуль, листинг 3.7, поможет сэкономить время при их создании.

### Листинг 3.7. Модуль LogConnect для подключения к базе данных

```
01: package LogConnect;
02:  use DBI;
03:  use strict;
```

```
04: sub connect {
05:   my($log_dbname,$log_dbhost);=
06:    my $log_dsn = "DBI:mysql:database=
        $log_dbname;host=log_dbho5t";
07:   my $log_dbh=DBI->connect($log_dsn, "user", "password");
08:   if (!defined($log_dbh}) {
09:      print "\nerror: Проблема с подключением к
          базе данных MySQL:\n";
10:      print DBI->errstr;
11:     print "-" x 25 "\n";
12:      return;
13:    }
14:    return $log_dbh;
15: }
16: 1;
```

**Строка** 1 начинает модуль и устанавливает имя пакета создаваемого модуля. Это имя файла модуля без расширения .pm.

**Строка 2** экспортирует модуль DBI, так как с его помощью мы будем устанавливать подключение к базе данных.

**Строка 3** включает ограничения компилятора.

**Строка 4** начинает метод *connect ()*. Обычно на этом месте стоит метод *new()*. Однако в этот раз мы не создаем новый класс или объектно-ориентированный модуль. Наша задача состоит в том, чтобы получить сочетание подпрограмм, которое можно будет использовать многократно.

В **строке** 5 создаются две переменные — $log_dbname и $log_dbhost, в которых будут храниться соответственно имя базы данных и имя сервера. Как было показано выше, их значения извлекаются из переменной окружения, которую мы создали в файле конфигурации Web-сервера.

В строке 6 создается имя DSN, как оно определено в модуле DBI, и присваивается переменной $log_dsn.

**Строка 7** устанавливает подключение к базе данных и записывает дескриптор базы данных в переменную $log_dbh.

**В строках 8—13** выполняется контроль ошибок. Если метод *connect ()* не возвращает нужное значение, отображается сообщение об ошибке, возвращаемое модулем DBI. Всегда рекомендуется выполнять какой-нибудь контроль ошибок при создании подключения.

**Строка 14** возвращает значение $log_dbh, т.е. дескриптор DBH, в вызывающий скрипт. Теперь этот вызывающий скрипт имеет открытое подключение к базе данных, которое он может использовать по своему усмотрению.

**Строки 15** и **16** завершают метод и возвращают значение истины.

Теперь у нас есть таблица MySQL, специальная переменная окружения, и мы можем подключиться к базе данных. Осталось написать скрипт vislog.cgi, который будет регистрировать посетителей.

```
01: #!/usr/bin/perl -wT
02: # vislog.cgi
```

**Строка 1** — как обычно. Обратите внимание, что здесь включены как предупреждения, так и проверка на загрязнение. Хотя в этот скрипт не поступают никакие данные от пользователя, он, тем не менее, остается скриптом CGI.

```
03:   use strict;
```

**В строке 3** включается ограничение опасных конструкций.

```
04:  use POSIX;
```

**Строка 4** импортирует модуль POSIX. Этот модуль позволяет работать почти со всеми идентификаторами интерфейса POSIX 1003.1. В данном примере он служит для того, чтобы красиво отформатировать данные времени в строку. Это выполняет метод *strftim()*. Следует проверить в документации по базе данных, имеются ли в ней встроенные поля для штампов даты и времени. Но для примера мы создали такое поле самостоятельно.

```
05:  use Socket;
```

**Строка 5** вводит модуль `Socket.pm`. Использование этого модуля ограничено одной определенной задачей: получение значения AF_INET. AF_INET — это константа, которая определена в `socket.h` и используется операционной системой в процессах связи через сокеты по протоколам TCP, UDP и т.д. Часто вместо `AF_INET` ставят просто значение 2 — наиболее обычное значение этой константы в `socket.h`, но делать это не рекомендуется. Хотя чаше всего AF_INET равно **2**, на какой-то машине это значение может быть другим, и тогда скрипт не сможет правильно работать и не будет переносимым. Модуль `Socket.pm` устанавливает для экспортируемой константы AF_INET значение, используемое в вашей операционной системе.

```
06:  use lib qw(.);
```

**Строка 6** должна быть знакома вам из предыдущего примера. Еще раз повторим, что режим проверки на загрязнение удаляет текущий рабочий каталог из @INC, И поэтому мы должны добавить его снова, так как в нем находится наш модуль.

```
07:  use LogConnect;
```

**Строка 7** импортирует модуль LogConnect, который мы только что создали. Мы применим его метод для подключения к базе данных.

```
08:  my $dbh = LogConnect->connect;
```

**Строка 8** вызывает метод *connect ()* нашего модуля LogConnect и присваивает значение возвращаемого дескриптора базы данных переменной `$dbh`.

```
09:  ray  $ip = $ENV{'REMOTE_ADDR'};
```

**В строке 9** значение переменной окружения `REMOTE_ADDR`, в которой хранится IP-адрес посетителя, присваивается скалярной переменной Sip.

```
10:  my $browser = $ENV{'HTTP_USER_AGENT'};
```

**Строка 10** обращается к переменной окружения HTTP_USER_AGENT, которая содержит информацию о Web-клиенте посетителя. Это значение присваивается переменной Sbrowser.

```
11:  my $referer = $ENV{'HTTP_REFERER'};
```

**Строка 11** получает значение `HTTP_REFERER` — URL Web-страницы, с которой посетитель пришел на этот сайт. Если такого значения нет, пользователь начал просмотр Web с этого сайта.

```
12:  my $here = $ENV{'REQUEST_URI'};
```

**Строка 12** обращается к переменной окружения `REQUEST_URI`. В этой переменной хранится **URI** запрашиваемой страницы. Это значение передается в Shere. **URI** — это путь к сценарию на вашем сервере без указания домена. Например, если URL сценария — http://www.you.cora/cgi-bin/script.pl, то URI будет /cgi-bin/script.pl: URL минус информация о домене.

```
13:  my @digits = split(/\./, $ip);
```

В строке 13 IP-адрес посетителя разбивается по октетам и создается массив `@digits`. Это делается затем, что эта информация должна быть передана в функцию *pack()* без точек.

```
14:  my $address = pack("C4", @digits);
```

В строке 14 целые числа в `@digits` при помощи *pack()* упаковываются в четырехбайтовое значение типа char без знака. В результате получается двоичная структура, значение которой сохраняется в $address.

```
15:  my $host = gethostbyaddr($address, AF_INET);
```

В строке 15 переменной $host присваивается значение, возвращаемое функцией *gethostbyaddr()*. Эта функция преобразует заданный IP-адрес в доменное имя, например `me.myself.com`. Именно здесь используется `AF_INET`. В функции *gethostbyaddr()* эта константа служит для определения типа найденного адреса. Если указано AF_INET, функция будет знать, что это адрес типа UDP, TCP и т.д.

```
16:  my $time = strftime("%Y-%m-%d %H:%M:%S",gmtime);
```

В строке 16 переменная $time получает значение функции *strftime()* (string format time — время в строковом формате), импортированной из модуля POSIX. Текущее время определяется с помощью функции Perl *gmtime()*. Она возвращает строку со временем по Гринвичу. Сама по себе эта функция не особенно полезна, но можно заменить *gmtime()* на *localtime()* и получить локальное время. В этом примере строка преобразуется в формат YYYY-MM-DD HH:MM:SS, который задан шаблоном `%Y-%m-%d %H:%M:%S`.

```
17:  my $query = qq(insert into access_log values
       ('$ip','$host','$browser','$here','$referer,
       '$time'));
```

В строке 17 формируется запрос на вставку данных в базу данных. Этот запрос SQL сохраняется в переменной $query.

```
18:  my $sth = $dbh->prepare($query);
```

Строка 18 создает переменную $dbh, которая будет содержать STH. Ей присваивается значение подготовленного запроса, возвращаемое методом DBI *prepare()*. Теперь запрос SQL готов к выполнению.

```
19:  $sth->execute;
```

В строке 19 выполняется запрос SQL и информация сохраняется в базе данных.

```
20:  $dbh->disconnect;
```

**Строка 20** закрывает открытое подключение к базе данных.

Этот скрипт несколько растянут, чтобы проиллюстрировать, что происходит на каждом этапе. Более компактный вариант представлен в листинге 3.8.

## Листинг 3.8. Другой способ регистрации посетителей

```
01:  #!/usr/bin/perl -wT
02:  use strict;
03:  use Socket;
04:  use POSIX;
05:  use lib qw(.);
06:  use LogConnect;
07:  my $dbh = LogConnect->connect;
08:   (my $digits = $ENV{'REMOTE_ADDR'}) =~ s/\.//g;
```

```
09:  my $address = pack("C4", $digits);
10:  my $host = gethostbyaddr($address, AF_INET);
11:  my $time = strftime("%Y-%m-%d %H:%M:%S", gmtime);
12:  my $sth = $dbh->prepare(qq{insert into access_log values
       ('$ENV{REMOTE_ADDR}','$host',
       '$ENV{HTTP_USER_AGENT}', '$ENV{DOCUMENT_URI}',
       '$ENV{HTTP_REFERER}',  '$time')});
13:  $sth->execute;
14:  $dbh->disconnect;
```

Различия между двумя скриптами очевидны. Этот скрипт SSI можно вызвать с любой Web-страницы, которая может выполнять SSI. Вызов имеет следующий вид.

```
<!—#exec cgi="/cgi-bin/vislog.cgi"-->
```

# Пример скрипта: простейший отчет

Теперь, когда вся эта информация хранится в базе данных, хорошо бы было найти способ ее использовать. Как мы говорили выше, данные такого рода полезны для создания отчетов. Одним из таких отчетов может быть статистика Web-клиентов, в которой приводятся данные о том, какие Web-клиенты используются для просмотра вашего сайта и в каком количестве. Это может помочь Web-разработчикам больше узнать о потребителях их разработок. Если окажется, что большинство посетителей использует клиенты Netscape, интенсивное применение таких специфических для Internet Explorer технологий, как VBScript и JScript, может быть нежелательно. Может оказаться, что основную массу клиентов составляют текстовые броузеры, такие как Lynx, и тогда надо сделать упор на текстовых возможностях.

Мы рассмотрим скрипт, который подключается к базе данных `visitors`, извлекает информацию из ее таблицы `access_log` и создает динамическую Web-страницу, на которой отображается статистика Web-клиентов. Многие из этих операций уже должны быть вам знакомы. Этот скрипт будет называться `viewlog.cgi`.

```
01:  #!/usr/bin/perl -wT
02:  # viewlog.cgi
03:  $|=1;
04:  use strict
```

Назначение строк 1–4 должно быть вам известно. Еще раз подчеркнем, что включение проверки на загрязнение и предупреждений рекомендуется для всех скриптов.

Переменная Perl `$|` — это специальная переменная, через которую программисты могут указать, чтобы Perl выполнял автоматический сброс выходного буфера. Если ее значение не равно нулю, Perl будет направлять вывод в Web-клиент после каждой операции записи, а не ждать, пока буфер заполнится. Нежелательно задавать `$|=1` только для Web-страниц, содержащих таблицы, так как некоторые Web-клиенты отображают таблицу, только когда ее код завершен и появился дескриптор `</TABLE>`. Так что иногда этот скрипт может работать не так, как ожидается. Это не ошибка Perl, а лишь особенность работы Web-клиента.

```
05:  use lib qw(.);
06:  use LogConnect;
```

В строках 5 и 6 снова вызывается `lib.pm`, чтобы гарантировать, что текущий рабочий каталог будет присутствовать в @INC, и импортируется модуль LogConnect.

```
07:  my $dbh = LogConnect->connect;
```

Строка 7 устанавливает подключение к базе данных с помощью *LogAccess: -.connect ()*; возвращаемый DBH сохраняется в переменной $dbh. Как

уже можно догадаться, мы рекомендуем давать переменным, в которых хранятся дескрипторы базы данных, имена, содержащие символы "dbh". Хорошее и простое соглашение об именах не приносит ничего, кроме пользы.

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

**Строка 8** выводит в клиент заголовок, который заставляет его интерпретировать поступающую информацию как HTML.

```
09: my $query = qq(select CLIENT from access_log);
```

**Строка 9** формирует запрос для получения необходимой информации из базы данных. Этот запрос выбирает в базе данных все колонки CLIENT.

```
10: my $sth = $dbh->prepare($query);
```

**Строка 10** готовит запрос к выполнению с помощью метода DBI `prepare ()`.

```
11: $sth->execute;
```

**Строка 11** выполняет запрос.

```
12: my %clients;
```

**В строке 12** объявляется хэш под названием `%clients`. В нем будут храниться типы клиентов, зарегистрированные в журнале, и количество каждого типа.

```
13: while (x$_ = $sth- > fetchrow) {
14:    $clients{$_}++;
15: }
```

**Строки 13—15** — цикл по данным, возвращаемым запросом SQL. Когда какой-либо Web-клиент встречается первый раз, его имя сохраняется в ключе хэша, а соответствующее ему значение становится равным 1. При каждом последующем появлении этого клиента значение данного ключа увеличивается на единицу.

```
16: $dbh->disconnect;
```

**В строке 16** производится отключение от базы данных.

```
17: my @clients = sort {$clients{$b} <=> $clients{$a}}
      keys %clients;
```

**Строка 17** — последний этап обработки перед выводом информации в клиент. Данные будут отображены в порядке убывания количества клиентов. Это выполняется путем создания массива (@clients в данном случае), элементы которого — имена Web-клиентов, отсортированные так, что сначала следуют наиболее часто встречающиеся клиенты. Сортировка ключей `%clients` производится по их значениям.

```
18: print «HTML;
19: <HTML>
20: <HEAD>
21: <TITLE>Простой отчет</TITLE>
22: </HEAD>
23: <BODY BGCOLOR="#ffffff">
24: <CENTER><H1>Простой отчет</H1></CENTER>
25: <CENTER><H3>Статистика Web-клиентов</H3></CENTER>
26: <HR NOSHADE>
27: <CENTER>
28: <TABLE BORDER=1 CELLPADDING=3 CELLSPACING=3>
29: HTML
```

**Строки 18—29** — начало Web-страницы, которое выводится во включенном документе. Заметьте, что начало таблицы также выводится здесь.

```
30:  foreach (@clients) {
31:  print qq(<TD ALIGN=center>$_</TD><TD ALIGN=center>
       $clients{$_}</TD><TR>);
32:  }
```

**Строки 30—32** — цикл foreach, который просматривает массив @clients и выводит каждый его элемент в две **ячейки** таблицы. В левую ячейку выводятся данные о Web-клиенте из переменной $_, которая содержит последнее значение итерации foreach. В правой ячейке выводится значение для этого клиента из %clients.

```
33:  print «HTML;
34:  </TABLE>
35:  </CENTER>
36:  </BODY>
37:  </HTML>
38:  HTML
```

**Строки 33—38** — включенный документ, содержащий заключительные дескрипторы HTML, которые завершают отображение в броузере пользователя. Примерный результат отображения показан на рис. 3.2.

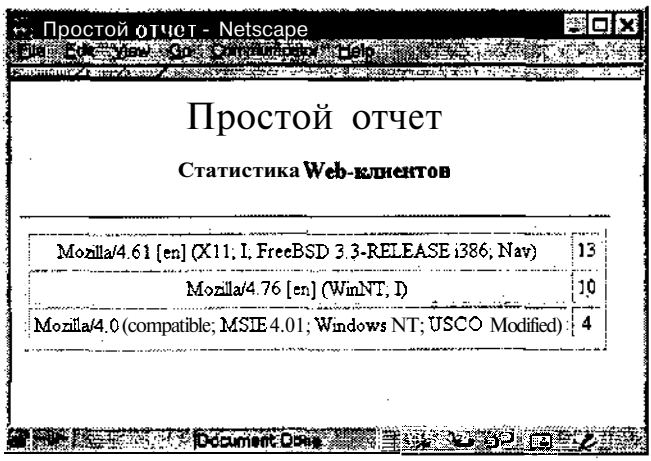



*Рис. 3.2, Экран статистики клиентов*

# Упражнения

Как и в случае других скриптов этой книги, у нас теперь есть Web-приложение, готовое к применению, но, вероятно, требующее некоторой настройки в соответствии с вашими потребностями. Кроме того, некоторые из показанных методов не дают самых простых и выразительных способов достижения цели, так как их назначение учебное, но, тем не менее, они достаточно эффективны. Мы хотели бы предложить читателю поупражняться с этим кодом, возможно, исключить некоторые этапы и добавить новые функции.

Одно из таких упражнений, которое должно помочь вам лучше понять концепции, изложенные в этой главе, состоит в создании отчета, содержащего более подробную информацию, на основании той же базы данных. Например, в колонке CLIENT фактически содержится и другая полезная информация, такая как версия Web-клиента и тип операционной системы. При помощи строковых функций и регулярных выражений эти данные можно извлечь из строк, возвращаемых полем CLIENT, и дополнить отчет, чтобы он содержал более подробную статистику, включая полное количество

посетителей, полное количество каждого из Web-клиентов, статистику версий Web-клиентов и используемых ОС.

У вас должен получиться скрипт, примерные результаты выполнения которого представлены на рис. 3.3.

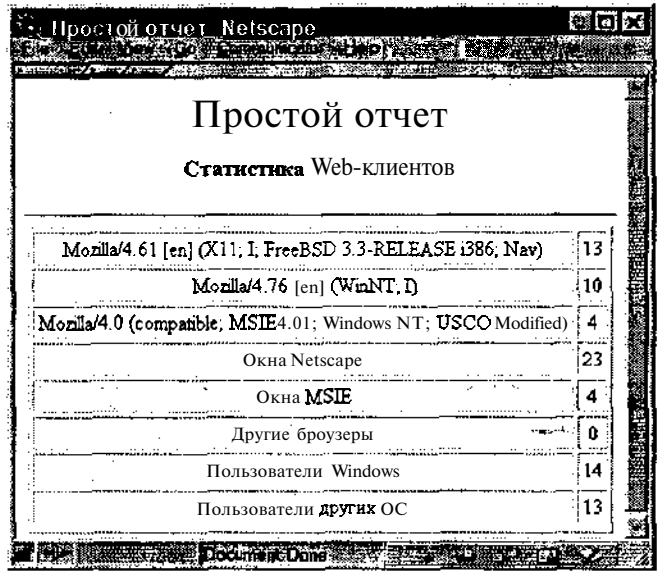



*Рис. 3.3. Экран подробной статистики клиентов*

# Что мы изучили

- Что такое `%ENV` И ЧТО она содержит.
- Как добавить новые переменные в окружение Web-сервера Apache, используя SetENV.
- Функция *gethostbyaddr*
- Как провести разбор `QUERY_STRING` "вручную".

# Листинги

**Листинг 3.9. `MyConnect.pm`**

```
01: package MyConnect;
02: use DBI;
03: use strict;
04: sub salesDB {
05:    my ($sales_dbname, $sales_dbhost) =
         split(/\@/,$ENV{'SALESDB'});
06:    my $sales_dsn = "DBI:mysql:database=
         $sales_dbname;host=$sales_dbhost";
07:    my $sales_dbh=DBI->connect($sales_dsn,
         "username","password");
08:    if C!defined($sales_dbh))  {
```

```
09:    print "\nerror: Проблема с подключением к базе
         данных MySQL:\n";
10:    print DBI->errstr;
11:    print "-" x 25 . "\n";
12:    return;
13: }
14:    return $sales_dbh;
15: }
16: 1;
```

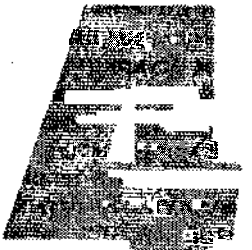
**ЛИСТИНГ 3.10. vislog.cgi**

```
01:  #!/usr/bin/perl -wT
02:  # vislog.cgi
03:  use strict;
04:  use POSIX;
05:  use Socket;
06: use lib qw(.);
07:  use LogConnect;
08:  my $dbh = LogConnect->connect;
09:  my $ip = $ENV{'REMOTE_ADDR'};
10: ray $browser = $ENV{'HTTP_USER_AGENT'};
11:  my Sreferer = $ENV{'HTTP_REFERER'};
12:  my Shere = $ENV{'REQUEST_URI'};
13:  my @digits = split(/\./, $ip);
14:  my $address = pack("C4", @digits);
15:  my $host = gethostbyaddr($address, AF_INET);
16:  my $time = strftime("%Y-%m-%d %H:%M:%S",gmtime);
17:  my $query = qq(insert into access_log values
       ('$ip','$host','$browser','$here','Sreferer,
       '$time'));
18:  my $sth = $dbh->prepare($query);
19:  $sth->execute;
20:  $dbh->disconnect;
```

**ЛИСТИНГ 3.11. viewlog.cgi**

```
01:  #!/usr/bin/perl -wT
02:  # viewlog.cgi
03:  $|=1;
04:  use strict
05:  use lib qw(.);
06:  use LogConnect;
07:  my $dbh = LogConnect->connect;
08:  print "Content-type: text/html\n\n";
09:  my $query = qq(select CLIENT from access_log);
10:  my $sth = $dbh->prepare($query);
11:  $sth->execute;
12:  my %clients;
13:  while ($_ = Ssth- > fetchrow) {
14:    $clients{$_}++;
15:  }
16:  $dbh->disconnect;
17:  my @clients = sort {$clients{$b} <=> $clients{$a}}
       keys %clients;
18:  print «HTML;
```

```
19:  <HTML>
20:  <HEAD>
21:  <TITLE>Простой отчет</TITLE>
22:  </HEAD>
23:  <BODY BGCOLOR="#ffffff">
24:  <CENTER><H1>Простой отчет</H1></CENTER>
25:  <CENTER><H3>Статистика Web-клиентов</H3></CENTER>
26:  <HR NOSHADE>
27:  <CENTER>
28:  <TABLE BORDER=1 CELLPADDING=3 CELLSPACING=3>
29:  HTML
30:  foreach (@clients) (
31: . print qq(<TD ALIGN=center>$_</TD><TD ALIGN=center>
      $clients{$_}</TD><TR>);
32:  }
33:  print «HTML;
34:  </TABLE>
35:  </CENTER>
36:  </BODY>
37:  </HTML>
38:  HTML
```



Глава

# Введение в Web-формы

## Введение

Задача получения любой информации через Web обязательно включает в себя использование форм HTML. Эти формы позволяют создавать Web-страницы, содержащие поля ввода данных. Практически в каждом приложении CGI где-нибудь непременно встречается форма HTML. Форма HTML— фактически только графический интерфейс пользователя (GUI) для Web-приложений.

Если бы не формы, почти все данные в Web были бы статичными, потому что никак нельзя было бы ввести новые данные! Формы делают Web из статической среды — динамической, позволяя передавать информацию *назад* на Web-сервер. Получение на сервер данных *пользователя* придает Web большую силу. Теперь становится возможным не только распространять информацию, но и собирать информацию о пользователе и в ответ предоставлять ему информацию, подобранную *специально* для него.

В настоящее время существует очень немного дескрипторов форм HTML, так что их изучение, как и применение, не должно быть *особенно* трудным. Спецификации Web находятся в состоянии постоянного изменения. Появляются новые технологии, такие как XML и XHTML, так что не удивляйтесь, если в ближайшем будущем Web-формы приобретут еще более широкие возможности. Не волнуйтесь, **HTML** никуда не денется. HTML просто станет более мощным, возможно, получив такие элементы, как табличный ввод, бегунки и многостраничные формы.

Далее мы опишем большую часть дескрипторов форм **HTML, но** не будем рассматривать их особенно подробно. Эта глава — только введение. Если вам понадобится всестороннее описание с объяснением *всех* возможностей, обратитесь на http://www.w3.org и просмотрите документы по текущей спецификации **HTML (в** настоящее время XHTML 1.01), а именно раздел по формам HTML.

## <FORM>

Все формы должны начинаться и завершаться дескриптором <FORM>. Когда мы говорим о формах, мы имеем в виду *логическую* форму на Web-странице (текст между открывающим дескриптором <FORM> И закрывающим дескриптором </FORM>). Web-страница может содержать несколько форм, поэтому не следует смешивать саму по себе Web-страницу с формой *на* Web-странице.

Формы не могут быть вложенными, т.е. форма не может находиться внутри другой формы. Если написать такой код, это вызовет неожиданные результаты, так как Web-броузеры не могут обрабатывать формы внутри других форм.

Дескриптор <FORM> также может содержать необязательные атрибуты ACTION, NAME, METHOD и ENCTYPE. См. пример в листинге 4.1.

**Листинг 4.1. Полный дескриптор FORM**

```
<FORM ACTION="/cgi-bin/myprogram.cgi" METHOD="POST"
NAME="form1" ENCTYPE="multipart/form-data">
```

ACTION сообщает форме HTML, какой скрипт CGI надо вызвать, когда пользователь отправит форму. Скрипт CGI и форма HTML могут находиться на разных серверах. Если скрипт CGI находится на другом сервере, в параметре ACTION должен быть указан полный URL. Если скрипт CGI находится на том же сервере, что и форма HTML, данные о сервере можно опустить, как это сделано в листинге 4.1.

NAME — имя формы. Этот атрибут чаще всего не указывается, но при работе с формой через JavaScript, в случае нескольких форм на одной странице или если скрипт CGI должен выполнять различные действия в зависимости от того, какая форма отправлена, нужно знать имя отправленной формы.

METHOD указывает, какой способ отправки должен применить броузер. Возможные варианты — GET и POST. Подробнее мы поговорим о них позже, а сейчас вам достаточно запомнить, что по умолчанию принят метод GET.

ENCTYPE сообщает броузеру, как кодировать данные для передачи. Если этот атрибут опущен, значение по умолчанию— application/x-www-form-urlencoded. Другое значение ENCTYPE — multipart/form-data. Этот последний метод более новый и более эффективный для передачи больших объемов данных и двоичных данных; он также требуется для скриптов, работающих с загрузкой файлов. Для броузеров новых версий, 4.x и выше, можно просто всегда указывать multipart/form-data, но, так как этот способ несовместим со старыми версиями броузеров, обычно он используется только для загрузки файлов.

**Листинг 4.2. Полный дескриптор FORM**

```
<FORM ACTION="/cgi-bin/myprogram.cgi" METHOD="POST"
NAME="form1" ENCTYPE="multipart/form-data">
```

В листинге 4.2 использованы все возможные атрибуты <FORM>. Атрибут ACTION указывает, что программа CGI, которая будет выполнена, называется myprogram.cgi. Атрибут METHOD указывает, что будет применен метод POST. ИМЯ ЭТОЙ формы — form1, а тип кодировки — multipart/form-data.

**Листинг 4.3. Компактный дескриптор FORM**

```
<FORM ACTION="/cgi-bin/myprogram.cgi">
```

В листинге 4.3 задан только атрибут ACTION. В этом случае мы вызываем программу CGI под названием `newprogram.cgi`. Так как METHOD и ENCTYPE не указаны, будут применены значения по умолчанию — соответственно GET И `application/x-www-form-urlencoded`. Атрибут NAME не требуется и не имеет значения по умолчанию, так что имени у этой формы просто нет.

Листинг 4.3. Минимальный дескриптор FORM

```
<FORM>
```

В листинге 4.4 используется один только дескриптор `<FORM>`! В этом случае значения всех атрибутов принимаются по умолчанию. Сервер знает, какое приложение CGI надо выполнить, так как, если значение ACTION не задано, по умолчанию принимается текущий скрипт, и, таким образом, он запускает сам себя. Это звучит довольно странно, но на самом деле этот механизм очень полезен, если скрипт CGI имеет несколько различных функций. Например, простой CGI, вызываемый без параметров, будет выводить форму HTML, но, если в него переданы данные, программа будет их обрабатывать. Так иногда делают, поскольку проще сопровождать одну программу, чем программу и связанный с ней файл HTML.

Если вы забудете поставить дескриптор `<FORM>` В начале формы HTML, Netscape не покажет никаких элементов этой формы, но некоторые версии Internet Explorer смогут отобразить ее. Если форма не будет завершена дескриптором `</FORM>`, результаты также будут зависеть от броузера.

## GET И POST

Методы GET и POST различаются способом передачи данных на сервер. GET передает данные на URL, и к этим данным можно обращаться через переменную окружения `QUERY_STRING`. Метод POST возвращает данные на сервер через стандартный ввод (`STDIN`). Оба метода имеют свои преимущества и недостатки.

Метод GET позволяет легко просматривать все данные на URL. Программист может создавать "заготовки" запросов в виде ссылок, передающих данные в URL.

Листинг **4.5. Пример получения данных через ссылку HTTP**

```
http://www.google.com/search?q=brent+michalski&sa=Google+Search
```

Заметьте, что в этом запросе данные поиска "brent michalski" передаются в переменной q=. Значение "Google Search" переменной `sa=` получено от кнопки отправки формы. Знаки +, s и - служат для того, чтобы в URL не было пробелов. Если бы между моим именем и фамилией стоял пробел, запрос передал бы только первую часть, "brent", или возвратил бы ошибку. Разные броузеры обрабатывают пробелы поразному. Согласно спецификациям в URL не должно быть пробелов.

Метод GET превосходно работает в большинстве случаев, но иногда он может быть неудобен. В этом методе есть ограничение на количество передаваемых данных. Фактическое количество зависит от Web-сервера. Кроме того, если вы не хотите, Чтобы пользователи видели передаваемые данные, метод GET неприменим, так как *все* данные отображаются в строке URL. Например, если в форму вводятся какие-то сведения секретного или личного характера, пользователи, вероятно, предпочли бы, чтобы эти данные не были видны никому. Поля ввода пароля — еще одна причина, по которой применение GET нецелесообразно. В следующем листинге и на рис. 4.1 и 4.2 показано, что получится, если передать пароль методом GET.

### Листинг 4.6. Пример пароля

```
<HTML>
<FORM ACTION="/cgi-bin/book/pwd.cgi">
  Введите секретное слово:
  <INPUT TYPE="PASSWORD" NAME="SECRET"><P>
  <INPUT TYPE="SUBMIT">
</FORM>
</HTML>
```

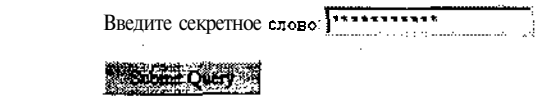



*Рис. 4.1. Поле для ввода пароля*

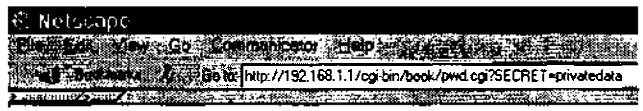



*Рис. 4.2. Строка запроса при методе GET*

Как можно видеть, данные, которые при вводе были скрыты (`<INPUT TYPE="PASSWORD">`), в строке URL отображаются как обычный текст! Слово *privatedata* в запросе — это пароль, который вы ввели.

При методе POST данные, передаваемые в программу CGI, не видны для пользователя. Недостаток этого метода состоит в том, что программист также не может видеть, что передается. Метод POST не имеет ограничений на количество данных, которые могут быть переданы на сервер, хотя у Web-сервера такое ограничение может быть. Иногда при написании и отладке приложений CGI полезно сначала применить метод `GET`, чтобы выяснить, что именно передается, а затем, при передаче скрипта в эксплуатацию, заменить GET на `POST`.

Еще одно замечание: вполне допускается использовать в формах метод POST и передавать данные в URL. Это может быть полезно в ситуациях, когда требуется передавать данные конфигурации или другую информацию, которую легче изменять прямо в строке URL, а не обращаться к форме каждый раз. В листинге 4.7 показано, как можно передать данные конфигурации в URL.

### Листинг 4.7. Передача файла конфигурации

```
http://www.myscripts.com/cgi-bin/myscript.cgi?config=test.cfg
```

Данные, возвращаемые из формы в программу CGI, можно декодировать вручную, но при этом надо быть очень внимательным. Модуль `CGI.pm` выполняет эту задачу уже давно, и вышло много версий, прежде чем он стал удобным и безопасным. Собственные подпрограммы декодирования надо создавать только в особых случаях, например, для изучения работы CGI или, если в этом для вас уже нет секретов, для усовершенствованных версий CGl.pm, которые будут работать лучше, чем стандартный модуль. В остальных ситуациях можно положиться на CGl.pm, который очень хорошо умеет декодировать значения. Можно даже вызывать метод POST, но передавать данные в URL, как в методе GET! ДЛЯ модуля CGl.pm это не составляет никакого труда. Самостоятельно же реализовать все эти возможности было бы непросто.

# Дескрипторы формы

Теперь, когда мы знаем, как начинать и завершать форму HTML, можно переходить к другим дескрипторам, с которыми вы наиболее вероятно столкнетесь при создании форм HTML. Мы рассмотрим обычно используемые и встречающиеся элементы формы и их атрибуты. Описание всех разнообразных возможностей выходит за рамки этой главы; любую и наиболее свежую информацию по этому вопросу можно получить на http://www.w3.org.

## <INPUT>

Дескриптор <INPUT> — вероятнее всего, наиболее часто используемый дескриптор. Он содержит множество различных атрибутов, и, чтобы применять его, вы должны быть хорошо знакомы с этими атрибутами. <INPUT> не имеет закрывающего дескриптора. Дескрипторы <INPUT> должны содержать атрибут <NAME>, так как программе CGI нужно знать имя элемента данных, чтобы обращаться к нему. Каждое поле в форме обычно, но не всегда, имеет уникальное имя.

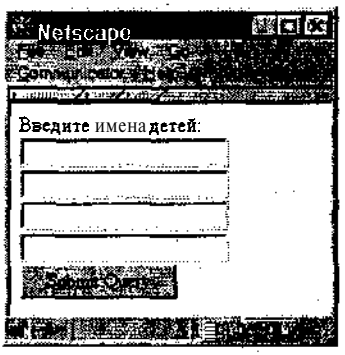



*Рис. 4.3. Форма с полями ввода текста*

Если имеется несколько элементов с одинаковыми именами, CGI.pm может обработать данные двумя разными способами. Он выдает все значения, если они читаются в массив, или только первое значение, если они читаются в скаляр.

На рис. 4.3 показана простая форма HTML. Предположим, что каждое из полей <INPUT> называется "kids": <INPUT TYPE="text" NAME="kids">. Посмотрим, как в зависимости от типа переменной, в которой сохраняются данные, CGI.pm по-разному анализирует их.

Листинг 4,8. Чтение данных функцией param

```
my @data = param ('kids');
my $data = param ('kids');
```

В первой строке этого листинга все элементы данных "kids" в форме HTML сохраняются в массиве @data. Во втором примере в скалярной переменной Sdata сохраняются только данные из первого поля ввода "kids".

## TYPE

Атрибут TYPE сообщает, какого рода это поле <INPUT>. Разные типы полей <INPUT> отображаются и действуют в форме HTML по-разному. Атрибут TYPE может иметь значения TEXT, RADIO, CHECKBOX, HIDDEN, PASSWORD, FILE, SUBMIT, RESET И IMAGE.

## <INPUT TYPE="TEXT">

Атрибут TEXT используется наиболее часто. Он служит для приема одной строки, введенной пользователем, совсем как поле ввода текста в приложении Windows. Атрибут TEXT позволяет также задавать атрибуты SIZE И MAXLENGTH. Атрибут SIZE сообщает броузеру, какой ширины в символах должно отображаться поле в форме HTML; значение по умолчанию — 20. MAXLENGTH определяет максимальное количество символов, которое пользователь может ввести. Как только количество символов превысит значение MAXLENGTH, поле станет недоступным для ввода.

**Листинг 4 ичение длины текста в поле ввода**

```
<INPUT TYPE="TEXT" NAME="state" MAXLENGTH="2" SIZE="3">
```

В этом листинге объявляется поле ввода текста с именем state. Оно имеет ширину 3 символа (SIZE) и позволяет вводить до 2 символов (MAXLENGTH). Рекомендуется устанавливать размер поля немного больше, чем максимальная длина, чтобы учесть различные броузеры и шрифты. Если MAXLENGTH не указан, по умолчанию принимается, что в поле INPUT можно вводить неограниченное число символов.

## <INPUT TYPE="PASSWORD">

PASSWORD — другой тип поля <INPUT> (см. листинг 4.10). Поля типа PASSWORD работают так же, как и поля типа TEXT, С тем отличием, что вместо фактически вводимых символов отображаются звездочки (*). Крайне важно понять, что это поле никак не шифрует или скрывает текст, когда он передается в программу CGI. Действительно, если для отправки формы, содержащей поле пароля, применить метод GET, пользователь увидит введенный пароль как простой текст в строке URL! Это иллюстрирует рис. 4.2. Для какой бы то ни было реальной защиты надо применять проверку подлинности на стороне сервера с файлом .htaccess или какой-то другой метод.

**Листинг 4.10. Поле ввода пароля**

```
<INPUT TYPE="PASSWORD" NAME="secret" MAXLENGTH="20" SIZE="10">
```

## <INPUT TYPE="HIDDEN">

Если задан атрибут HIDDEN, данные не могут быть изменены пользователем, так как поле <INPUT> не отображается на Web-странице. Поля типа HIDDEN часто применяются для сохранения данных в формах, состоящих из нескольких "экранов". Эти поля позволяют скрипту как бы "запоминать" предыдущие значения. В действительности данные каждый раз передаются на Web-сервер.

Подобно атрибуту PASSWORD, атрибут HIDDEN никак не шифрует данные. Содержимое полей HIDDEN также отображается в URL, когда для отправки формы применяется GET, и поэтому пользователь легко может изменить его, просто вписав другие данные в URL и повторно отправив форму. Поля HIDDEN не являются действительно скрытыми еще и потому, что пользователь может увидеть их данные, просматривая исходный код Web-страницы. Следовательно, в эти поля не надо помещать важные данные, такие как цены. Злонамеренный пользователь может сохранить страницу HTML на своем компьютере, изменить цены в скрытых полях и затем отправить заказ на основе этой страницы. Это все равно что позволить покупателям в магазине самим наклеивать на товары ярлычки с ценами — не самая удачная идея.

**Листинг 4.11. Скрытое поле ввода**

```
<INPUT TYPE="HIDDEN" NAME="widget" VALUE="Тест">
```

## VALUE

Во всех описанных до сих пор дескрипторах форм HTML можно также использовать атрибут VALUE, чтобы установить значение по умолчанию. Этот атрибут был указан в листинге 4.11 для скрытого поля ввода, но его так же можно было применить и

в примерах полей TEXT И PASSWORD. Значение, указанное в этом атрибуте, отображается при первой загрузке формы (в PASSWORD отображаются звездочки). Кроме того, если в форме имеется кнопка RESET, при ее нажатии восстанавливаются все значения по умолчанию. В целом, этот атрибут может быть очень полезен, чтобы снабдить форму HTML начальными значениями.

## <INPUT TYPE="CHECKBOX"> и <INPUT TYPE="RADIO">

Атрибут CHECKBOX позволяет представить пользователю группу пунктов (флажков), из которых он может сделать выбор. В случае CHECKBOX можно выбрать более одного пункта за раз.

"Родственник" атрибута CHECKBOX — атрибут RADIO. ОН также представляет пользователю для выбора группу пунктов (радиокнопок), но из группы может быть выбран только один пункт. Если пользователь помечает один пункт, со всех остальных пометка автоматически снимается.

Пункты CHECKBOX и RADIO группируются по одинаковому атрибуту NAME, присваиваемому каждому пункту в группе. Допускаются "группы" только из одного пункта, но, если создать группу из одного пункта RADIO, пользователь не сможет изменить однажды сделанный выбор, если только форма не имеет кнопки сброса (RESET). Пример флажков и радиокнопок приведен в листинге 4.12. В нем будет легче разобраться, если взглянуть на рис. 4.4, изображающий форму с этими элементами.

### Листинг 4.12. Флажки и радиокнопки

```
<INPUT TYPE= "CHECKBOX"
NAME="sports ">Хоккей<BR>
<INPUT TYPE="'CHECKBOX"
NAME="sports">Футбол<BR>
<INPUT TYPE= "CHECKBOX"
NAME="sports">Соккер<BR>
<P>
<INPUT TYPE="RADIO"
NAME="favorite">Scooby Doo<BR>
<INPUT TYPE="RADIO"
NAME="favorite">Shaggy<BR>
<INPUT TYPE="RADIO"
NAME="favorite">Velma<BR>
<INPUT TYPE="RADIO"
NAME="favorite">Scrappy Doo<BR>
<P>
<INPUT TYPE="CHECKBOX"
NAME="os">BeOS<BR>
<INPUT TYPE="CHECKBOX" NAME="os"
CHECKED>BSD<BR>
<INPUT TYPE="CHECKBOX" NAME="os"
CHECKED>Linux<BR>
<INPUT TYPE="CHECKBOX"
NAME="os">Windows<BR>
```

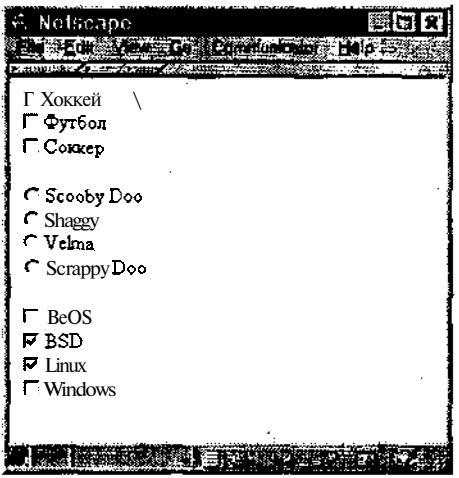



*Рис. 4.4. Форма с флажками и радиокнопками*

Обратите внимание, что флажки "Linux" и "BSD" уже установлены. В полях типа RADIO и CHECKBOX атрибут CHECKED служит для выбора пункта по умолчанию. Из группы радиокнопок можно выбрать только одну, поэтому атрибут CHECKED указывается только один раз. В группе флажков одновременно может быть выбрано по умолчанию несколько пунктов, как можно видеть на рис. 4.4.

## <INPUT TYPE="FILE">

Еще один тип поля ввода — поле FILE. ЭТО поле используется для загрузки файла. Оно отображается в виде поля ввода текста и кнопки с надписью Browse... (Обзор...) рядом с ним (рис. 4.5). Когда пользователь щелкает на кнопке Обзор..., открывается диалоговое окно выбора файла (рис. 4.6), в котором пользователь может выбрать файл для загрузки на сервер из своей системы. По соображениям безопасности поле ввода этого типа не может иметь значения по умолчанию. Загрузка файлов подробно рассматривается в гл. 7.

### Листинг 4.13. Поле загрузки файла

```
<INPUT TYPE="FILE" NAME="upload1">
```

Результат отображения этого кода в броузере показан на рис. 4.5.

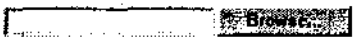



*Рис. 4.5. Поле ввода/обзора файлов*

На рис. 4.6 показано диалоговое окно, которое появляется, когда пользователь щелкает на кнопке Browse... (Обзор...).

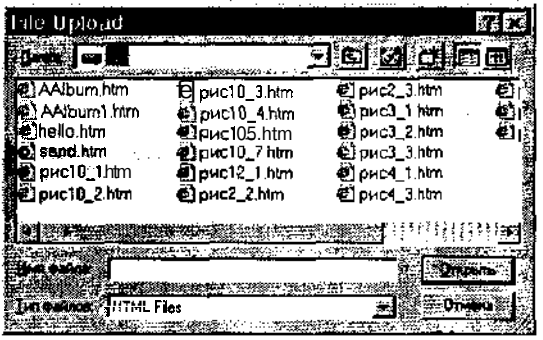



*Рис. 4.6. Диалоговое окно обзора файлов*

## <INPUT TYPE="SUBMIT"> и <INPUT TYPE="RESET">

Поля этих двух типов создают в форме HTML кнопки. Поле <INPUT TYPE="SUBMIT"> служит для отправки данных заполненной формы и запуска программы CGI, на которую эта форма указывает. По умолчанию для этой кнопки принят довольно странный текст: Submit Query (Подача запроса). С помощью атрибута VALUE можно изменить текст, который будет отображаться на ней.

*Маленький секрет:* пробелы в атрибуте VALUE ПОЗВОЛЯЮТ ДО некоторой степени управлять шириной кнопки. Например, в листинге 4.14 показан код кнопок на рис. 4.7. Обратите внимание на ширину второй кнопки. Ее удалось сделать больше, вставив по пять пробелов с каждой стороны слова *Проба*. Как и в любом другом поле <INPUT>, текст из атрибута VALUE передается в CGI, так что в форме может быть несколько кнопок SUBMIT с различными значениями и можно будет определить, на какой из них щелкнул пользователь. Если кнопка была сделана более широкой за счет пробелов, как в этом примере, следует помнить, что для программы CGI значения "Проба" и "  Проба  " различны. Не забудьте внести в код соответствующие изменения для обработки пробелов.

### Листинг 4.14. Поле отправки формы

```
<INPUT TYPE="SUBMIT" VALUE="Проба">
<BR>
<INPUT TYPE="SUBMIT" VALUE="      Проба      ">
```

Поле <INPUT TYPE="RESET"> отображается точно так же, как поле <INPUT TYPE="SUBMIT">, только на кнопке помещается текст Сброс. При щелчке на кнопке RESET все элементы формы HTML принимают значения по умолчанию. Если элемент не имеет значения по умолчанию, он очищается.

*Рис. 4.7. Кнопки отправки формы*

## <INPUT TYPE="IMAGE">

Поле <INPUT TYPE="IMAGE"> позволяет изображению выполнять функцию кнопки SUBMIT. При отправке этого элемента в CGI передаются данные name.x и name.y. x и y — это координаты точки изображения, на которой щелкнул пользователь, а name — имя самой кнопки.

### Листинг 4.15. Поле изображения

```
<INPUT TYPE=IMAGE" SRC="camel.jpg" NAME="mybutton" BORDER=0>
```

Этот код выводит изображение camel.jpg без отображения каймы. Если опустить атрибут BORDER, по умолчанию принимается ширина каймы вокруг изображения 1, что обычно нежелательно. Если пользователь щелкнет на этом изображении в точке с координатами 10,23, в программу CGI будет передано mybutton.x=10 и mybutton.y=23. После этого программа CGI может предпринять действия соответственно этим данным.

## <SELECT> И <OPTION>

Поля <SELECT> и <OPTION> не входят в состав дескриптора INPUT, как во всех предыдущих примерах. Напротив, это отдельные дескрипторы HTML. Они применяются для создания раскрывающегося списка или списка пунктов, которые может выбрать пользователь. Дескрипторы <OPTION> включаются в блок <SELECT>, создавая последовательный список вариантов. Пример приведен в листинге 4.16.

### Листинг 4.16. Возможность выбора в списке нескольких пунктов

```
<SELECT NAME="list1" MULTIPLE>
<OPTION>Linux</OPTION>
<OPTION>FreeBSD</OPTION>
<OPTION>MacOS</OPTION>
<OPTION>OS/2</OPTION>
<OPTION>BeOS</OPTION>
<OPTION>AIX</OPTION>
<OPTION>Windows</OPTION>
</SELECT>
<SELECT NAME="list2">
<OPTION>Linux</OPTION>
<OPTION>FreeBSD</OPTION>
<OPTION>MacOS</OPTION>
<OPTION>OS/2</OPTION>
<OPTION>BeOS</OPTION>
```

```
<OPTION>AIX</OPTION>
<OPTION>Windows</OPTION>
</SELECT>
```

В форме HTML этот листинг отображается, как на рис. 4.8.

Если сделать выбор, эти списки приобретут следующий вид.

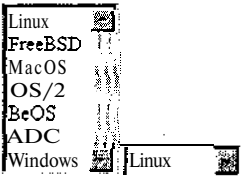



*Рис. 4.8. Два списка*

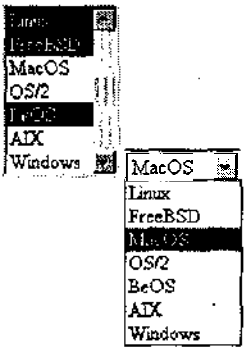



*Рис. 4.9. Списки с выбранными элементами*

Правый список на рис. 4.9 после выбора обычно закрывается, но мы оставили его раскрытым, чтобы показать, что происходит, когда пользователь выбирает пункт. Левый список позволяет выбрать несколько пунктов, тогда как в правом списке это невозможно. Чтобы выбрать несколько пунктов, пользователь должен удерживать клавишу <Shift> или <Ctrl> во время щелчка мышью. Чтобы сделать пункт выбранным по умолчанию при загрузке формы, укажите в открывающем дескрипторе <OPTION> атрибут SELECTED, как показано в листинге 4.17.

### Листинг 4.17. Выбор пункта по умолчанию

```
<OPTION>Linux</OPTION>
<OPTION SELECTED>FreeBSD</OPTION>
<OPTION>MacOS</OPTION>
```

## <TEXTAREA>

В заключение мы рассмотрим дескриптор <TEXTAREA>. ОН применяется для ввода больших фрагментов текста, таких как комментарии или сообщения в системе электронной почты, на базе Web. <TEXTAREA> требует закрывающего дескриптора </TEXTAREA>. Также в этом дескрипторе следует указывать атрибуты ROWS И COLS, чтобы броузер знал, сколько строк и колонок отображать.

Существует также атрибут WRAP, не упоминаемый в спецификациях HTML, но работающий как в Netscape, так и в Internet Explorer. Этот атрибут может иметь значения PHYSICAL и VIRTUAL. При указании PHYSICAL В конце каждой строки предполагается вставка символов перевода строки, тогда как VIRTUAL предполагает лишь перенос текста на следующую строку без каких-либо дополнительных символов. Слово *предполагает* мы употребляем потому, что этот атрибут не содержится в стандартах и нигде не указано, как броузеры должны осуществлять перенос текста. Атрибут WRAP рекомендуется использовать в большинстве случаев, потому что без него текст, вводимый пользователем, так и будет идти без переноса, по одной прямой (наподобие рекламного кролика на батарейках "с золотой каемкой"), пока пользователь не нажмет клавишу <Enter>. Если же указать атрибут WRAP, текст будет автоматически перенесен на следующую строку, когда он достигнет края текстового поля. Проиллюстрируем это на двух примерах.

### Листинг 4.18. Текстовая область

```
<TEXTAREA ROWS="4" COLS="40">
</TEXTAREA>
```


*Рис. 4.10. Текстовая область*

### Листинг 4.19. Автоматический перенос в текстовой области

```
<TEXTAREA ROWS="4" COLS="40" WRAP="PHYSICAL">
</TEXTAREA>
```

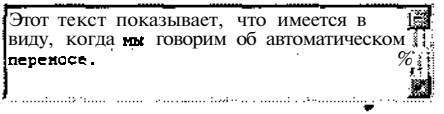



*Рис. 4.11. Текстовая область с переносом текста*

На рис. 4.10 текстовая область имеет как вертикальную, так и горизонтальную полосы прокрутки. Кроме того, в этом примере нельзя видеть весь текст целиком, хотя в обоих примерах был введен один и тот же текст. На рис. 4.11 горизонтальная полоса прокрутки исчезла, так что текстовая область занимает меньше места, и можно видеть весь введенный текст, так как он автоматически переносится на правом краю текстовой области.

Чтобы какой-нибудь текст появлялся по умолчанию, надо только ввести его между открывающим и закрывающим дескрипторами, как в листинге 4.20.

### Листинг 4.20. Текст по умолчанию в текстовой области

```
<TEXTAREA ROWS="4" COLS="40" WRAP="PHYSICAL">
Это текст по умолчанию.
</TEXTAREA>
```

Есть и другие дескрипторы и атрибуты, которые мы не рассмотрели. Однако они используются нечасто, и те, что мы описали, скорее всего, будут применяться более чем в 98% случаев, когда вы будете разрабатывать формы HTML. Чтобы узнать об остальных или изучить все эти дескрипторы и атрибуты более глубоко, вы можете обратиться к обширной документации на `http://www.w3.org`.

Хорошо продуманные и хорошо написанные формы производят приятное впечатление на пользователя. Пользователи, которым они понравились, могут заходить на сайт снова и даже сообщать о нем другим.

Постарайтесь сделать так, чтобы ваши формы были простыми, но полезными, и убедитесь, что они способны зафиксировать всю информацию о пользователе, которая вам нужна. Создать дружественный интерфейс не означает просто написать страницу с несколькими полями ввода. Это требует времени на проектирование и тщательную проработку.

# Чтение данных из формы с помощью CGI.pm

Значение CGI.pm для программистов Perl/CGI трудно переоценить. Линкольн Штайн оказал большую услугу сообществу Perl, написав и отладив этот модуль, чтобы им могли пользоваться все. CGI.pm берет на себя всю черную работу по анализу информации, поступающей из форм HTML, так что для программиста получить данные из формы становится не сложнее, чем вызвать функцию *param()*. Помните, как в главе 3 мы говорили о способах чтения данных из формы HTML? Мы рассмотрели очень примитивный способ чтения и разбора, и для учебных целей он прекрасно подходит. Но ведь мы программисты, а, как говорит Ларри Уолл, три основных черты программиста — лень, нетерпеливость и самомнение. Так что мы будем придерживаться этого определения и будем ленивыми. Пусть CGI.pm делает за нас всю черную работу, а мы сможем заняться более важными частями программы.

Листинг 4.21 показывает, как CGI.pm упрошает работу с формами.

### Листинг 4.21. Простая форма HTML

```
<HTML><HEAD>
  <TITLE>Пример формы</TITLE>
</HEAD>
<BODY>
<FORM NAME="form_example" ACTION="/cgi-bin/book/form1.cgi"
METHOD="POST">
Имя:     <INPUT TYPE="TEXT" NAME="f_name"><BR>
Фамилия: <INPUT TYPE="TEXT" NAME="l_name"><BR>
<BR>
Ваш любимый цвет:<BR>
  <INPUT TYPE="RADIO" NAME="color" VALUE="red">Красный<BR>
  <INPUT TYPE="RADIO" NAME="color" VALUE="blue">Синий<BR>
  <INPUT TYPE="RADIO" NAME="color" VALUE="green">Зеленый<BR>
  <INPUT TYPE="RADIO" NAME="color" VALUE="yellow">Желтый<BR>
<BR>
Ваш любимый вид спорта:<BR>
  <INPUT TYPE="CHECKBOX" NAME="sports" VALUE="hockey"
CHECKED>Хоккей<BR>
  <INPUT TYPE="CHECKBOX" NAME="sports" VALUE="football">Футбол<BR>
  <INPUT TYPE="CHECKBOX" NAME="sports" VALUE="baseball">Бейсбол<BR>
  <INPUT TYPE="CHECKBOX" NAME="sports"
VALUE="basketball">Баскетбол<BR>
  <INPUT TYPE="CHECKBOX" NAME="sports" VALUE="golf">Гольф<BR>
<BR>
<INPUT TYPE="SUBMIT" NAME="doit" VALUE="Отправить информацию">
</FORM>
</BODY>
</HTML>
```

Этот код HTML создает простую форму с несколькими полями ввода, показанную на рис. 4.12. Форма будет выглядеть не точно так, как показано, поскольку на рисунке уже введены данные и можно щелкнуть на кнопке Отправить информацию. А теперь мы посмотрим, насколько просто можно с помощью CGI.pm узнать, какая именно информация была введена.

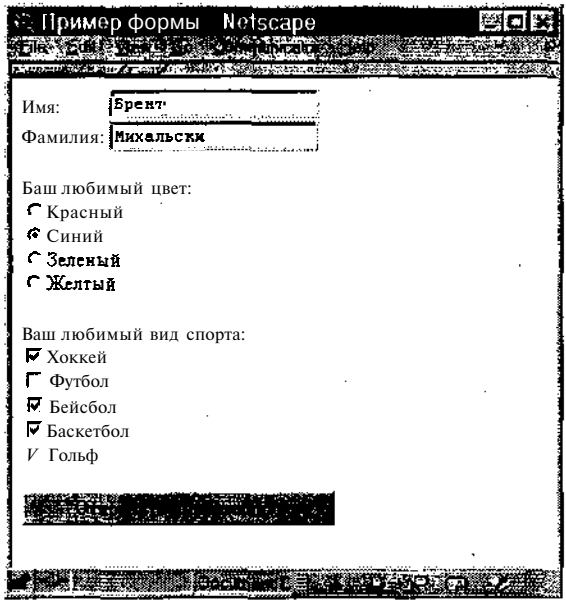



*Рис. 4.12. Заполненная форма*

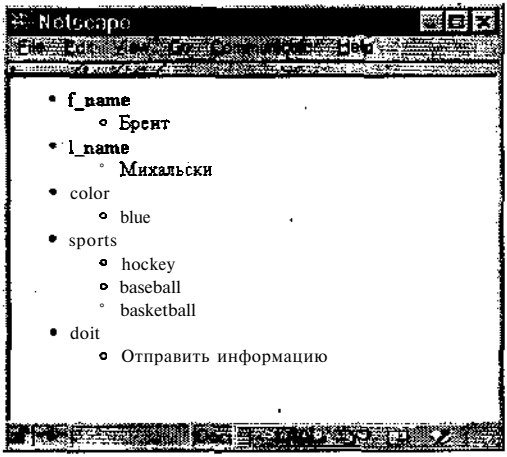



*Рис. 4.13. Результаты функции CGI.pm dump()*

На рис. 4.13 показан вывод программы CGI. Это просто имена полей, набранные полужирным шрифтом, которые сопровождаются списком значений, введенных пользователем. Если один и тот же атрибут NAME есть у нескольких элементов формы, после имени поля перечисляется несколько значений- Посмотрите на листинг 4.22. Это *полный* текст программы CGI, которая анализирует форму и выводит результаты, показанные на рисунке!

## Листинг 4.22. Вывод данных, переданных из формы

```
01:  #!/usr/bin/perl -wT
02:  use strict;
03:  use CGI qw(:standard);
```

```
04:  use CGI::Carp qw(fatalsToBrowser);
05:  print header;
06:  print dump();
```

Весь код анализа формы HTML и вывода ее результатов занял всего шесть строк Perl! Можно даже удалить строки use strict и CGI: :Carp, и программа останется работоспособной, так что фактически достаточно четырех строк кода. Этот пример доказывает, что CGI.pm избавляет программиста от рутинной работы и позволяет сосредоточиться на других деталях приложения, а не тратить время на разбор данных формы.

Функция CGI.pm *dump()* в предыдущем примере выводит все значения, которые были переданы в форме. Хотя *dump ()* — хорошее средство отладки, она не годится для написания реальных приложений CGI, которые будут видеть другие пользователи. Вершина полезности для приложений CGI — это функция *param ()*.

Назначение *param()* — просто получить данные, которые были отправлены из формы HTML, и прочитать их в программу CGI. Возьмем пример, который мы только что рассмотрели, но вместо *dump()* поставим функцию *param ()*. Мы получим намного более дружественную страницу.

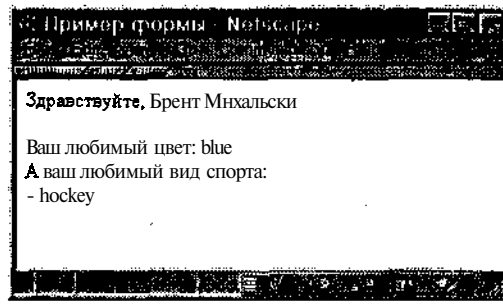



*Рис. 4.14. Динамический вывод — единственное число*

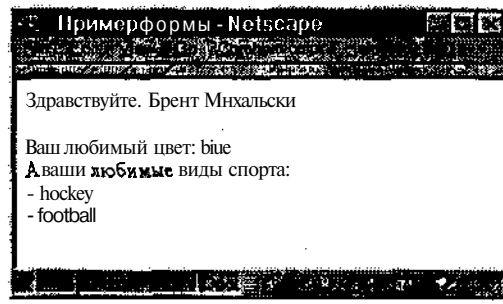



*Рис. 4.15. Динамический вывод — множественное число*

Замечаете различие между рис. 4.14 и 4.15? На рис. 4.14 выбран только один вид спорта, и фраза поставлена в единственном числе. На рис. 4.15 видов спорта несколько, и поэтому мы употребили множественное число. Внимание к подобным мелочам заставит ваше приложение выделиться среди других. Если оставить фразу статичной, пользователю может показаться, что вы, программист, поленились привести ее в порядок. Чтобы сделать это, требуется чуть больше поработать, но результат стоит того. Давайте исследуем код, который производит такой эффект, как показано на этих рисунках.

```
01: #!/usr/bin/perl -wT
02: use strict;
03: use CGI qw(:standard);
04: use CGI::Carp qw(fatalsToBrowser);
```

**Строки** 1–4 мы применяем почти в каждой программе CGI этой книги, и делаем это по серьезным причинам.

**Строка 1** сообщает системе, где найти Perl, и включает предупреждения (-w) и проверку на загрязнения (т). Проверка на загрязнения — это мера безопасности, которая предохраняет от выполнения команд с опасными данными, полученными от пользователя. При включенных предупреждениях программист получает от интерпретатора больше сообщений о возможных ошибках в программе. Включать предупреждения рекомендуется, так как чем более подробно реагирует интерпретатор, когда вы разрабатываете программу, тем лучше. Это также предотвращает досадные ошибки.

Строка 2 включает строгий синтаксис. Этот режим добавляет еще больше мер защиты от обычных ошибок, таких как присвоение одинаковых имен двум переменным или использование текста без кавычек.

Строка 3 загружает модуль `CGI.pm` и импортирует стандартный набор функций.

**Строка** 4 загружает модуль CGI: :Carp и импортирует метод *fatalsToBrowser()*. Благодаря этому большая часть программных ошибок будет отображаться в броузере в виде соответствующих сообщений, а не стандартного сообщения об ошибке сервера 500.

```
05: my $first_name = param ('f_name');
06: my $last_name = param('l_name');
07: my $fav_color = param ('color');
```

В строках 5–7 создается несколько переменных и им присваиваются значения, прочитанные функцией `CGI.pm` *param()* из формы HTML. Имя, которое указано в параметре функции — это имя элемента формы, для которого требуется получить данные. Эти имена элементов формы чувствительны к регистру, так что проверьте их написание.

*param()* — чрезвычайно гибкая и мощная небольшая функция. Существует четыре различных способа использования *param()*.

- `$scalar = param('имя_элемента');`

Если возвращаемые значения читаются в скаляр, как в строках 4–7, *param()* возвращает одиночное значение, и именно оно сохраняется в скалярной переменной. Если в форме имеется несколько элементов с этим именем, функция *param()* возвращает первое, и только первое значение.

- `@array = param('имя_элемента');`

Если возвращаемые значения читаются в массив, а не в скаляр, функция *param()* возвращает все значения и сохраняет их в массиве. Если это была группа одноименных флажков или список с выбором нескольких элементов, в массив возвращаются все выбранные элементы.

- `$scalar = param();`

Если вызвать пустую функцию *param()*, не указывая параметр, функция сохранит в скалярной переменной число имен элементов формы.

- `@array = paramO;`

Если вызвать пустую функцию *param()* и прочитать возвращаемое значение в массив, массив будет заполнен *именами* всех элементов формы.

```
08:  my @fav_sports = param ('sports');
```

В строке 8 читаются все пункты, которые были помечены *в* группе флажков "ваш любимый вид спорта" в форме HTML.

```
09:  my $sport_count = @fav_sports;
10: ray $sport_text;
```

**Строка 9** сохраняет количество элементов массива `@fav_sports` в переменной `$sport_count`.

**В строке 10** просто объявляется переменная `$sport_text`. Так как в программе используется прагма `strict`, все переменные до своего применения должны быть объявлены.

```
11:  print header;
12:  print start_html ('Пример формы');
```

**В строке 11** вызывается функция `CGI.pm` *header()*. Модуль `CGI.pm` применяется почти для каждой задачи программирования CGI, а не только для получения значений из форм HTML. Функция `header()` возвращает заголовок **HTTP,** в данном случае "Content-type: text/html\n\n". Следовательно, **строка 11** просто выводит данные заголовка.

**В строке 11** вызывается функция `CGI.pm` *start_html()*. Эта функция возвращает текст, необходимый для начала страницы HTML. Листинг 4.23 фактически повторяет данные, выводимые **строкой 12.**

## Листинг 4.23. Типичный заголовок страницы HTML

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>Пример формы</TITLE>
</HEAD><BODY>
```

Намного проще запомнить **строку 12,** чем весь текст из этого листинга.

```
13:  print qq (Здравствуйте, $first_name $last_name<P>);
14:  print qq(Ваш любимый цвет: $fav_color<BR>);
```

**Строки 13–14** просто выводят некоторый текст, который отображается на готовой странице. Помещение текста в *qq()* равносильно заключению его в двойные кавычки, но при этом не приходится удалять или заменять кавычки внутри текста. Так как HTML обычно содержит много кавычек, функция *qq()* очень полезна.

```
15:  if $sport_count > 1) {
16:    $sport_text = "ваши любимые виды спорта";
17:  }
18:  else {
19:    $spqrt_text = "ваш любимый вид спорта";
20:  }
```

**Строки 15—20** — оператор `if...else`.

В **строке 15** проверяется, больше ли переменная `$sport_count` единицы. Если это так, начинается блок кода.

**Строка 16** — то, что выполняется, если условие в **строке 15** истинно. Здесь переменной `$sports_text` присваивается текстовая строка "ваши любимые виды спорта".

**Строка 17** закрывает блок `if`.

**Строка 18** — оператор `else`, в которой начинается блок кода, выполняемый, если условие ложно.

В строке 19 переменной `$sports_text` присваивается значение "ваш любимый вид спорта".

**Строка** 20 завершает оператор if. . .else. **Строки 15-20** устанавливают отображаемые слова в единственное или во множественное число, что выглядит так, как будто программист сам занимался изменением этого текста.

```
21:  print qq(A $sport_text:<BR>);
```

**Строка 21** выводит поясняющее предложение перед видами спорта, которые выбрал пользователь.

```
22:  foreach(@fav_sports) {
23:    print qq(- $_<BR>);
24:  }
```

**Строки 22—24** — цикл, в котором выводится каждый элемент массива `@fav_sports`. Цикл `foreach` проходит каждый элемент массива, присваивая его значение переменной `$_`.

**Строка 23** выводит текущее значение.

**Строка 24** закрывает блок `foreach`.

```
25: print end_html ();
```

**Строка 25** — последняя строка программы; в ней вызывается еще одна функция `CGI.pm`. Функция *end_html()* просто возвращает текст `</BODY></HTML>`. Этот текст запомнить не очень трудно, но иногда еще проще запомнить эту дополнительную функцию. Вы сани можете выбрать структуру и формат выводимого текста, так что помните, что этот всегда можно сделать несколькими способами.

# Как добиться того, чтобы пользователь был доволен

У пользователей, которые посещают ваш сайт, должно остаться приятное впечатление от него. Это все равно, как если бы они посетили обычный, не виртуальный магазин. Когда в реальном магазине вас вежливо обслуживают, вероятность, что вы получите от этого магазина хорошее впечатление и, может быть, зайдете еще раз, становится большей. Это верно и для Web-сайтов: вы должны добиться того, чтобы ваши пользователи получили удовольствие. Страница с благодарностью никогда не мешает, так что постарайтесь, чтобы пользователи, отправив заполненную форму, получали какой-то позитивный отклик. Дайте им знать, что форма принята успешно, обратитесь к ним по имени и покажите, что было передано. Если пользователь покидает ваш сайт довольным, больше вероятность, что вы получите постоянного посетителя.

# Заключительный пример

Мы проработали в этой главе массу материала, но нужен еще один, последний пример, чтобы вы получили еще немного опыта перед тем, как отправитесь в мир программирования CGI. В этом примере мы соберем такую информацию, как имя пользователя, адрес и другие данные. Затем дадим пользователю возможность проверить данные, причем введенную информацию будем хранить в скрытых полях. После проверки пользователь сможет щелкнуть на кнопке и "отправить" информацию. В действительности форма ничего не будет делать с данными (это тема следующих глав), но вы увидите простой пример использования формы для сбора информации и последующей ее проверки пользователем перед обработкой.

```
01:  #!/usr/bin/perl -Tw
02:  # form3.cgi
03:  use strict;
04:  use CGI qw(:standard);
05:  use CGI::Carp qw(fatalsToBrowser);
```

Строка 1 сообщает системе, где искать Perl, и включает проверку на загрязнение и предупреждения.

Строка 2 — комментарий с именем программы.

Строка 3 предписывает Perl использовать прагму `strict`.

Строка 4 импортирует модуль CGI со стандартными функциями.

Строка 5 загружает модуль `CGI::Carp`.

Строки 3–5 будут присутствовать почти в каждой нашей программе.

```
06:  print header;
```

Строка 6 вызывает функцию `CGI.pm` *header (),* которая выводит стандартный заголовок HTTP.

```
07:  my ($F_Name, $L_Name, $review, $Address,
08:     $Eye_Group, $Eye_Color, $City, $State,
09:     $Zip, $Page_Title, $Buttons, $Butt_Val,
10:     $Hidden);
```

Строки 7–10 — на самом деле одна "строка" Perl, в которой объявляется несколько глобальных переменных. При использовании прагмы `strict` все переменные должны быть объявлены прежде, чем они могут использоваться.

```
11:  Initialize_Values();
12:  Print_Form();
```

Строка 11 вызывает подпрограмму *Initialize_Values().* Эта подпрограмма получает данные из формы HTML и выполняет все другие задачи, которые должны быть завершены перед выводом страницы.

Строка 12 вызывает подпрограмму *Print_Form(),* которая выводит страницу HTML.

```
13:  sub Initialize_Values {
```

В строке 13 начинается подпрограмма *Initialize_Values()*

```
14:     $Butt_Val   = param('sendit')
15:     $review     = param('review')
16:     $F_Name     = param('f_name')
17:     $L_Name     = param('l_name')
18:     $Address    = param('address');
19:     $City       = param('city');
20:     $State      = param('state');
21:     $Zip        = param('zip');
22:     $Eye_Color  = param('eyecolor');
```

В строках 14–22 при помощи функции *param()* из `CGI.pm` данные, переданные с Web, считываются в скалярные переменные.

```
23:     if ($Butt_Val =~ /Редактировать данные/){ $review = ""; }
24:    if (($Butt_Val =~ /Отправить данные/) &&  (@review > 0)) {
25:       Print_Thanks();
26:     }
```

Строка 23 проверяет, содержит ли $Butt_Val строку "Редактировать данные". Если это так, значит пользователь щелкнул на кнопке Редактировать данные, и переменной $review надо присвоить пустую строку.

**Строка 23** проверяет, содержит ли $Butt_Val строку "Отправить данные", *и* больше ли значение $review нуля.

**Строка 25** выполняется, если условие в **строке 24** соблюдено. В ней вызывается подпрограмма *Print_Thanks(),* которая выводит благодарность пользователю.

**Строка 26** завершает блок if, начатый в **строке 24.**

```
27:     $F_Name   = Text_Field ("f_name",   20, 20, $F_Name);
28:     $L_Name   = Text_Field("l_name",    20, 20, $L_Name);
29:     $Address  = Text_Field("address",   20, 20, $Address);
30:     SCity     - Text_Field("city",      20, 20, SCity);
31:     SState    = Text_Field("state",      2,  3, $State);
32:     $Zip      = Text_Field("zip",       10, 12, $2ip);
```

**В строках 27—32** при помощи функции *Text_Field()* формируется тип поля, которое будет отображено. Если программа выполняется впервые или если пользователь желает обновить свои данные, поля будут полями ввода текста. Если пользователь только просматривает свои данные, поля недоступны для редактирования. В функцию *Text_Field()* передаются имя поля, максимальная длина данных, ширина и значение по умолчанию, а функция возвращает отображенный текст.

```
33:     if(!$review) {
```

**Строка 33** начинает блок if...else. Заданное здесь условие выполняется, если мы не будем пересматривать данные **("если-*не*-$review").** Иными словами, мы проверяем, содержит ли $review значение. Если да, мы переходим к блоку else в **строке 44. В** противном случае выполняется следующий блок кода.

```
34:     $Eye_Group = radio_group( -name    => 'eyecolor',
35:                               -default => $Eye_Color,
36:                               -values  => ['blue','green',
37:                                 'brown','grey','red'],
38:                              );
```

**Строки 34—38—** это один оператор Perl, в котором вызывается функция CGI.pm *radio_group()* и формируется группа радиокнопок. По умолчанию принимаются значения, полученные с вызывающей страницы **HTML.** Передаваемые данные — это значения, которые отображаются для пользователя и помешаются в поле VALUE= в форме.

```
39:     $Buttons = submit( -name => "sendit",
40:                        -value => "Отправить данные");
```

**В строках 39—40** с помощью функции CGi.pm submit() генерируется код **HTML** для кнопки отправки форм. Возвращаемое значение сохраняется в $Buttons.

```
41:     $Hidden = ""; # Очистка $Hidden
42:     $Page_Title = "Введите ваши данные";
43:   }
```

**Строка 41** очищает все данные, которые находились в переменной SHidden.

**Строка 42** задает заголовок страницы.

**Строка 43** закрывает этот раздел блока if. . .else.

```
44:   else {
```

**Строка 44** начинает следующий раздел блока if. . .else.

```
45:     $Eye_Group = $Eye_Color;
46:     $Hidden .= hidden(-name => "eyecolor",
47:                       -value => $Eye_Color);
```

**Строки 46—47** добавляют новые данные в переменную $Hidden. **В строке 46** вызывается функция *hidden ()*, которая генерирует код HTML для скрытой переменной формы и сохраняет в ней информацию о цвете глаз.

```
48:    $Buttons = submit  (-name => "sendit",
49:                        -value => "Редактировать данные");
50:    $Buttons .= submit(-name => "sendit",
51:                       -value => "Отправить данные");
```

**В строках 48—51** создаются две кнопки, необходимые для страницы пересмотра данных. Здесь применяется та же функция *submit()*, что использовалась в строке 39.

```
52:     $Page_Title =
       "Исправьте данные или нажмите кнопку 'Отправить данные'";
53:     }
54:  }
```

**Строка 52** создает заголовок страницы.

**Строки 53—54** закрывают соответственно блок if... else и подпрограмму.

```
55:  sub Print_Thanks {
56:    print <<HTML;
```

**Строка 55** начинает подпрограмму *Print_Thanks()*. Эта подпрограмма выводит простую страницу с благодарностью, чтобы пользователь знал, что его форма принята.

**Строка** 56 начинает включенный документ, в котором выводится страница с благодарностью.

```
57:     <HTML><HEAD><TITLE>Спасибо!</TITLE></HEAD>
58:     <BODY>
59:     <CENTER><B>Спасибо, $F_Name!!!<P>
60:     Ваши данные очень важны для нас!
61:     </B></CENTER>
62:     </BODY></HTML>
```

**Строки 57—62** — просто код HTML, образующий страницу с благодарностью. Обратите внимание на **строку 59:** здесь мы выводим имя пользователя ($F_Name), чтобы страница имела более "направленный" вид.

```
63:     HTML
```

**Строка 63** завершает включенный документ, начатый в **строке 56.**

```
64:     exit;
65:  }
```

**Строка 64** производит выход из программы. В этой подпрограмме нам надо просто вывести страницу и выйти из программы. Это надо сделать здесь, так как иначе программа продолжит выводить данные, что собьет пользователя с толку.

**Строка 65** закрывает подпрограмму.

```
66:  sub Print_Form {
67:    print <<HTML;
```

**Строка 66** начинает подпрограмму *Print_Form ()*.

**Строка 67** начинает включенный документ, в котором выводится страница HTML.

```
68:     <HTML><HEAD>
69:     <TITLE>$Page_Title</TITLE>
70:     </HEAD>
```

В **строках 68—70** начинается код HTML и выводится заголовок страницы. Обратите внимание, что здесь мы не используем функцию *start_html()*. Можно было бы ее вызвать, но в Perl и это можно сделать... Дальше вы знаете.

```
71:     <BODY>
72:     <FORM NAME="form_example" METHOD="POST">
73:     <INPUT TYPE="HIDDEN" NAME="review" VALUE="1">
74:     $Hidden
```

**В строках 71—74** продолжается вывод **HTML. В строке 74** мы помещаем в HTML все "скрытые" переменные, чтобы можно было сохранять данные при переходе от одной страницы к другой.

```
75:     <TABLE BORDER=1 CELLSPACING=0>
76:     <TR>
77:     <TD COLSPAN=2 ALIGN=CENTER>
78:     <B>$Page_Title</B>
79:     </TD>
80:     </TR>
81:     <TR>
82:     <TD>Имя:</TD><TD>$F_Name</TD>
83:     </TR>
84:     <TR>
85:     <TD>Фамилия:</TD><TD>$L_Name</TD>
86:     </TR>
87:     <TR>
88:     <TD>Адрес:</TDxTD>$Address</TD>
89:     </TR>
90:     <TR>
91:     <TD>Город:</TD><TD>$City</TD>
92:     </TR>
93:     <TR>
94:     <TD>Штат/почтовый индекс:</TDxTD>$State $Zip</TD>
95:     </TR>
96:     <TR>
97:     <TD>Цвет глаз:</TD><TD>$Eye_Group</TD>
98:     </TR>
99:     <TR>
100:    <TD COLSPAN=2 ALIGN=CENTER>
101:    $Buttons
102:    </TD>
103:    </TR>
104:    </TABLE>
105:    <BR>
106:    </FORM></BODY></HTML>
```

**Строки 75—106** — код HTML таблицы, содержащей названия и данные полей формы.

```
107:    HTML
108: }
```

**Строка 107** завершает включенный документ, в котором выводится страница.

**Строка 108** завершает подпрограмму.

```
109: sub Text_Field {
110:   my ($name, $max, $size, $value) = @_;
111:   my $field;
```

**В строке 109** начинается подпрограмма *Text_Field()*. Эта подпрограмма генерирует HTML или текст, необходимый для отображения текстовых полей в форме **HTML.**

**Строка ПО** считывает все данные, переданные в подпрограмму, и сохраняет их в новых переменных.

**Строка 111** объявляет скалярную переменную.

```
112:     SHidden .= hidden( -name => $name, -value => $value);
```

Строка **112** дописывает новые данные в переменную $Hidden. Так как через эту подпрограмму передаются всё текстовые поля, код, генерирующий скрытые поля, избавляет от ввода массы текста вручную.

```
113:     if (!$review){
114:       field = textfield( -name      => $name,
115:                          -maxlength => $max,
116:                          -size      => $size,
117:                          -value     => $value);
118:     }
```

В строке **113** проверяется, содержит ли $reviewed какие-либо данные. Если нет, начинается первая часть блока if. . .else.

В строках **114—117** с помощью функции *textfield* генерируется необходимый текст.

Строка **118** закрывает блок if. . .else.

```
119:     else {
120:       $field = $value;
121:     }
```

Строки **119—121** — вторая часть блока if. . .else, начатого в строке И3. Если $review *содержит* данные, поля ввода не отображаются, а просто Sfield присваивается значение $value.

```
122:     return ($field);
123:   }
```

В строке 122 подпрограмма возвращает скаляр $field.

Строка 123 завершает подпрограмму.

Вот мы и закончили этот пример. Он должен помочь вам, когда вы начнете писать приложения CGI, которые могут откликаться на ввод пользователя, а также позволяют ему пересматривать введенные данные перед обработкой. Операция пересмотра очень важна для коммерческих приложений. Если удастся, пусть даже на несколько процентов, сократить ошибки при вводе, вы сможете сэкономить деньги, поскольку придется исправлять меньше данных.

Проектируя формы HTML, старайтесь, чтобы они были просты, но эффективно фиксировали нужную информацию. Лучший способ добиться этого — рассчитать все заранее. Чем более продуманным будет ваш пользовательский интерфейс, тем он будет лучше. Не забывайте про обратную связь с пользователем! Пользователям нравится, когда они чувствуют теплое "отношение" приложения к себе.

# Упражнения

- Создайте форму HTML с полем ввода текста, текстовой областью, группой радиокнопок, группой переключателей, раскрывающимся списком, скрытым полем и полем пароля. Напишите приложение CGI, которое будет читать и отображать всю введенную информацию.
- Напишите приложение CGI, которое читает числа и выводит их сумму.
- Напишите программу с полями ввода текста и подсказками, что надо ввести, например, "существительное", "глагол" или "цвет". Пусть программа читает введенные слова и подставляет их в какой-нибудь связный текст, например, рассказ. Иногда получаются забавные результаты.

# Листинги

**Листинг 4.24. Чтение и представление данных из формы**

```
01:  #/usr/bin/perl -Tw
02:  #
03:  use strict;
04:  use CGI qw(:standard);
05:  use CGI::Carp qw(fatalsToBrowser);
06:  print header;
07:  my ($F_Name, $L_Name, $review, $Address,
08:      $eyecolor, $City, $color, $State,
09:      $Title, $Buttons, $Butt_Val,
10:      $Hidden);
11:  Initialize_Values();
12:  Print_Form();
13:  sub Initialize_Values {
14:    $Butt_Val = param('sendit')
15:    $review   = param('review')
16:    $F_Name   = param('f_name')
17:    $L_Name   = param('l_name')
18:    $Address  = param('address');
19:    $City     = param('city');
20:    $State    = param('state');
21:    $Zip      = param('zip');
22:    $Eye_Color = param('eyecolor');
23:    if ($Butt_Val =~ /Редактировать данные/){ $review = ""; }
24:    if (($Butt_Val =~ /Отправить данные/) && (@review > 0)) {
25:      Print_Thanks();
26:    }
27:    $F_Name   = Text_Field("f_name",   20, 20, $F_Name);
28:    $L_Name   = Text_Field("l_name",   20, 20, $L_Name);
29:    $Address  = Text_Field("address", 20, 20, $Address);
30:    $City     = Text_Field("city",     20, 20, $City);
31:    $State    = Text_Field("state",     2,  3, $State);
32:    $Zip      = Text_Field("zip",      10, 12, $Zip);
33:    if(!$review) {
34:    $Eye_Group = radio_group ( -name    => 'eyecolor',
35:                               -default => $Eye_Color,
36:                               -values  => ['blue','green',
37:                                'brown','grey','red'],
38:                             );
39:    $Buttons = submit( -name => "sendit",
40:                       -value => "Отправить данные");
41:    $Hidden = ""; # Очистка $Hidden
42:    $Page_Title = "Введите ваши данные";
43:    }
44:  else {
45:    $Eye_Group = $Eye_Color;
46:    $Hidden .= hidden(-name => "eyecolor",
47:                      -value => $Eye_Color);
48:    $Buttons = submit(-name => "sendit",
49:                      -value => "Редактировать данные");
50:    $Buttons .= submit(-name => "sendit",
51:                       -value => "Отправить данные");
52:    $Page_Title =
     "Исправьте данные или нажмите кнопку 'Отправить данные'";
```

```
53:     }
54:   }
55:   sub Print_Thanks {
56:     print «HTML;
57:     <HTML><HEAD><TITLE>Спасибо!</TITLE></HEAD>
58:     <BODY>
59:     <CENTER><B>Спасибо, $F Name! ! !<P>
60:     Ваши данные очень важны для нас!
61:     </B></CENTER>
62:     </BODY></HTML>
63:     HTML
64:     exit;
65:   }
66:   sub Print_Form {
67:     print «HTML;
68:     <HTML><HEAD>
69:     <TITLE>$Page_Title</TITLE>
70:     </HEAD>
71:     <BODY>
72:     <FORM NAME="form_example" METHOD="POST">
73:     <INPUT TYPE="HIDDEN" NAME="review" VALUE="1">
74:     $Hidden
75:     <TABLE BORDER=1 CELLSPACING=0>
76:     <TR>
77:     <TD COLSPAN=2 ALIGN=CENTER>
78:     <B>$Page_Title</B>
79:     </TD>
80:     </TR>
81:     <TR>
82:     <TD>Имя:</TD><TD>$F_ Name</TD>
83:     </TR>
84:     <TR>
85:     <TD>Фамилия:</TD><TD>$L_Name</TD>
86:     </TR>
87:     <TR>
88:     <TD>Адрес:</TDxTD>$Address</TD>
89:     </TR>
90:     <TR>
91:     <TD>Город:</TD><TD>$City</TD>
92:     </TR>
93:     <TR>
94:     <TD>Штат/почтовый индекс:</TD><TD>$State $Zip</TD>
95:     </TR>
96:     <TR>
97:     <TD>Цвет глаз:</TDxTD>$Eye_Group</TD>
98:     </TR>
99:     <TR>
100:     <TD COLSPAN=2 ALIGN=CENTER>
101:     SButtons
102:     </TD>
103:     </TR>
104:     </TABLE>
105:     <BR>
106:     </FORM></BODY></HTML>
107:     HTML
108:   }
109:   sub Text_Field {
110:     my(Sname, $max, $size, $value) = @_;
```

```
111:    my $field;
112:    $Hidden .= hidden( -name => $name, -value => $value);
113:    if (!$review){
114:      field = textfield( -name      => $name,
115:                         -maxlength => $max,
116:                         -size      => $size,
117:                         -value     => $value);
118:    }
119:    else {
120:      $field = $value;
121:    }
122:     return ($field);
123:  }
```


# Работа с cookie

## Введение

Сорт домашнего печенья — cookie, о котором мы сейчас поговорим, не упоминается в поваренной книге миссис Филдс. Эти cookie — маленькие фрагменты данных, которые передаются в заголовке HTTP и хранят информацию о пользователе. Cookie применяются в броузерах уже давно. В Netscape они появились еще в версии броузера 1.x.

Но, если cookie содержат информацию, почему они так по-кулинарному называются? Объяснение термину *cookie* было дано Лу Монтулли (Lou Montoulli), главным разработчиком протоколов в подразделении Netscape, которое занималось разработкой клиентов. Согласно Монтулли, cookie — это "известный термин информатики, который используется при описании "непрозрачного" фрагмента данных, хранимого посредником. Это определение точно соответствует использованию термина; просто это слово не применяется за пределами компьютерных кругов". Вот поэтому и было выбрано слово *cookie*.

Так как среда Web не поддерживает состояний, каждое посещение пользователем сайта рассматривается как первое посещение. Броузер подключается, получает необходимую информацию, отображает ее и затем отключается. Между броузером и сервером нет постоянного сообщения. Но если броузер и сервер никак не взаимодействуют, как тогда удается некоторым сайтам *запоминать* информацию о пользователях, когда они возвращаются на сайт? Ответ прост: через cookie!

Cookie — очень полезный для Web-программиста инструмент, позволяющий, например, сохранять товары в "тележке для покупок" в виртуальных магазинах, запоминать имя пользователя, его предпочтения или даже хранить данные регистрации пользователя. Кстати, последний способ — сохранение в cookie регистрационной информации — не рекомендуется, так как она никак не шифруется и не защищается. Но, несмотря на это, многие сайты записывают в cookie имя пользователя и пароль, так что пользователю не надо каждый раз вводить их при входе на сайт.

# Безопасность

Из того, что cookie непригодны для хранения регистрационных данных, вовсе не следует, что они перестают быть очень простым и эффективным способом хранения любой другой пользовательской информации. Через cookie невозможно передать вирус, так как они представляют собой простые текстовые файлы, которые не выполняются и не могут выполняться. Кроме того, cookie доступны только для сервера или домена, в котором они были установлены, что дает дополнительную безопасность, препятствуя неправомочным сайтам просматривать данные cookie, записанные другими сайтами.

# Ограничения

Спецификация cookie устанавливает минимально допустимые возможности.. Все Web-броузеры, поддерживающие cookie, обеспечивают *по крайней мере* эти минимальные возможности. Web-броузеры могут поддерживать cookie и в более широких пределах, но они должны обеспечивать этот минимум, так что программисту есть из чего исходить при разработке приложений, использующих cookie. Программист должен иметь в виду, что пользователь может отключить cookie или что Web-броузер не будет принимать cookie по какой-то другой причине.

- Общее количество cookie — 300
- Размер cookie — до 4 килобайт
- До 20 cookie на каждый сервер или домен

Когда одно из этих ограничений, например, общее число cookie, превышается, самое старое cookie стирается и в файл cookie записываются новые данные. Если размер cookie превосходит допустимый (4 Кбайт), cookie может быть усечено до этого размера.

Cookie загружаются в память при запуске броузера и записываются на диск, когда броузер завершает работу, а cookie еще не потеряло силы. Можно заметить, что, если в системе устанавливается новое cookie и броузер затем терпит сбой, значение нового cookie теряется. Причина этого в том, что броузер не завершает работу нормально и поэтому не дает возможности записать cookie на диск.

# Составные части cookie

Cookie состоит из шести различных частей. Эти части не называются крошками!

- *Имя* cookie
- *Значение* этого cookie
- *Срок действия*
- Информация о *пути*
- *Домен,* для которого действительно cookie
- *Безопасность* (использует ли сервер SSL)

## Имя

Имя — единственный обязательный элемент cookie. Имя и значение, как правило, устанавливаются вместе, но при очистке содержимого cookie элемент значения обычно остается пустым.

### Значение

Значение — вторая половина пары имя-значение. Значение — это текст, который хранится в cookie. Фактически, пара имя-значение в cookie очень похожа на пару имя-значение в хэше Perl. Рекомендуется, чтобы текст был в кодировке URL: пробелы заменены на эквиваленты %xx, но это не обязательно.

### Срок действия

Срок действия — дата, когда срок годности cookie истекает. Если срок действия не указан, cookie исчезает, когда броузер завершает работу. Срок обьтчно имеет следующий формат:

```
'День_недели, DD-MMM-YYYY HH:MM:SS GMT
```

Этот формат даты обязателен. Если дата установлена неправильно, броузер либо проигнорирует ее и удалит cookie при выходе, либо cookie не будет установлено вообще из-за ошибки формата даты. Позже мы рассмотрим несколько приемов, позволяющих сделать срок действия "начинкой" cookie. *Подсказка:* `CGI.pm`.

### Домен

На основании поля домена броузер определяет, должна ли текущая загружаемая страница послать cookie на сервер. Если при установке cookie домен не указан, по умолчанию принимается имя сервера, который генерирует отклик. Сервер может устанавливать cookie только для домена, в котором он находится.

Один важный факт, на который следует обратить внимание: при указании домена требуется *по крайней мере* две точки. Чем это вызвано? Если бы cookie было разрешено устанавливать серверу с доменом, например, `.com`, это означало бы, что cookie будет глобально доступно для любого адреса . com.

```
Допускается: domain=.perlguy.net
Не допускается: domain=.perlguy.net  # Только одна точка!
```

### Путь

Путь позволяет еще сильнее ограничить круг страниц, которые могут видеть cookie. Если задать путь, например, /foo, то cookie будет отправлено, только если загружаемая страница находится в каталоге /foo. Это означает, что если пользователь запросит страницу в корневом домене или в домене `/blech`, cookie не будет выдано.

Интересно, что заданное значение пути определяет не точный путь, в котором должна находиться страница, а скорее шаблон соответствия или регулярное выражение. Например, если в cookie задан путь /foo, cookie будет отправлено в ответ на запросы не только из иерархии /foo, но и из каталогов /foobar и /foowhatever! Запомните это, когда будете работать с cookie на своем сайте.

### Безопасность

Если cookie помечено как безопасное, оно может передаваться только по безопасному методу, такому как SSL. Если безопасность не задана, по умолчанию для cookie разрешается передача по незащищенным методам.

## Работа с cookie вручную

Так как работа с cookie вручную требует определенного искусства и существуют методы лучше, мы рассмотрим только один простой пример и сразу перейдем к этим более легким методам. Скрипты в этом разделе довольно невелики, поэтому мы будем сразу приводить весь код, а затем объяснять его (см. листинг 5.1).

## Листинг 5.1. Установка cookie вручную

```
01: #!/usr/bin/perl -wT
02:  # Пример 5-1
03:  use strict;
04:  print qq(Content-type: text/html\n);
05:  print qq(Set-Cookie: username=Fred Flintstone; );
06:  print qq(expires=Mon, 01-Jan-2001 00:00:00 GMT; );
07:  print qq(path=/\n\n);
08:  print qq(В вашем броузере установлено cookie...<P>);
09:  print qq(<A HREF="example_5-2.cgi">);
10:  print qq(Щелкните, чтобы увидеть cookie</A>);
```

**Строка 1** — обычная начальная строка в наших примерах. Она сообщает системе, где найти Perl, и включает проверку на загрязнение и предупреждения.

**Строка 2** — просто комментарий, чтобы можно было сразу узнать, какая это программа.

**Строка 3** включает строгий синтаксис. Хотя в этой программе не используются никакие функции или переменные, и строгий синтаксис на самом деле не нужен, программирование с включенной прагмой `strict` — хорошая привычка, которую рекомендуется приобрести. Поэтому мы включаем строгий синтаксис для всех программ вне зависимости, нужно им это или нет.

**Строка 4** выводит тип заголовка HTTP. Генерируемый нами документ имеет тип `text/html`. Однако обратите внимание, что вместо двух символов конца строки здесь стоит только один. Это вызвано тем, что следующая строка — cookie — также входит в состав заголовка. Два конца строки (пустую строку) нельзя передавать, пока заголовок не будет сгенерирован полностью. Если забыть об этом и поместить два конца строки в **строку** 4 вместо **строки 5,** в которой устанавливается cookie, данные cookie будут просто выведены в броузер. Попробуйте и посмотрите, что получится.

В **строках 5—7** устанавливается собственно cookie. Все данные обычно передаются в одной строке, но, чтобы листинг поместился на странице, мы разбили эту строку на три части. **В строке 7,** так как заголовок теперь уже закончен, мы передаем два символа конца строки. Этим мы сообщаем броузеру, что заголовок завершен и надо принимать содержимое.

**Строка 8** просто выводит некоторый текст, чтобы пользователь знал, что происходит.

**Строки 9—10** выводят ссылку, которая указывает на нашу следующую программу. Эта программа читает cookie и отображает его в броузере, так что пользователь может проверить, какие данные установлены. Текст, выводимый первой программой, показан на рис. 5.1.

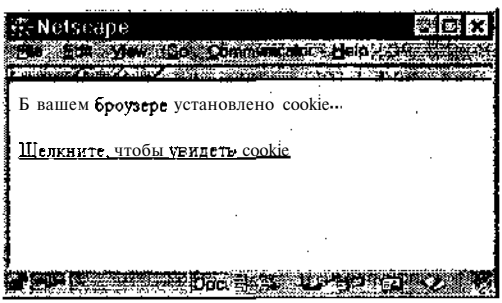



*Рис. 5. J. Вывод программы ручной установки cookie*

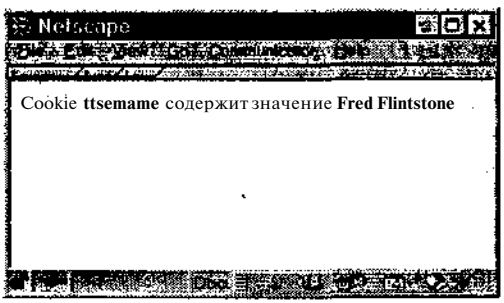



*Рис. 5.2. Вывод программы получения cookie вручную*

Перед запуском этой программы настройте Web-броузер так, чтобы он предупреждал вас при установке cookie. Предупреждения — очень полезное средство держать под контролем передачу cookie с сервера на броузер и обратно. Кроме того, предупреждения помогают при отладке скриптов.

Итак, мы вручную установили cookie. Теперь напишем небольшой скрипт, с помощью которого можно будет просматривать содержимое cookie и проверять, работают ли они так, как ожидается. Это небольшое приложение, код которого приведен в листинге 5.2, читает cookie и выводит его данные в броузер (рис. 5.2).

### Листинг 5.2. Получение cookie вручную

```
01:  #!/usr/bin/perl -wT
02:  # Пример 5-2
03:  use strict;
04:  my ($key, $value) = split(/=/, $ENV{HTTP_COOKIE});
05:  print qq(Content-type: text/html\n\n);
06:  print qq(Cookie <B>$key</B> содержит значение
       <B>$value</B>);
```

**Строки 1—3** — те же, что и в предыдущем примере.

В строке 4 вызывается функция `split ()`, которая разделяет cookie по знаку равенства (=). Cookie возвращается с сервера в броузер в переменной окружения `HTTP_COOKIE`. Поэтому, чтобы получить cookie, в функцию *split ()* передается параметр `$ENV['HTTP_COOKIE']`.

**Строка 5** выводит стандартный заголовок **HTTP.**

**Строка 6** выводит имя и значение cookie, благодаря чему мы можем видеть, что оно работает.

Эти два примера довольно просты, но они представляют собой хорошую отправную точку. Гораздо сложнее будет установить cookie так, чтобы срок его действия истекал через месяц, или присвоить ему значения из хэша или массива. Представьте, сколько кода придется тогда писать! К счастью, Линкольн Штайн снова приходит к нам на помощь со своим CGI.pm. Остальная часть этой главы посвящена тому, как использовать функции модуля CGI, поскольку они очень облегчают работу.

# Как испечь cookie с помощью CGI .pm

Применение `CGI.pm` для обработки cookie намного упрощает задачу программиста. Как и при обработке форм HTML, CGI .pm берет на себя основную часть работы.

Установка cookie в `CGI.pm` сводится к передаче тех же параметров в функцию *cookie (),* но этот модуль предоставляет больше возможностей. Предположим, что нам требуется, чтобы срок действия cookie истекал через три месяца. Если попытаться реализовать это вручную, придется как-то определять дату и день недели через три месяца от настоящего момента. А в CGI.pm достаточно просто передать в параметре *expires* значение `+3M`. Модуль CGI проделывает все преобразования формата даты. Такие функции делают работу с cookie намного проще!

Мы начнем с повторения той же задачи, которую выполнили в предыдущем примере вручную. Затем перейдем к более сложным примерам. И на этот раз, так как в программе всего 12 строк, мы приведем ее текст полностью и затем объясним, что она делает (см. листинг 5.3).

Листинг 5.3. Установка cookie с помощью **CGI.pm**

```
01:  #!/usr/bin/perl -wT
02:  # Пример 5-3
03:  use strict;
04:  use CGI qw(:standard);
05:  my $cookiel = cookie( -name => 'username2',
06:                        -value => 'Barney Rubble',
07:                        -expires => '+1y',
08:                      );
09:  print header( -cookie => $cookiel );
10:  print qq(
       В вашем броузере установлено еще одно cookie...<P>);
11:  print qq(<A HREF="example5-4.cgi">);
12:  print qq(Щелкните, чтобы увидеть cookie</A>) ;
```

**Строки 1–3** — как в предыдущих примерах. **В строке 4** загружается модуль CGI.

В **строках** 5–8 с помощью одноименной функции устанавливается cookie. Обратите внимание, что элементы cookie имеют прежние имена. Отличие состоит в том, что теперь данные передаются в хэше, а не в парах имя-значение, как при ручном методе. В этом примере устанавливается срок действия cookie один год (+1y).

**Строка 9** выводит стандартный заголовок HTTP с помощью функции CGI.pm *header <).* Установка cookie в броузер пользователя производится при передаче его в параметре функции *header().* Допускается передача и нескольких **cookie** (это мы рассмотрим в последующих примерах).

**Строки 10-12** выводят некоторую информацию, чтобы пользователь знал, что происходит. Здесь также выводится ссылка на другую программу, которая позволяет проверить, правильно ли установлено cookie.

Текст вывода этой программы, показанный на рис. 5.3, столь же интересен, как и в первом примере. Однако теперь у нас есть два установленных cookie, потому что каждое cookie имеет собственное имя.

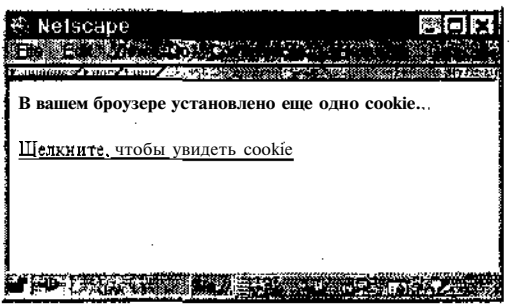



*Рис. 5.3. Вывод программы установки cookie с помощью CGI.pm*

Эта программа очень похожа на первый пример, который мы рассмотрели в начале главы. По сути, в ней даже больше строк, чем в программе ручной установки. Но только из того, что программа длиннее, не следует, что она сложнее. Когда мы перейдем к программам, в которых устанавливается несколько cookie или данные передаются в массиве или в хэше, станет виден настоящий выигрыш от применения модуля CGI. А сейчас попробуем прочитать cookie, которое мы только что установили (см. листинг 5.4). После этого мы сможем перейти к более интересным примерам.

**Листинг 5.4. Чтение cookie с помощью CGI.pm**

```
01:  #!/usr/bin/perl -wT
02:  # Пример 5-4
03:  use strict;
04:  use CGI qw(:standard);
05:  my $old_value = cookie ('username');
06:  my $new_value = cookie('username2');
07:  print header;
08:  print qq(Новое cookie <B>username2</B>);
09:  print qq (содержит значение <B>$new_value</B><P>);
10:  print qq(CTapое cookie <B>username</B>);
11:  print qq(содержит значение <B>$old_value</B>);
```

**Строки 1-3** — опять такие же, как в предыдущих примерах.

В **строке** 4 загружается модуль CGI и стандартные функции, как в листинге 5.3.

**В строках 5—6** создаются переменные `$old_value` и `$new_value`, в которых будет храниться содержимое cookie. Значения этих переменных устанавливаются сразу же с помощью функции *cookie ()* из модуля CGI. Эта функция — еще одна из изюминок модуля `CGI.pm`. В листинге 5.3 мы использовали функцию *cookie()* для создания cookie. Теперь мы с помощью этой же функции получаем cookie из броузера. Для этого надо просто вызвать функцию с указанием имени cookie. Возвращаемое значение — содержимое cookie, если оно существует.

**Строка 7** выводит стандартный заголовок HTTP. Так как здесь в функцию *header ()* не передаются параметры, то cookie не устанавливается.

**В строках 8—11** выводится информация, которая обычно передается из cookie, чтобы пользователь видел, что программа работает.

Результат выполнения этого кода — рис. 5.4. Здесь можно видеть, что было установлено два разных cookie. Одно из них называлось username2, а второе — username. Одно из этих cookie было установлено вручную, а другое — при помощи функций модуля CGI. Как можно видеть, не имеет значения, как именно cookie было установлено, поскольку после установки cookie можно читать любым методом.

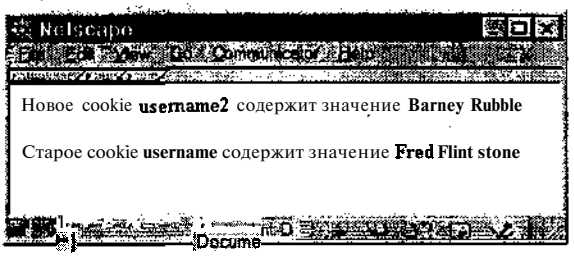



*Рис. 5.4. Чтение cookie с помощью CGI.pm*

# Управление предпочтениями пользователя через cookie

Теперь мы перейдем к более практическому применению cookie. В этом примере cookie будет управлять заголовком страницы, ее цветом, цветом текста и именем пользователя. Пример также продемонстрирует установку нескольких cookie, и вы поймете, что это почти так же просто, как установка одного cookie. Так как эта программа длиннее, мы будем описывать ее по частям.

```
01:  #!/usr/bin/perl -wT
02:  # Пример 5-5
03:  use strict;
04:  use CGI qw(:standard);
05:  use CGI::Carp qw(fatalsToBrowser);
```

**Строки 1–5** — стандартные для всех наших программ. **Строка 1** сообщает системе, где искать Perl, и включает проверку на загрязнение и предупреждения. **Строка 2** — комментарий, **строка 3** включает строгий синтаксис, **строка 4** загружает модуль CGI, а **строка 5** — модуль `CGI::Carp`.

```
06:  my ($Page_Cookie1, $Page_Cookie2);
07: ray $Flag = 0;
```

**Строка 6** создает переменные, в которых будут храниться cookie.

Строка 7 создает переменную с именем $Flag и присваивает ей начальное значение 0. Значение $Flag станет 1, если мы обнаружим, что какие-то cookie должны быть изменены.

```
08: my %Values = cookie('page_values');
09: my $Title = cookie('page_title');
```

**В строке 8** создается переменная-хэш `%Values` и ей с помощью *cookie ()* из модуля CGI присваиваются значения из cookie `page_values`.

В **строке** 9 создается переменная $Title и ей присваиваются значения из cookie `page_title`. Заметьте, что функция *cookie ()* в **строке 8** возвращает хэш, но в **строке 9** — скаляр. Эта функция может возвращать хэш, массив или скаляр в зависимости от контекста, в котором она используется.

```
10:  foreach (param()){
11:    next unless param($_);
12:      $Values{$_} = param($_);
13:      SFlag = 1;
14:  }
```

**Строки 10—14** — цикл по всем параметрам, переданным из формы **HTML.** Вызов *param()* внутри *foreach()* в **строке 10** совершенно правомерен, так как функция *param()*, если в нее ничего не передано, возвращает список всех параметров. Следовательно, *foreach ()* — цикл по списку, возвращенному функцией *param ()*.

В **строке** 11 команда next производит выход из данной итерации цикла и переход к следующей, если текущий параметр не содержит никаких данных.

Механизм работы этого кода следующий: next завершает текущую итерацию цикла, в котором эта функция находится. *unless ()* — это то же самое, что if *not (false)*. Следовательно, мы проверяем, не возвратила ли функция param{$_) значение лжи (если данных нет). Оператор *next* выполняется, только если *param ()* не возвратила данных.

По-русски это можно выразить так: "Закончить итерацию цикла, если функция *param ()* не возвратила никаких данных".

В **строке** 12 ключу $_ хэша `%Values` (т.е., текущему значению *foreach ())* ставится в соответствие значение, возвращаемое функцией *param ()* с параметром $_. Так как мы устанавливаем только одно значение за раз, хэш Values имеет префикс $, а не `%`.

**Строка 13** устанавливает значение $Flag равным 1. Благодаря этому программа знает, что были внесены изменения. Мы не будем заниматься установкой нового cookie, если ничего не изменилось.

**Строка 14** завершает цикл foreach.

```
15:  $Values{Textcolor} = "#000000" unless($Values{Textcolor});
16:  $Values(Bgcolor} = "#FFFFFF" unless($Values{Bgcolor});
17:  $Values{Name} = "NoName" unless($Values{Name});
```

**В строках 15–17** устанавливаются значения по умолчанию, если какое-то из значений еще не задано. Мы должны убедиться, что Values содержит хотя бы какое-нибудь значение, чтобы его можно было использовать для атрибутов страницы, и, если это не так, установить значение по умолчанию. Функция *unless ()* здесь проверяет, пуст ли данный элемент хэша или нет. Если элемент хэша пуст, ему присваивается значение по умолчанию.

```
18:  $Title = "Пример Cookie" if (STitle eq "");
```

**В строке 18** устанавливается заголовок страницы по умолчанию, если $Title не содержит данных.

```
19:  if (param('Title')){
20:    $Title = param('Title');
21:    $Flag = 1;
22:  }
```

В **строках 19—22** выясняется, содержит ли поле Title заголовка HTML данные. Если это так, выполняется блок кода.

**В строке 20** переменной $Title присваивается значение, переданное из формы HTML.

В **строке 21** переменной $Flag присваивается значение 1, что означает необходимость установки cookie заново.

**Строка 22** закрывает блок if.

```
23:  if ($Flag)(
```

**Строка** 23 начинает блок if, который выполняется, если переменная $Flag содержит ненулевое значение.

```
24:      $Page_Cookiel = cookie (-name => 'page_values',
25:                              -expires => '+2m',
26:                             -value => \%Values );
```

**В строках 24—26** мы создаем первое cookie, которое будет установлено. Строка 24 вызывает функцию *cookie ()* из модуля CGI. -name, -expires и -value — это параметры функции *cookie ()*. Другие параметры указываются в **строках** 25 и 26. В строке 26 фактически передается ссылка на хэш.

Функция *cookie (}* имеет шесть различных параметров. Единственный обязательный параметр — name.

- name — имя устанавливаемого cookie. Это единственный обязательный параметр.
- expires — временной интервал, на протяжении которого cookie имеет силу (срок действия). Для этого cookie мы устанавливаем срок действия две минуты (+2т), чтобы можно было проверить, что срок действия действительно истекает. В этом параметре можно задавать секунды (s), минуты (т), часы (h), дни (d), настоящий момент (now), месяцы (м) и годы (у); можно также передавать строку даты и времени, как в примере ручной установки cookie. Допускается указывать и отрицательные значения, если требуется, чтобы срок действия истекал немедленно.
- name — устанавливаемое значение cookie. Модуль CGI позволяет задавать здесь ссылки на хэши или массивы. Если попытаться сделать это вручную, придется писать гораздо больше кода, так как хэши и массивы надо будет раскрывать в строки перед преобразованием в строку cookie. А после чтения cookie хэш или массив придется формировать снова. Модуль CGI проделывает все это автоматически.
- domain — диапазон доменов, в которые броузер будет отправлять это cookie. По умолчанию cookie отправляется только на сервер, который создал его.

- path позволяет задать частичный путь, для которого cookie будет иметь силу. Если к одному серверу относятся несколько cookie, можно установить путь, например, /shoppingcart. Тогда cookie будет отправлено только в том случае, если страница исходит из каталога /shoppingcart или одного из его подкаталогов.
- secure может иметь значения истины (1) или лжи (0). В первом случае cookie отправляется, только если используется безопасное подключение. Значение по умолчанию — ложь, так что этот параметр обычно не используется.

```
27:    $Page_Cookie2 = cookie(-name => 'page_title',
28:                           -expires => '+2m',
29:                           -value => $Title );
```

**В строках 27-29** таким же способом устанавливается другое cookie. Это cookie содержит заголовок страницы.

Вместо того, чтобы создавать два cookie, легко можно было бы установить одно cookie с несколькими значениями. Однако тогда мы не увидим примера установки нескольких cookie! Допускается установка нескольких cookie одновременно. Просто надо помнить об ограничениях на количество и размер отправляемых cookie.

```
30:    print header(-cookie => [$Page_Cookiel, $Page_Cookie2]);
31:  }
```

**Строка 30** выводит заголовок HTTP и устанавливает оба cookie. Cookie передаются в заголовке HTTP, поэтому все cookie, которые требуется установить, достаточно указать в параметрах функции *header ()*. Если передается одно cookie, квадратные скобки [ ] можно опустить, но в случае нескольких cookie они необходимы.

**Строка 31** просто закрывает часть if структуры if. . .else.

```
32:  else {
33:    print header();
34: }
```

**Строка** 32 начинает часть else структуры if. . .else. Эта часть выполняется, если cookie устанавливать не нужно. В этом случае в **строке** 33 просто вызывается функция *header ()* без параметров.

**Строка 34** закрывает структуру if. . .else.

```
35:  print «HTML;
36:  <HTML><HEAD><TITLE>$Title</TITLE></HEAD>
37:  <BODY BGCOLOR="$Values{Bgcolor}"
38:  TEXT="$Values{Textcolor}">
39:  <FORM>
40:  <H2>Добро пожаловать, $Values{Name}!!!</H2>
41:  <TABLE>
42:  <TR>
43:  <TD>Имя:</TD>
44:  <TDxINPUT TYPE="text" NAME="Name"></TD>
45:  </TR>
46:  <TR>
47:  <TD>Заголовок:</TD>
48:  <TD><INPUT TYPE="text" NAME="Title"></TD>
49:  </TR>
50:  <TR>
51:  <TD>Цвет фона:</TD>
52:  <TD><INPUT TYPE="text" NAME="Bgcolor"x/TD>
53:  </TR>
54:  <TR>
55:  <TD>Цвет текста:</TD>
```

```
56:  <TD><INPUT TYPE="text" NAME="Textcolor"></TD>
57:  </TR>
58:  </TABLE>
59:  <P>
60:  <INPUT TYPE="submit" VALUE="Отправить запрос">
61:  </FORM>
62:  </BODY>
63:  </HTML>
64:  HTML
```

**Строки 35—64** — включенный документ, который выводит готовую страницу. В **строках 36, 37, 38 и 40** упоминаются значения, с которыми мы работали в начале этой программы. Они исходят либо из cookie, либо из формы HTML, либо принимаются по умолчанию.

Когда эта программа запускается в первый раз, экран выглядит примерно так, как на рис. 5.5. Это стандартный вид, так как пользователь еще не выбрал никаких предпочтений. Введите в поля какие-нибудь данные и щелкните на кнопке Отправить запрос. Будут установлены cookie, и вид страницы изменится. Когда пользователь введет данные и нажмет кнопку отправки, страница будет перезагружена с новыми параметрами, как на рис. 5.6.

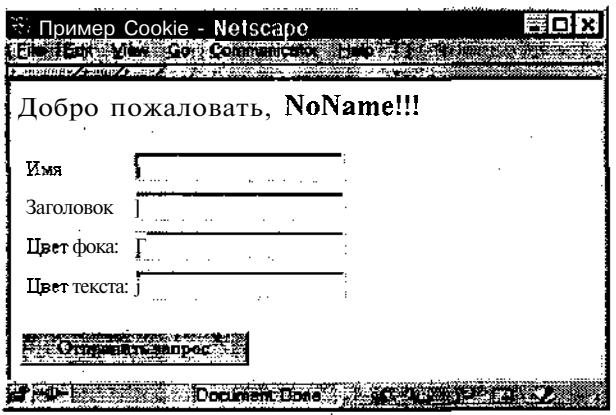



*Рис. 5.5. Установка предпочтений пользователя*

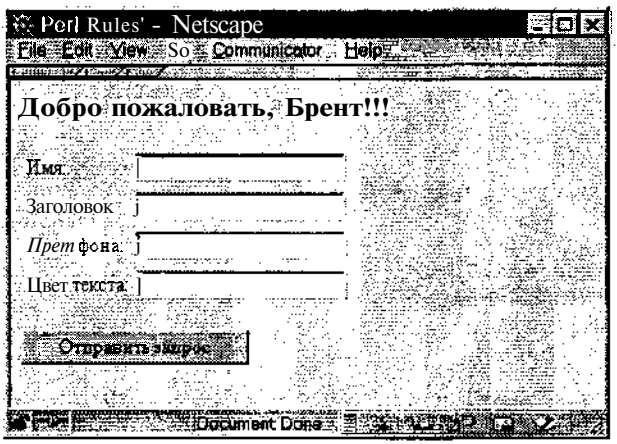



***Рис.** 5.6. Страница с установленными предпочтениями*

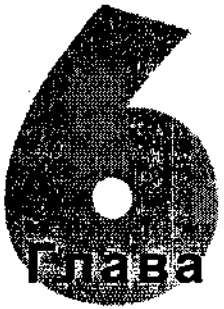
Как мы помним, cookie в этом примере действуют только две минуты, так что если подождать подольше, цвета опять станут стандартными, как на предыдущем рисунке. Двухминутный срок действия не очень удобен для предпочтений пользователя, но он демонстрирует, как задать время действия cookie, и позволяет легко проверить, что после этого времени cookie действительно перестает работать.

Наличие cookie, которое прекращает свое действие через короткое время, скажем, 15 минут, в некоторых случаях может быть очень практичным. Например, вы работаете в сетевом банковском приложении и хотите на какое-то время оставить свое рабочее место. Естественно, доступ к системе не может оставаться открытым неопределенный срок. Поэтому во многих системах устанавливается срок действия cookie около 15 минут. Каждый раз, когда пользователь выполняет какое-то действие в приложении, проверяется срок действия cookie. Если этот срок еше не истек, он устанавливается заново, и действие пользователя обслуживается приложением. Если действие cookie закончилось, программа не будет выполнять требуемую задачу, а потребует зарегистрироваться снова. Так можно предотвратить ту ситуацию, когда кто-то посторонний подойдет к компьютеру в отсутствие зарегистрированного пользователя и выполнит нежелательную операцию, например, перевод денег.

Хотя cookie — не более чем маленькие фрагменты данных, которые можно только устанавливать и считывать, это чрезвычайно мощная составляющая программирования Web-приложений. Они позволяют управлять длительностью сеансов, сохранять предпочтения пользователя, фиксировать небольшие, но важные элементы данных или даже запоминать, сколько раз пользователь посетил страницу.

# Упражнения

- Создайте страницу, которая увеличивает значение cookie на единицу и сообщает пользователю, сколько раз он посетил страницу.
- Создайте страницу, которая хранит в cookie идентификатор пользователя и ищет по нему информацию в таблице базы данных.




# Счетчики обращений

## Введение

С самого начала популярности Web разработчикам Web-сайтов требовалось знать, сколько пользователей посещает их сайты, и иметь возможность передавать эту информацию другим. Сегодня мы можем видеть на Web-сайтах счетчики посещений всех сортов и разновидностей. Поскольку этот тип приложений так популярен, мы посчитали нужным показать, как можно сделать счетчик посещений самому.

Счетчики посещений стали настолько популярными, что в Web появились даже бесплатные услуги такого рода. Многие из них дают доступ к своему сценарию, который отслеживает посещения вашего сайта. Но зачем прибегать к чьей-то помощи, когда вы можете выполнить эту задачу для себя или для ваших клиентов самостоятельно? Если сейчас вы пользуетесь какой-либо из подобных служб, к концу главы вы поймете, что это вам уже не понадобится, так как вы будете в состоянии создать собственный скрипт CGI для этой цели.

Существуют различные способы применения и отображения счетчиков. Один из них — вставка на стороне сервера (Server Side Includes — SSI), которая отображает количество посещений в виде текста или изображения. Другой способ основан на использовании в скрипте дескриптора HTML `<IMG>` И показе числа посещений в форме изображения. В этой главе мы расскажем об использовании обоих методов. Вы также научитесь создавать графический счетчик, которому не нужны никакие изображения. В заключение мы несколько "подправим" счетчик так, чтобы он показывал, сколько людей еще *не было* на вашем Web-сайте!

Число посещений в примерах этой главы сохраняется в простом текстовом файле. Этот способ хранения прост, быстр и мобилен. Для этой цели можно очень просто применить простую таблицу базы данных, но мы предпочли использовать текстовый файл, чтобы продемонстрировать, как безопасно им управлять, а также по причинам,

указанным выше. Счетчики посещений имеют одну особенность: они не различают отдельных посетителей, а учитывают общее число "заходов" на страницу. Большая часть счетчиков, доступных и используемых сегодня, работает именно так и фиксирует каждый экземпляр страницы, загружаемый с Web-сервера. Подчеркнем — *каждый загружаемый экземпляр*, так как счетчик не увеличивается, если пользователь просматривает страницу из кэша. Подробнее о кэшировании см. в главе 2.

Примеры этой главы являются приложениями совместного доступа. Это означает, что один скрипт счетчика может обслуживать несколько страниц или несколько Web-сайтов. Эта функциональная возможность появляется благодаря тому, что каждый скрипт имеет параметр, в котором указывается имя и расположение файла посещений. Поэтому скрипт может получать правильные данные. Другое условие состоит в том, что файл данных хранится в том же каталоге, что и сам скрипт. Поэтому вам придется изменить путь в соответствии с конфигурацией вашей системы.

В этой главе используются следующие модули и прагмы.

- `lib`
- `strict`
- `CGI`
- `Fcntl`
- `GD`
- `Image::Size`
- `LWP::UserAgent`
- `Untaint`

# Пример: текстовый счетчик SSI

SSI очень часто используется для отображения количества обращений. Этот способ имеет значительную гибкость, поскольку позволяет выводить изображения, текст или их сочетание. Счетчик, который отображает только текст, имеет некоторые преимущества, в частности, не нуждается в файлах изображений. Этот способ прекрасно подходит для отображения чисел, так как не нужно запрашивать и выводить изображения. Графический счетчик вряд ли замедлит загрузку страницы, но, если на вашем сайте много изображений, зачем еще увеличивать их количество, если этого можно избежать? С другой стороны, если ваш сайт имеет высокий трафик, загрузка множества изображений на многих страницах вызывает дополнительную нагрузку на сеть и может снизить общую скорость загрузки страниц (конечно, на это влияет и размер изображений и их размещение).

Следующий пример `count.pl` отображает количество посещений в текстовом виде с помощью SSI. Убедитесь, что вы можете использовать SSI[1] в ваших каталогах. Чтобы можно было вызвать скрипт, включите в текст Web-страницы следующую строку SSI.

```
<!--#include virtual="../cgi-bin/count.pl text_counter.dat"-->
```

Это код `count.pl`.

```
01:  #!/usr/bin/perl -wT
02:  # count.pl
03:  use strict;
04:  use Fcntl qw(:flock);
05:  use Untaint;
```

[1] *Для этого Web-сервер должен поддерживать SSI. Большинство серверов поддерживают SSI, но некоторые, например, Microsoft Personal Web Server — нет.*

**В строках 1-5** загружаются все необходимые модули и прагмы. Константы, загружаемые из `fcntl.h`, позволяют управлять блокировкой (функция *flock ()).*

```
06:  my $counter = untaint(qr(.*\.dat$), $ARGV[0]);
```

**Строка 6** инициализирует переменную `$counter`. Этой переменной присваивается очищенное значение первого параметра, переданного в скрипт — имя и расположение файла данных. Значение очищается, только если оно заканчивается расширением файла `.dat` — такой образец передан в метод *untaint ()*. Таким образом, этот скрипт можно использовать для разных файлов данных, так как он работает независимо от того, какой файл указан.

```
07:  my $silent = 0;..
```

**Строка 7** определяет переменную `$silent`. Этот счетчик позволяет как записывать количество посещений и не показывать его клиенту, так и сообщать клиенту это число. Если `$silent` имеет значение 0, количество посещений будет отображаться, если 1 — нет.

```
08:  my $link - 1;
```

**Строка 8** определяет переменную `$link`, которая также используется для настройки. Как мы увидим ниже, этот скрипт может отображать количество посещений как гиперссылку. Это происходит, если `$link` равно 1; при значении 0 количество отображается просто как текст.

```
09:  my $link_URL = "http://www.perl.org/";
```

**Строка** 9 определяет `$link_URL`. Это URL, на который указывает счетчик, если Slink равно I.

```
10:  my $SEMAPHORE = $counter . ".lck";
```

**Строка 10** инициализирует переменную, указывающую на файл семафора. С помощью этого файла мы гарантируем, что условие состязания не наступит. Возникновение состязания в скрипте Perl при использовании текстового файла — слишком частая оплошность, допускаемая многими программистами на Perl, так что мы должны уделить несколько минут этой проблеме. Очень часто разработчики программируют примерно такую последовательность событий.

### Листинг 6.1. События, которые приводят к возникновению состязания

```
01:  Открытие файла для чтения.
02:  Блокировка файла.
03:  Чтение файла.
04:  Закрытие файла.
05:  Повторное открытие файла для записи.
06:  Блокировка файла.
07:  Запись в файл.
08:  Закрытие файла.
```

Вы уже поняли, в чем проблема? Здесь возникает условие состязания, так как одна программа может открыть файл для чтения, когда другая открыла его для записи, и так как файл может быть разблокирован прежде, чем данные будут полностью записаны на диск. Это может вызывать искажение данных, потому что две программы могут пытаться работать с файлом двумя различными способами в одно и то же время. Это обычная ошибка, и, чтобы избежать ее, мы будем блокировать некоторый файл, полностью отдельный от того файла, с которым мы будем работать. Когда этот файл семафора заблокирован, мы можем безопасно работать с файлом данных, поскольку, если одновременно выполняется другой экземпляр этой программы, каждый из них, чтобы начать работу с файлом данных, должен будет ждать разблокировки файла семафора. Соответствующий порядок событий показан в листинге 6.2.

### Листинг 6.2. События, которые препятствуют возникновению состязания

```
01: Открытие файла семафора для записи.
02: Получение монопольной блокировки файла семафора,
03: Открытие файла данных.
04: Работа с файлом данных.
05: Закрытие файла данных.
06: Закрытие файла семафора.
```

Как можно видеть из этого листинга,, операции с файлом данных не выполняются, пока файл семафора не будет монопольно блокирован. Следовательно, не может возникнуть ситуация, когда два экземпляра скрипта (или два скрипта) будут работать с файлом данных одновременно. В отношении блокировки следует запомнить один важный факт: функция Perl *flock ()* на самом деле не препятствует всем другим процессам системы изменять файл. Она, как и функция UNIX flock(2), лишь устанавливает рекомендательную (advisory) блокировку, не блокируя файл физически от всех других процессов. Иначе говоря, если другой процесс попытается заблокировать этот же файл с помощью *flock ()*, первый процесс подаст голос: "Стоп! Я работаю с этим файлом. Подождите, пока я закончу". Процесс, пытающийся установить блокировку, будет ждать. Но если другая программа попытается обратиться к файлу данных, не попробовав предварительно заблокировать соответствующий файл семафора с помощью *flock ()*, она сможет работать с этим файлом совершенно свободно. Это всегда следует помнить при разработке приложений CGI и планировать непротиворечивый процесс блокировки.

Чтобы продемонстрировать, что случится, когда два экземпляра скрипта попробуют обратиться к файлу данных, открытому по методике листинга 6.2, запустите скрипт, показанный в листинге 6.3, в двух (или более) разных окнах консоли одновременно.

### Листинг 6.3. Демонстрация flock() в скрипте командной строки

```
01: #!/usr/bin/perl -wT
02: use Fcntl qw(:flock);
03: my $file = 'test_lock.txt';
04: my $SEMAPHORE = $file . '.lck';
05: open(S, ">>$SEMAPHORE") or die "$SEMAPHORE: $!";
06: flock(S, LOCK_EX) or
      die "Не выполняется flock() для $SEMAPHORE: $!"
07: open (FH, ""$file") or die "Нельзя открыть $file: $!";
08: print "Собираюсь писать\n";
09: print FH "Пишу ($S)\n";
10: print "Написал\n";
11: close FH;
12: print "Останавливаюсь...\n";
13: sleep 10;
14: print "Продолжаю...\n";
15: close S;
```

Вы увидите, что произойдет, когда один из скриптов остановится, ожидая получения монопольной блокировки файла. Также будет видно, что одновременно работать с файлом данных может только один экземпляр.

```
11: open(S, ">>$SEMAPHORE") ||
    die "Нельзя открыть файл семафора (SSEMAPHORE): $!";
12:  flock (S, LOCK_EX);
```

В строках 11 и 12 мы открываем файл семафора для записи и пробуем установить на него монопольную блокировку. Если *flock ()* обнаруживает, что файл уже блокирован, она будет ждать, пока не станет возможным получить монопольную блокировку. Файл семафора открывается в режиме записи, поскольку иногда на файл, открытый только для чтения, монопольная блокировка не предоставляется.

```
13:  if (-e $counter) {
14:    open (FILE, $counter) | |
         die "Нельзя прочитать счетчик. $!";
15:    my $visits = <FILE>;
16:    close (FILE);
17:  }
```

**Строки 13-17** открывают и читают файл счетчика. Значение — количество посещений — сохраняется в переменной $visits, и **строка** 15 закрывает файл. Это происходит, только если файл счетчика существует, так как **строка 14** завершает скрипт, если операция чтения не удается.

```
18:  open (FILE, ">$counter") | |
       die "Нельзя записать в счетчик. $!";
19:  print FILE ++$visits;
20:  close (FILE);
```

**В строке 18** создается исходный файл данных счетчика. В **строках** 18, 19 **и** 20 этот файл повторно открывается для записи и в него записывается новое количество посещений — значение $visits после увеличения на единицу.

```
21:  close S;
```

**Строка 21** закрывает файл семафора. Этот файл не надо разблокировать, так как он автоматически разблокируется при закрытии. Кроме того, чтобы не создавать условия состязания, после разблокировки файла он должен быть закрыт. В противном случае другой процесс может попытаться открыть файл, прежде чем вы закончите изменение и закроете его.

```
22:  if (!$silent) {
23:    if (Slink) {
24:      print qq(Вы посетитель номер
           <A HREF="$link_URL">$visits</A>!);
25:    } else {
26:      print Svisits;
27:    }
28:  }
```

**В строках 22—28** количество посещений отображается в клиенте. **Строка** 22 проверяет, надо ли выдавать клиенту значение счетчика (переменная $silent). Если она равна 0, скрипт может закончить свою работу, так как в файл счетчика уже записаны самые свежие данные. Если нет, **строка** 23 проверяет, как должно быть отображено значение счетчика — в виде гиперссылки или простого текста. В первом случае выводится гиперссылка, а во втором — просто значение $visits. Все, дело сделано! Экран этого текстового счетчика показан на рис. 6.1.

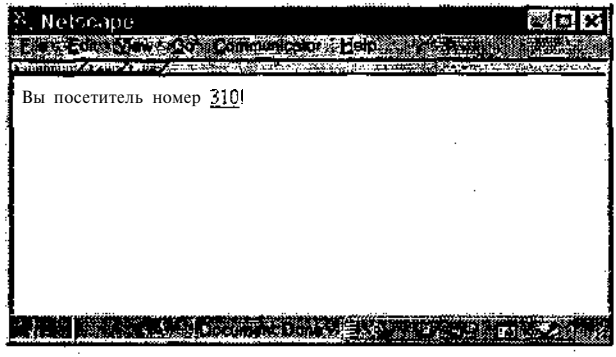



*Рис. 6.1. Экран текстового счетчика SSI*

Этот пример крайне прост, но в нем действительно реализовано все, что нужно для счетчика на Web-сайте. Нужно сказать, что современные счетчики не обязательно должны быть графическими. Большие и причудливые изображения цифр уже вышли из моды. Но вам или вашим клиентам графические счетчики могут еще нравиться. Счетчики с изображениями, конечно, имеют свою нишу в Web, и в следующем примере мы покажем, как создать графический счетчик — опять с использованием SSI.

# Пример: графический счетчик SSI

Следующий скрипт счетчика, `count2.pl`, тоже может быть скриптом совместного доступа. В него передаются два параметра, первый из которых — имя файла данных, как в предыдущем примере. Второй параметр — "стиль" изображений. Благодаря этому несколько пользователей могут работать с одним и тем же скриптом, а также выбирать свой стиль изображений, или же различные стили могут применяться на разных страницах одного сайта, чтобы сделать его интереснее. Под "стилем" мы понимаем изображения различного размера, цвета, формы, начертания и так далее.

Если этот скрипт должен работать в окружении совместного доступа, скорее всего он будет установлен в обычном каталоге `cgi-bin`, а файлы данных и изображения будут размещаться в каких-то других специальных каталогах. Структура каталогов, например, может выглядеть так.

**Листинг 6.4. Пример структуры каталогов для совместного доступа**

```
/usr/local/apache/cgi-bin
                        содержит count2.pl
/usr/local/apache/htdocs/counter
/usr/local/apache/htdocs/counter/data
                        содержит файлы данных .
/usr/local/apache/htdocs/counter/images/*.png
/usr/local/apache/htdocs/counter/images/plain/*.png
/usr/local/apache/htdocs/counter/images/fancy/*.png
/usr/local/apache/htdocs/counter/images/funny/*.png
и т.д.
```

Такую структуру каталогов удобно использовать в общедоступном окружении. Добавить новый стиль изображений здесь просто — нужно только создать новый каталог стиля с соответствующими изображениями. В каждом каталоге стиля должно быть 10 изображений — по одному для каждой цифры от 0 до 9, в файлах с именами <цифра>.рпд. Это простой и гибкий способ, позволяющий неограниченно расширять коллекцию стилей. Теперь перейдем к самому скрипту `count2.pl`.

```
01: #!/usr/bin/perl -wT
02: # count2.pl
03: use strict;
04: use Fcntl qw(:flock);
05: use Untaint;
06: use Image::Size qw(html_imgsize);
```

**Строки 1–6** — как в предыдущем примере, за исключением **строки 6.** Эта строка вызывает модуль `Image::Size`, а также импортирует его метод *`html_imgsize()`*. Модуль `Image::Size` используется для определения размера изображений. Метод *`html_imgsize()`* не только выдает размер изображения, но и возвращает строку, пригодную для использования в дескрипторе <IMG>. Подробнее об этом см. в **строке 21.**

```
07: my Scounter = untaint(qr(.*\.dat$), $ARGV[0]);
08: my Sstyle = untaint(qr(^(\.{0,2}/)*?[\w-/]+$), $ARGV[1]);
```

**В строках 7 и 8** очищаются параметры. Как в предыдущем примере, имя файла данных счетчика, переданное в первом параметре, сохраняется в переменной Scounter, если оно имеет расширение .dat. **Строка 8** очищает параметр — название стиля. Предполагается, что стиль — это имя каталога, и поэтому оно может включать алфавитно-цифровые символы и дефис. Здесь также проверяется, относительный ли это путь к каталогу или абсолютный. Благодаря этой проверке каталоги изображений могут располагаться где угодно, а не только в текущем рабочем каталоге.

```
09: my $SEMAPHORE = $counter . ".lck";
10: open(S, ">>$SEMAPHORE") ||
     die "Нельзя открыть файл семафора ($SEMAPHORE): $!";
11: flock (S, LOCK_EX);
12: open (FILE, Scounter) ||
     die "Нельзя прочитать счетчик. $!";
13: my $visits = <FILE>;
14: close (FILE);
15: open (FILE, ">$counter") ||
     die "Нельзя записать в счетчик. $!"
16: print FILE $visits++;
17: close (FILE);
18: close S;
```

**В строках 9—18** производится блокировка файла семафора и чтение/запись данных счетчика в файл с предотвращением состязания.

```
19: exit unless -d $style;
```

**Строка** 19 проверяет, существует ли указанный каталог стиля. Если он не существует, скрипт завершается по *exit()*. Это помогает избегать попыток открытия несуществующих изображений.

```
20: my @digits = split(//,$visits);
```

**Строка 20** создает массив @digits, каждый элемент которого — цифра, составляющая число посещений ($visits). Этот массив будет использоваться в следующих строках для отображения соответствующих цифр.

```
21: foreach (@digits) {
22:   my $size = html_imgsize("$style/$_.png");
23:   print qq(<img src="$style/$_.png">);
24: }
```

**Строки 21—24** — цикл по элементам массива. Вспомните, что каждый его элемент — отдельная цифра. **В строке 22** передается путь к изображению (каталог стиля, косая черта, переменная $_, которая содержит текущую цифру, и расширение .png) в метод *html_imgsize()*. Этот метод очень удобен; он возвращает строку, полностью готовую к использованию в дескрипторе HTML вставки изображения. Например, если переданное изображение имеет размер 10x10, будет возвращена строка WIDTH=16 HEIGHT=10. Эта строка сохраняется в $size. **В строке 23** готовый дескриптор **HTML** передается клиенту. Результатом, очевидно, будет отображение каждой цифры в графическом виде. Вот и еще один простой способ добавить счетчик на ваш Web-сайт! Экран готового счетчика показан на рис. 6.2. Чтобы вызвать этот скрипт, требуется следующая строка SSI.

```
<!—#exec cmd="../cgi-bin/count2.pl img_counter.dat myStyle"-->
```

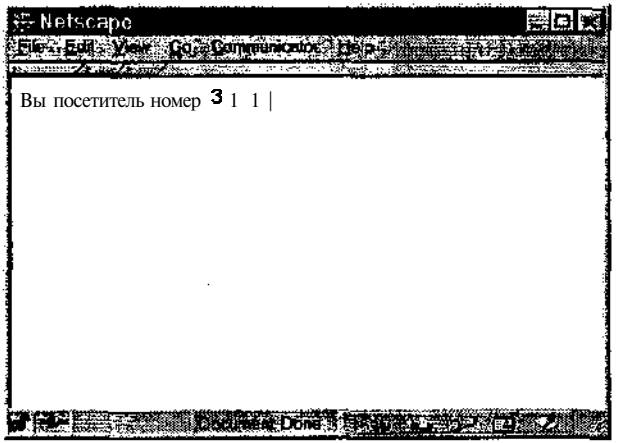



*Рис. 6.2. Экран, выводимый скриптом count2.pl*

# Пример: текстовый счетчик SSI "со сдвигом"[55]

Мы уже видели примеры создания с помощью SSI текстовых и графических счетчиков, которые показывают посетителям количество затребованных экземпляров Web-страницы. Это хорошо, но можно добавить в этот счетчик чуть больше юмора — сделать так, чтобы он показывал, сколько людей *еще не посетило* вашу Web-страницу! Этот текстовый счетчик (скрипт `populat.pl`) отнимает количество посещений от численности мирового населения, оценку которого дает Бюро переписи США[2], и отображает *приблизительное* количество людей в мире, еше не видевших вашей Web-страницы. Для вызова этого скрипта на Web-страницу следует поместить следующую команду SSI.

```
<!--#include virtual="../cgi-bin/populat.pl img_counter.dat"-->
```

Код `populat.pl` начинается так.

```
01:  #!/usr/bin/perl -wT
02:  # populat.pl
03:  use strict;
04:  use Untaint;
05:  use LWP::UserAgent;
```

В строках 1–5 можно заметить только один новый модуль— LWP: :UserAgent. Модули `LWP`, в частности, выполняют задачу получения данных с удаленного URL, в нашем случае, от Бюро переписи США, так что мы можем получить его оценку численности мирового населения.

```
06:  my $counter = untaint(qr(.*\.dat$), $ARGV[0]);
```

Строка 6 очищает входящий параметр точно так же, как в предыдущих примерах.

```
07:  my $ua = new LWP::UserAgent;
```

В строке 7 создается новый объект LWP: :`UserAgent`. Этот объект, в буквальном смысле, будет нашим агентом пользователя Web (user agent — UA) или клиентом.

---

[2] *Больше об оценках численности мирового населения можно узнать на Web-сайте Бюро переписи США по адресу* `http://www.census.gov/ipc/www/popwnote.html`.

```
08:   $ua->agent("GetCount/0.1");
```

В **строке** 8 устанавливается информация о продукте для нашего клиента. Эти данные будут представлены в разделе `User-Agent` запроса на удаленный Web-сервер.

В этом примере мы назвали наш "продукт" `GetCount` и присвоили ему версию `0.1`. Эта информация будет отображена в журнале Web-сервера на удаленной машине.

```
09:   my $req = new HTTP::Request GET =>
        'http://www.census.gov/cgi-bin/ipc/popclockw';
```

**В строке 9** создается объект `HTTP::Request`. Этому объекту сообщается метод запроса GET — и запрашиваемый URL. Модуль `HTTP::Request` не нужно импортировать командой *use()*, так как `LWP::UserAgent` делает это автоматически.

```
10:   $req->header(Accept => "text/html");
```

**В строке 10** устанавливается значение Accept для заголовка запроса HTTP. Тип MIME принимаемых документов будет `text/html`.

```
11:   my $res = $ua->request($req);
```

**В строке 11** через объект UA вызывается метод *request()* модуля `LWP::UserAgent`. Этот метод вызывает в фоновом режиме Web-страницу и возвращает объект со всеми данными ответа. В нашем случае новый объект — `$res`.

```
12:   if ($res->is_success) {
```

**Строка 12** с помощью метода *is_succe()* проверяется, был ли запрос успешным. Если да, возвращается значение истины, если нет — лжи.

```
13:     my $html = $res->content;
```

**Строка 13** с помощью метода *content()* извлекает содержимое, или HTML, из объекта ответа. Содержимое Web-страницы сохраняется в виде строки в `$html`.

```
14:     (my $num = $html) =~
          s!^.*<h1>(.*?)</h1>.*$!$!si;
```

**Строка 14** выполняет регулярное выражение, которое извлекает из строки HTML в $html число (мировое население). На Web-страшше Бюро переписи США, откуда исходит эта информация, численность мирового населения помещается внутри первой пары дескрипторов `H1`. Но нельзя забывать, что в будущем это размещение может измениться, и тогда эту строку придется отредактировать. Все остальное содержимое этой строки регулярное выражение отбрасывает, а извлеченное значение сохраняется в $num.

```
15:     open(LOG, $counter) || die
          "Нельзя прочитать счетчик. $!";
16:     my $log = <LOG>;
17:       close LOG;
```

**В строках 15, 16 и 17** открывается файл данных счетчика, как в предыдущих примерах, и значение из него считывается в переменную $log.

```
18:     $num =~ s!,!!д;
```

**Строка 18** удаляет все запятые из $num. Число, полученное и сохраненное в **$num,** содержит запятые как разделители групп разрядов (этого не будет, только если численность мирового населения окажется меньше 1000 (1,000) человек!), и, чтобы с этим числом можно было производить математические операции, запятые должны быть удалены (или превращены в символы подчеркивания _).

```
19:     $num = $num-$log;
```

**В строке 19** количество посещений из счетчика вычитается из численности мирового населения. Теперь мы знаем, сколько людей еще должно посетить нашу страницу.

```
20:     1 while $num =~ s!^(\d+)(\d{3})!$1,$2!;
```

**Строка 20** — цикл, в котором с помощью простого регулярного выражения в результат вновь вставляются запятые. Эта строка выполняется, пока подстановка возвращает значение истины (т.е. производится успешно). В результате значение `$num` (из которого ранее были удалены запятые) форматируется в прежнем виде.

```
21:     print $num;
```

**Строка 21** выводит число в клиент.

```
22:  } else {
23:    print "[ошибка при вычислении]";
24:  }
```

**Строки 22—24** завершают скрипт, отображая сообщение, если запрос закончился неудачно. На рис. 6.3 показан экран этого примера.

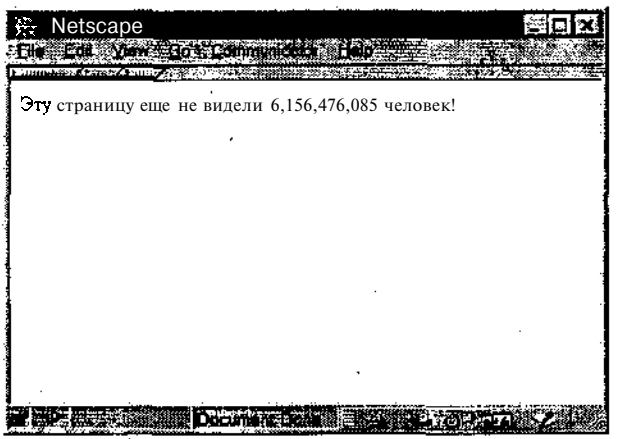



*Рис. 6.3. Экран счетчика мирового населения*

# Пример: графический счетчик без изображений

Как графический счетчик может обойтись без изображений? Это можно сделать, показывая изображение, которому не соответствует физический файл. В следующем примере мы сделаем именно это — покажем клиенту изображение, созданное "на ходу". Модуль GD позволяет создавать такие изображения на основе количества посещений. Самый обычный счетчик в следующем примере после небольших манипуляций с GD приобретает совершенно другой вид. Этот скрипт — не SSI, как в предыдущих примерах, он вызывается из дескриптора HTML <IMG>. Это очень полезно, если вы не хотите вызывать SSI на каждой странице со счетчиком или не имеете доступа к SSI[3].

Как можно запустить скрипт Perl из дескриптора HTML <IMG>? Когда на странице встречается дескриптор `<IMG>`, на сервер направляется запрос на изображение или любые другие данные, указанные в параметре `src`. Еще один запрос направляется на Web-сервер для получения данных. Например, если на Web-странице имеется 10 изображений, на Web-сервер направляется 11 запросов — 1 для самой страницы и 10 для изобра-

[3] *Если у вас нет доступа к SSI, вы, скорее всего, не сможете управлять Web-сервером. В этом случае рекомендуется попросить системного администратора разрешить вам использовать SSI.*

жений. Поэтому через дескриптор <IMG> МОЖНО отправить запрос на скрипт Perl, который возвратит изображение. Но если скрипт не возвратит правильный заголовок **MIME** для изображения и изображение правильного формата, пользователь увидит искаженное или незаконченное изображение. А теперь перейдем к скрипту (gd_count.pl).

```
01:  #!/usr/bin/perl -wT
02:  # gd_count.pl
03:  use strict;
04:  use Untaint;
05:  use GD;
06:  use CGI qw(param header);
```

**Строки 1~6** должны быть вам уже знакомы. **Строка 5** импортирует модуль GD, который будет использоваться для создания изображений. **Строка 6** импортирует два метода CGI.pm: *param()* и *header <).*

```
07:  print header("image/png");
```

**Строка 7** выводит заголовок "Content-type". Здесь мы указываем, что будет передано содержимое с типом MIME image/png, так как мы будем создавать и отображать картинку формата PNG[4].

```
08:  my $counter = untaint(qr(.*\.dat$), param('counter'));
09:  open(LOG, $counter) ||
       die "Нельзя прочитать счетчик. $!";
10:  my $log = <LOG>;
11:  close LOG;
```

**Строки 8—11** в объяснении уже не нуждаются.

```
12:  ray $im = new GD::Image((length($log)*9),20);
```

В **строке** 12 создается новый объект GD: : Image. В метод *new()* передаются требуемые ширина и высота изображения. В данном случае длина изображения (первый параметр) — это длина значения $log (количества посещений), умноженная на 9. В результате для каждой цифры количества посещений выделяется 9 пикселов, чего вполне достаточно для шрифта, который мы будем использовать. Второй параметр устанавливает высоту изображения в 20 пикселов, что также будет хорошо выглядеть с используемым шрифтом.

```
13:  my $black = $im->colorAllocate(0,0,0);
14:  my $white = $im->colorAllocate(255,255,255);
```

В **строках** 13 **и** 14 определяются два цвета, черный и белый. Первый определенный цвет GD использует как цвет фона для изображения, поэтому первым мы определяем черный цвет (Sblack). Затем определяется белый цвет для текста ($white). Метод *colorAllocate ()* имеет три параметра — красную, зеленую **и** синюю составляющие цвета. Частичный список кодов RGB приведен в приложении Д.

```
15:  $im->string(gdLargeFont,2,2,$log,$white);
```

**В строке 15** создается изображение из строки текста. В метод GD: :Image *string()* передаются пять параметров. Первый, gdLargeFont — тип шрифта, который будет применен для текста. Второй и третий параметры — координаты X и Y начала текста в изображении. Четвертый параметр, $log — собственно текст. И наконец, пятый параметр — цвет текста ( в данном случае белый).

---

[4] *Так как патенты на алгоритм сжатия LZW, используемый при создании изображений формата GIF, принадлежат компании Unisys, GD.pm в настоящее время создает только изображения формата PNG. Этот новый формат поддерживается не всеми броузерами. Существуют программы для преобразования файлов PNG в другие форматы. Самую свежую информацию по этому вопросу можно найти на http://www.png.org.*

```
16:  binmode(STDOUT);
17:  print $im->png;
```

**Строка 16** облегчает переносимость на системы Win32, устанавливая двоичный режим стандартного вывода, чтобы система Win32 правильно отобразила двоичные данные изображения. **Строка 17** конвертирует изображение в формат PNG и выводит его в клиент. Результат показан на рис. 6.4.

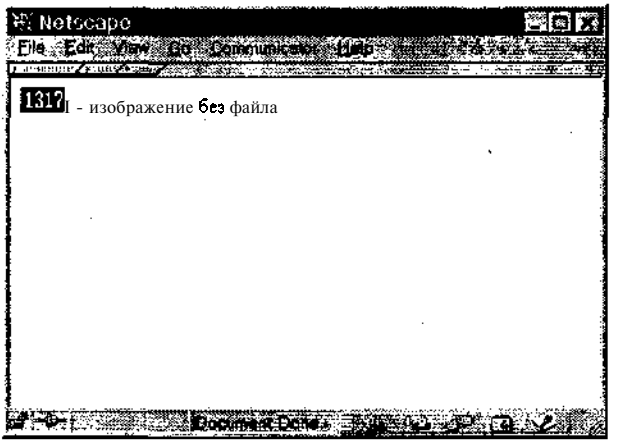



*Рис. 6.4. Экран счетчика, сгенерированного модулем GD*

# Заключение

В этой главе вы узнали, как создать гибкие, позволяющие совместный доступ счетчики для Web-страницы или сайта. Каждый метод имеет свои преимущества и возможные недостатки. Например, текстовый счетчик выглядит довольно скромно, счетчик населения получает информацию с удаленного сайта, графическому счетчику нужны файлы изображений (что может вызвать нежелательную дополнительную нагрузку), а счетчик без изображений требует установки модуля GD и связанных с ним пакетов. Лучше всего поэкспериментировать с каждым счетчиком и выбрать один или несколько типов, которые наилучшим образом удовлетворяют ваши потребности или потребности ваших клиентов.

# Упражнения

Как вы, возможно, заметили, скрипты в последних упражнениях только читали данные из файла. Первое задание для вас — добавить во все эти скрипты возможность обновлять данные.

Измените какой-либо из счетчиков так, чтобы блокировать попытки пользователя увеличить количество посещений, перезагружая страницу много раз. Простое решение этой задачи — использовать переменную окружения (см. главу 4) которая будет уникальной для клиента, и фиксировать пять (или около того) последних IP-адресов, с которых производилось обращение к сайту. Если текущий IP-адрес запроса совпадает с одним из них, увеличение счетчика запрещается.

Хотя в приложениях этого типа хранение данных в текстовом файле — самый быстрый и эффективный способ, может потребоваться применить для этой цели базу данных. Создайте в базе данных таблицу и напишите код для записи и считывания данных счетчика из нее с помощью **DBI.**

# Листинги

Листинг 6.5. Полный текст **count.pl**

```
01: #!/usr/bin/perl -wT
02: # count.pl
03: use strict;
04: use Fcntl qw(:flock);
05: use Untaint;
06: my Scounter = untaint(qr(.*\.dat$), SARGV[0]);
07: my $silent = 0;
08: my Slink = 1;
09: my $link_URL = "http://www.perl.org/";
10: my SSEMAPHORE = Scounter . ".lck";
11: open(S, ">>$SEMAPHORE") ||
    die "Нельзя открыть файл семафора (SSEMAPHORE): $!";
12: flock (S, LOCK_EX);
13: if (-e Scounter) {
14:   open (FILE, Scounter) || •
        die "Нельзя прочитать счетчик. $!";
15:   my Svisits = <FILE>;
16:   close (FILE);
17: }
18: open (FILE, ">$counter") ||
      die "Нельзя записать в счетчик. $!";
19: print FILE ++$visits;
20: close (FILE);
21: close S;
22: if (!$silent) {
23:   if ($link) {
24:     print qq(<A HREF="$link_URL">$visits</A>);
25:   } else {
26:     print Svisits;
27:   }
28: }
```

Листинг 6.6. Полный текст **count2.pl**

```
01: #!/usr/bin/perl -wT
02: # count2.pl
03: use strict;
04: use Fcntl qw(:flock);
05: use Untaint;
06: use Image::Size qw(html_imgsize);
07: my Scounter = untaint(qr(.*\.dat$), $ARGV[0]);
08: my Sstyle = untaint(qr(^(\.{0,2}/)*?[\w-/]+$) , $ARGV[1]);
09: my SSEMAPHORE = Scounter . ".lck";
10: open(S, ">>$SEMAPHORE") ||
      die "Нельзя открыть файл семафора (SSEMAPHORE): $!";
11: flock (S, LOCK_EX);
12: open (FILE, Scounter) ||
      die "Нельзя прочитать счетчик. $!";
13: my Svisits = <FILE>;
14: close (FILE);
15: open (FILE, ">$counter") ||
      die "Нельзя записать в счетчик. $!"
16: print FILE $visits++;
17: close (FILE);
```

```
18:  close S;
19:  exit unless -d Sstyle;
20:  my @digits = split(//,$visits);
21:  foreach (@digits) {
22:    my $size = html_imgsize("$style/$_.png");
23:    print qq(<img src="$style/$_.png">);
24:  }
```

**Листинг 6.7. Полный текст populat.pl**

```
01: #!/usr/bin/perl -wT
02: # populat.pl
03: use strict;
04: use Untaint;
05: use LWP::UserAgent;
06: my $counter = untaint(qr(.*\.dat$), $ARGV[0]);
07: my $ua = new LWP::UserAgent;
08: $ua->agent("GetCount/0.1");
09: my $req = new HTTP::Request GET =>
      'http://www.census.gov/cgi-bin/ipc/popclockw';
10: $req->header(Accept => "text/html");
11: my $res = $ua->request($req);
12: if ($res->is_success) {
13:    my $html = $res->content;
14:    (my $num = $html) =~
        s!^.*<h1>(.*?)</h1>.*$!$1!si;
15:   open(LOG, $counter) || die
        "Нельзя прочитать счетчик. $!";
16:    my $log = <LOG>;
17:      close LOG;
18:   $num =~ s! , ! !g;
19:    $num = $num-$log;
20:    1 while $num =~ s!^(\d+) (\d{3}) !$1,$2!;
21:    print $num;
22:  } else {
23:    print "[ошибка при вычислении]";
24:  }
```

**Листинг 6.8. Полный текст gd_count.pl**

```
01: #!/usr/bin/perl -wT
02: # gd_count.pl
03: use strict;
04: use Untaint;
05: use GD;
06: use CGI qw(param header);
07: print header("image/png");
08: my $counter = untaint(qr(.*\.dat$), param('counter'));
09: open(LOG, $counter) ||
      die "Нельзя прочитать счетчик. $!";
10:  my $log = <LOG>;
11:  close LOG;
12:  my $im = new GD::Image((length($log)*9),20);
13:  my $black = $im->colorAllocate(0,0,0);
14:  my $white = $im->colorAllocate(255,255,255) ;
15: $im->string(gdLargeFont,2,2,$log,$white);
16:  binmode(STDOUT);
17:  print $im->png;-
```


# Загрузка файлов на базе Web

## Введение

Загрузка файлов — это мощный, но удивительно редко рассматриваемый инструмент Web. Собирая информацию о загрузке файлов, я обнаружил, что по этому вопросу сейчас доступно очень немного данных. Загрузка файлов — слишком полезная функция, чтобы ее можно было обойти молчанием. Но мы не будем молчать о ней, а постараемся помочь вам вникнуть в нее.

### Использование загрузки файлов

Функция загрузки файлов через CG1 значительно расширяет возможности программиста при разработке программ CG1. Эта функция позволяет загружать файлы на Web-сервер и предоставляет пользователям программы CGI очень простой способ передачи своих данных на Web-сайт.

Загрузка файлов, если не разобраться в ней как следует, кажется довольно ограниченной функцией, но на самом деле она позволяет очень многое. Ваши пользователи благодаря ей смогут загружать графику, звукозаписи в формате MP3, файлы конфигурации, документы HTML, документы Office и все, что вы посчитаете нужным. Подумайте обо всех этих возможностях!

Брент разработал на основе загрузки файлов через CGI целую систему управления документами (document management system — DMS). Эта DMS была не особенно большой: 250 пользователей и около 2500 документов. Она была способна управлять разными версиями одного документа, посылать по электронной почте напоминания тем пользователям, которые слишком долго задерживают свои документы, и устанав-

ливать возможности доступа в зависимости от прав данного пользователя в системе. Если документ был кем-то затребован и изменен, система отображала его имя, телефон, адрес электронной почты и многие другие данные. Система DMS полностью была основана на разрешениях. Если пользователь не имел разрешение на доступ к каталогу или файлу, он просто не видел его. Эта система заменила аналогичную "коммерческую" систему управления документами, которая стоила несколько *сотен тысяч* долларов и требовала специально выделенный сервер и системного администратора. Наша система DMS на базе Web могла работать на существующем файловом сервере и обходилась без отдельного администратора базы данных.

# Основы загрузки файлов

Мы рассказали о том, на что способна загрузка файлов, но пока не привели никаких примеров. Наш первый пример будет небольшим, но нам нужна какая-нибудь отправная точка. На рис. 7.1 показан пользовательский интерфейс, созданный в листинге 7.1.

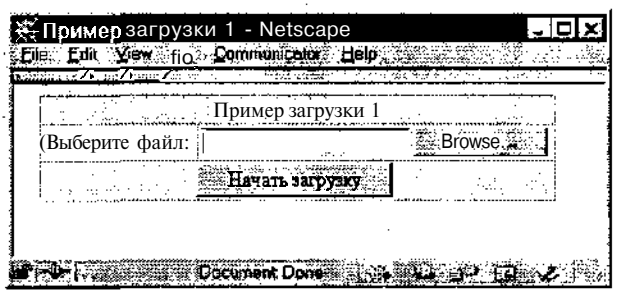



*Рис. 7.1. Пример загрузки файла*

Этот пример может показаться не особенно полезным, но не стоит беспокоиться. Хотя интерфейс пользователя по ходу этой главы останется таким же простым, компонент CGI будет все более усложняться. Эта книга посвящена CGI, а не HTML или проектированию интерфейсов пользователя, поэтому большинство интерфейсов будут очень несложными. Вы сами сможете написать такие продвинутые страницы, какие захотите, используя свои знания HTML. Ниже приведен код HTML для интерфейса первого примера.

**Листинг 7.1.** Страница ввода файла для загрузки

```
<HTML>
<HEAD><TITLE>Пример загрузки 1</TITLEx/HEAD>
<BODY BGCOLOR="#FFFFFF">
<FORM NAME="upload" METHOD=POST ACTION="/cgi-bin/upload1.cgi"
ENCTYPE="multipart/form-data">
<CENTER>
<TABLE BORDER=1 CELLSPACING=0>
<TR BGCOLOR="#E0E0E0">
<TD COLSPAN=2>
<CENTER><B>Пример загрузки 1</B></CENTER>
</TD>
</TR>
<TR>
<TD>
<B>Выберите файл:</B>
</TD>
<TD>
```

```
<INPUT TYPE="FILE" NAME="filename" VALUE="Обзор...">
</TD>
</TR>
<TR BGCOLOR="#E0E0E0">
<TD COLSPAN=2><CENTER>
<INPUT TYPE="SUBMIT" NAME="submit" VALUE="Начать загрузку">
</CENTERx/TD>
</TR>
</TABLE>
</CENTER>
</FORM>
</BODY></HTML>
```

Этот файл HTML имеет несколько заслуживающих внимания особенностей. Он был бы очень похож на код любой другой формы HTML, если бы не последний атрибут дескриптора FORM, ENCTYPE = "multipart/form-data". Принятый по умолчанию для форм **HTML** тип содержимого `application/x-www-form-urlencoded` неэффективен при передаче больших объемов двоичных данных или текста, содержащего символы, не **входящие в ASCII.** Тип содержимого `multipart/form-data` — это фактически "новый" способ построения форм **HTML. Он** не применяется для всех форм по причине необходимости поддерживать обратную совместимость со старыми Web-броузерами. Дополнительную информацию о формах HTML можно получить на `http://www.w3.org/TR/REC-html40/interact/forms.HTML`.

```
<FORM NAME="upload" METHOD=POST ACTION="/cgi-bin/upload1.cgi"
ENCTYPE="multipart/form-data">
```

Без указания `multipart/form-data` загрузка файлов не будет работать. Поэтому не забывайте делать это. Теперь давайте пройдем по коду и посмотрим, как он работает.

```
01:  #!/usr/bin/perl -Tw
02:  # upload1.cgi
03:  use strict;
04:  use CGI qw (:standard);
05:  use CGI::Carp qw(fatalsToBrowser);
```

Строки 1–5 мы уже видели во многих примерах этой книги. Они просто сообщают системе, где найти Perl, включают предупреждения и проверку на загрязнение и загружают пару необходимых модулей.

```
06:  $CGI::POST_MAX = 1024 * 250;  # Предел загрузки 250 Кбайт
```

**Строка** 6 — настройка `CGI.pm`, ограничивающая размер файлов, которые могут быть загружены на сервер. Загрузка файла начнется, даже если файл больше этого предельного значения, но, как только оно будет достигнуто, будет сгенерирована ошибка сервера. Это ограничение предотвращает атаки, вызывающие отказ в обслуживании, а также предупреждает пользователя, который может просто не понимать, что он загружает слишком большой файл. В данном случае отказ от обслуживания вызывает загрузка злонамеренным пользователем очень больших файлов с целью разрушить ваш сервер или вызвать другие проблемы, такие как заполнение жесткого диска сервера. Этот предел ограничивает *общий* объем всех загружаемых данных. Иными словами, учитывается не только выбранный для загрузки файл, но и весь текст, который был отправлен через форму.

```
07:  my $File_Name = param('filename');
```

В **строке** 7 объявляется глобальная переменная $File_Name и ей присваивается значение, переданное с вызывающей Web-страницы через функцию `CGI.pm` *param ()*.

Термин *глобальный* здесь применяется довольно свободно. По умолчанию в Perl все переменные глобально доступны. Но так как мы используем строгий синтаксис, мы должны быть более осторожны с объявлением переменных и областью их действия. Все переменные, объявленные с ключевым словом my за пределами подпрограмм, являются глобальными для программы. Все такие переменные, объявленные внутри подпрограммы или блока кода, доступны только из этой подпрограммы или блока. Чтобы облегчить распознавание области действия переменных, мы будем начинать имя глобальных переменных с прописной буквы, а имена неглобальных переменных будут включать только строчные буквы.

```
08:  my $Mime = uploadInfo ($File_Name) ->{Content-Type};
```

Строка 8 объявляет глобальную переменную $Mime и при помощи функции CGI.pm *uploadInfo()* записывает в нее тип MIME загруженного файла.

```
09: Print_Results();
```

Строка 9 вызывает подпрограмму *Print_Results()*.

```
10:  sub Print_Results {
11:    print header;
12:    print start_html('Пример загрузки файла 1');
```

Строка 10 начинает подпрограмму *Print_Results()*.

Строка И вызывает функцию CGI.pm *header ()*, которая выводит заголовок HTTP.

Строка 12 вызывает функцию *start_html()*, которая начинает код HTML результирующей страницы.

```
13:      print qq(<PRE><B>Имя файла:</B> $File_Name\n);
14:      print qq(<B>Тип MIME:</B> $Mime\n);
15:      print qq(<B>Содержимое файла:</B>\n);
```

Строки 13—15 выводят некоторые сведения о загруженном файле.

```
16:      while (<$File_Name>) { print; }
```

Строка 16 совершает цикл по $File_Name и выводит по одной строке за раз. Если после оператора *print()* не указано, *что* выводить, Perl по умолчанию выводит текущее значение переменной $_. Способности Perl удивительны — всего одна строка кода отображает все содержимое файла!

Но позвольте! Разве $File_Name — не строка? Как можно проходить ее в цикле, как будто это дескриптор файла? Все в порядке — это Линкольн Штайн придал своему CGI.pm просто замечательные возможности. Переменная, полученная с вызывающей Web-страницы, как в строке 7, если рассматривать ее как строку, будет строкой, содержащей имя файла, и, в зависимости от броузера, путь к нему. Но если рассматривать ее как дескриптор файла, она будет вести себя как дескриптор файла, и можно будет перебирать данные в ней, как в обычном файле. Спасибо, Линкольн!

```
17:      print qq(</PRE>);
18:      print end_html;
19:  }
```

Строка 17 выводит закрывающий дескриптор <PRE>.

Строка 18 выводит заключительные дескрипторы HTML (при помощи функции *end_html()* из CGI.pm).

Строка 19 завершает подпрограмму.

Вот так, всего 19 строк кода! На рис. 7.2 показан примерный результат работы этой программы.

Загруженный файл — это обычный файл текстовой "подписи". Программа CGI отображает имя файла и его содержимое. В этом примере мы даже не позаботились

сохранить файл где-нибудь на сервере, а просто вывели данные из временного файла, созданного модулем CGI.pm. Теперь попробуем загрузить графический файл и посмотрим, что произойдет (рис. 7.3).

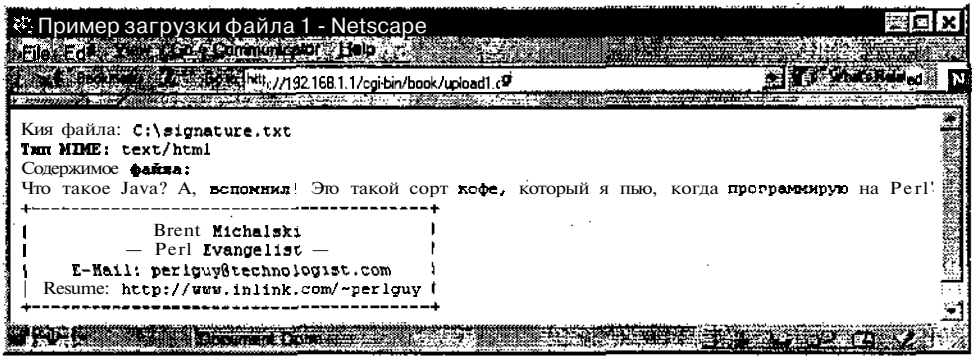



*Рис. 7.2. Результат загрузки текстового файла*

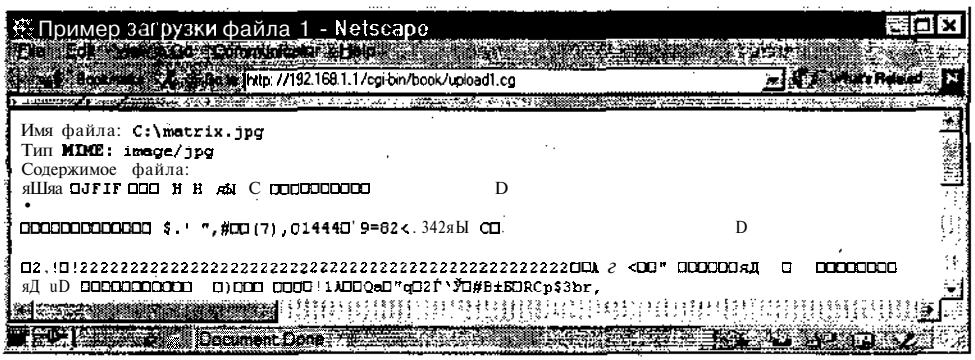



*Рис. 7.3. Результат загрузки графического файла*

Ой! Что случилось? Чтобы правильно отобразить двоичный файл, надо приложить еще немножко труда. Выше, в **строке 11** (print header) мы сообщили серверу, что вывод будет иметь тип text/html, так как это значение принято по умолчанию для функций *header()*, если в нее. не передаются параметры. Если мыустановим для сервера текстовый вывод и попытаемся передать ему двоичный файл (например, графику), мы получим кучу непонятных символов, так как сервер будет считать принятые данные текстом.

К счастью, эту проблему легко устранить, и сейчас мы как раз это и сделаем.

```
01:  #!/usr/bin/perl -Tw
02:  # upload2.cgi
03:  use strict;
04:  use CGI qw(:standard) ;
05:  use CGI::Carp qw(fatalsToBrowser);
06:  $CGI::POST_MAX = 1024 * 250;  # Предел загрузки 250 Кбайт
```

**Строки 1–6** в пояснении уже не нуждаются.

```
07:  my $File_Name = param('filename');
08:  my $Mime = uploadInfo($File_Name)->{Content-Type};
```

**Строка 7** объявляет глобальную переменную $File_Name и присваивает ей значение, переданное с вызывающей Web-страницы через функцию CGI.pm *param()*.

**Строка 8** объявляет глобальную переменную $Mime и при помощи функции CGI.pm *uploadInfo()* записывает в нее тип MIME загруженного файла. *Предполагается,* что броузеры, поддерживающие загрузку файлов, передают тип MIME загружаемого файла. Подпрограмма *uploadInfo* дает простой способ получения информации MIME.

```
09:  print header(-type=>$Mime);
10: Print_Results();
```

**Строка 9** сообщает серверу, какие данные переданы. По умолчанию для подпрограммы *header()* принят тип text/html, но он отменяется, если передать в нее параметр -type=>$Mime.

**Строка 10** вызывает подпрограмму *Print_Results*.

```
11:  sub Print_Results {
12:  ray  $data;
```

**Строка 11** начинает подпрограмму *Print_Results ()*.

**Строка 12** объявляет переменную $data. Эта переменная будет доступна только в подпрограмме, так как она была объявлена внутри подпрограммы.

```
13:    if($Mime !~ /text/) {
14:      binmode($File_Name);
15:      while (read($File_Name, $data, 1024)) { print Sdata; }
```

**В строке 13** выясняется, содержит ли значение SMime слово *text.* Если это не так (not text = true), начинается первая часть блока if...else.

В **строке** 14 через функцию *binmode ()* интерпретатору Perl сообщается, что данные в дескрипторе файла $File_Name являются двоичными. Вызов *binmode()* не требуется на машинах Unix, так как они правильно обрабатывают символы перевода строки, но на машинах Windows это необходимо. В противном случае загруженный файл, содержащий двоичные данные, может быть поврежден.

**Строка** 15 выводит данные. Так как этот файл двоичный, его нельзя вывести построчно, поскольку в двоичных файлах нет понятия "строка". Поэтому для вывода двоичных данных применяется функция *read()*. Помещенная в цикл *while ()*, она будет читать данные, пока их больше не останется. Тогда цикл закончится, и будет продолжено выполнение программы.

В функцию *read()* передаются три параметра: $File_Name — дескриптор файла, созданный в строке 7. $data — временная переменная, в которой будет храниться очередная *порция* данных. Наконец, 1024 — число байт, которые читаются в каждой итерации цикла. Итак, цикл *while ()* будет читать 1024 байта, выводить порцию данных, а затем возобновлять эти действия, пока не будут обработаны все данные.

```
16:    } else {
17:      print start_html ('Пример загрузки файла 2');
18:      print qq(<PRE>);
19:      print qq(<B>Имя файла:</B> $File_Name\n);
20:      print qq(<B>Содержимое файла:</B>\n);
```

**Строки 16—20** выполняются, если загруженный файл имел тип MIME *text.* Если это так, мы обрабатываем данные точно так же, как в предыдущем примере. По сути, **строки 17—20** точно повторяют код предыдущего примера.

```
21:      while (<$File_Name>) { print; }
```

**Строка 21** выводит текстовые данные. Она также повторяет код предыдущего примера.

```
22:      print qq(</PRE>);
23:      print end_htral;
24:    }
25: }
```

**Строка 22** выводит закрывающий дескриптор <PRE>.

**Строка 23** выводит заключительные дескрипторы **HTML.**

**Строки 24—25** завершают блок if. . .else и подпрофамму.

Итак, мы создали программу, которая позволяет пользователю загружать как текст, так и двоичные данные, и получать их отображение. Но что, если потребуется сохранить зафуженные файлы на сервере? Сейчас мы добавим функцию **сохранения** зафуженных файлов на Web-сервере, чтобы пользователь смог получить свои файлы обратно в любой момент. Затем сделаем так, чтобы пользователи могли добавлять описание для своего файла при зафузке. На рис. 7.4 показано, как тогда будет выглядеть новый интерфейс пользователя.

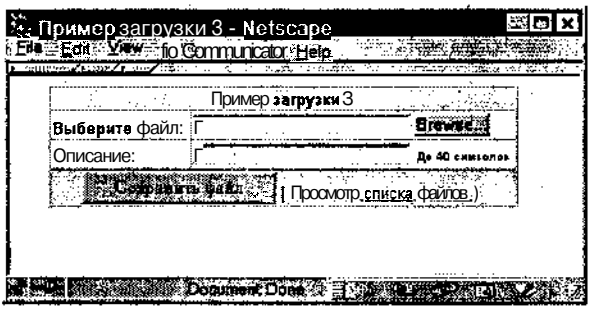



*Рис. 7.4.. Экран для загрузки файла*

Заметьте, что этот интерфейс очень похож на предыдущие два примера. Он отличается от них только дополнительным полем Описание и гиперссылкой Просмотр списка файлов. Эта гиперссылка вызывает фактически отдельную программу CGI viewfiles.cgi, которая просто отображает имена файлов с их описаниями. Листинг 7.2 содержит код HTML этого нового интерфейса пользователя.

## Листинг 7.2. Загрузка файлов с описаниями

```
<HTML>
<HEAD><TITLE>Пример загрузки 3</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF">
<FORM NAME="upload" METHOD=POST
ACTION="/cgi-bin/book/upload3.cgi"
ENCTYPE="multipart/form-data">
<CENTER>
<TABLE BORDER=1 CELLSPACING=0>
<TR BGCOLOR="#E0E0E0">
<TD COLSPAN=2><FONT FACE=ARIAL SIZE=2>
<CENTER><B>Пример загрузки 3</B></CENTER>
</FONT></TD>
</TR>
<TR>
<TD><FONT FACE=ARIAL SIZE=2>
<B>Выберите файл:</B>
</FONT></TD>
<TD><FONT FACE=ARIAL SIZE=2>
<INPUT TYPE="FILE" NAME="filename">
</FONT></TD>
</TR>
<TR>
<TD><FONT FACE=ARIAL SIZE=2>
<B>Описание:</B>
</FONT></TD>
<TD><FONT FACE=ARIAL SIZE=2>
```

```
<INPUT TYPE="text" NAME="description" MAXLENGTH=40>
<FONT FACE="ARIAL" SIZE=1>До 40 символов</FONT>
</FONT></TD>
</TR>
<TR BGCOLOR="#E0E0E0">
<TD COLSPAN=2><CENTER>
<INPUT TYPE="SUBMIT" NAME="submit" VALUE="Сохранить файл">
<FONT FACE=ARIAL SIZE=2>
[ <A HREF="/cgi-bin/book/viewfiles.cgi">Просмотр списка файлов
</A> ]</FONT>
</CENTER></TD>
</TR>
</TABLE>
</CENTER>
</FORM>
</BODY></HTML>
```

Для этого интерфейса мы создадим чуть более сложную программу, но она Останется относительно короткой. Имея менее 70 строк, эта программа будет довольно мошной системой загрузки файлов. При ее выполнении получится примерно следующий результат (рис. 7.5).

Эти данные не особенно содержательны и, вероятнее всего, не из той области, которая предназначена для пользователей. Эта страница — скорее сводка информации для удобства разработчика. В коммерческой системе представление данных должно быть более дружественным. Если щелкнуть на ссылке Просмотр файлов, появится страница, изображенная на рис. 7.6. Здесь имеется таблица, содержащая имена всех файлов в каталоге загрузки, а также их описания.

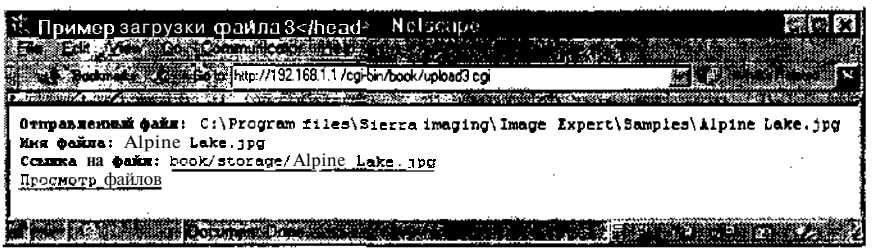



*Рис. 7.5. Результаты работы примера 3*

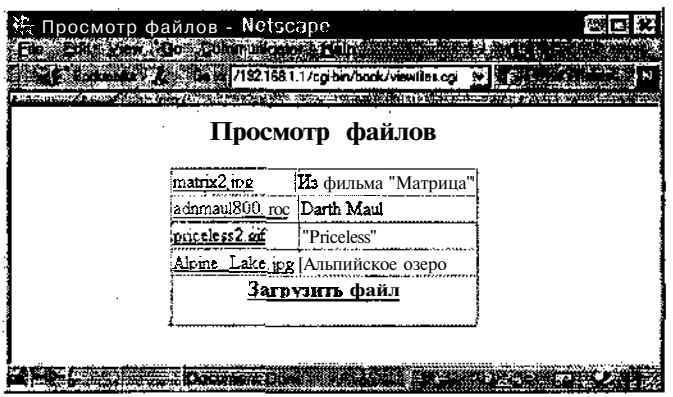



*Рис. 7.6. Экран просмотра файлов*

Сначала мы проанализируем новый код загрузки файла, а затем перейдем к программе viewfiles.cgi.

```
01:  #!/usr/bin/perl -Tw
02:  # upload3.cgi
03:  use strict;
04:  use DBI;
05:  use File::Basename;
06:  use CGI qw(:standard);
07:  use CGI::Carp qw(fatalsToBrowser);
```

Строки 1–7 должны быть уже очень знакомы вам. Исключение составляют строки 4 и 5, в которых импортируются новые модули. Модуль DBI служит для доступа к базам данных. DBI означает DataBase Interface (интерфейс базы данных).

DBI также иногда расшифровывают как DataBase Independent (независимый от базы данных), так как модуль DBI требуется для возможности доступа ко всем базам данных, а для каждого определенного типа базы данных нужен еще отдельный модуль. Модуль File::Basename используется для извлечения имен файлов из полных имен, например, myfile.cgi из /usr/local/bin/myfile.cgi.

```
08: my $Directory = "/usr/www/html/book/storage";
09: my $Url_Path = "/book/storage";
```

В строках 8–9 объявляются глобальные переменные, в которых будут храниться некоторые данные о местоположении. Такой способ сохранения данных вовсе не обязателен. Однако, если подобные строки помещаются в начале программы, это позволяет впоследствии легко узнать, в каких каталогах будет работать программа, уменьшает количество возможных ошибок и делает программу более четкой и аккуратной.

```
10:  my $File_Name = param('filename');
11:  my $Description = param('description');
12:  my $File = Get_File_Name($File_Name);
```

Строки **10–11** при помощи функции CGI.pm *param()* получают с вызывающей Web-страницы имя и описание файла.

Строка 12 вызывает функцию *Get_File_Name()* и передает в нее значение $File_Name. Возвращаемое значение сохраняется в новой переменной $File.

```
13:  $CGI::POST_MAX = 1024 * 250;  # Предел загрузки 250 Кбайт
```

Строка 13 устанавливает предел объема данных, передаваемых в программу. В результате разрешается передавать только 250 Кбайт или меньше.

```
14:  Store_Results();
15:  Store_Description();
16:  Print_Results();
```

В строках 14–16 вызываются функции, которые сохраняют результаты и описания и затем выводят результаты. Эти три функции фактически выполняют всю работу программы.

```
17:  sub Store_Description {
18:    my $DBH = DBI->connect("DBI:mysql:book","book","addison");
```

Строка 17 начинает функцию *store_Description ()*.

Строка 18 создает переменную $DBH И присваивает ей значение, возвращенное методом *DBI->connect()*. В числе значений, передаваемых в этот метод, можно заметить драйвер и имя базы данных (DBI :mysql:book), имя пользователя (book) и пароль (addison). Переменную, содержащую значение этого вызова функции, обычно принято называть $DBH. DBH означает DataBase Handle (дескриптор базы данных), а сохраняемое значение — дескриптор подключения к базе данных.

```
19:    my $sth_insert =
20:      $DBH->prepare( qq{INSERT INTO files
         (Description,FileName) VALUES (?,?)} )
21:    or die $DBH->errstr;
```

Строки **19–21** — на самом деле один длинный оператор **Perl**, в котором создается еще один дескриптор. Этот дескриптор называется `$sth_insert`, что означает StatemenT Handle for Insert (дескриптор команды для вставки).

## Совет

Рекомендуется присваивать дескрипторам команд имена, начинающиеся с `$sth`, после которого следует описание действия команды. В программе может быть несколько дескрипторов команд различного назначения, поэтому лучше, чтобы их имена были значимыми (т.е. `$sth_insert`, `$sth_update`, `$sth_delete` и т.д.).

Функция *prepare()* преобразует переданную в нее команду SQL в скомпилированную форму, которая выполняется быстрее. В MySQL и **mSQL** команды на самом деле не компилируются, а просто сохраняются, так как эти серверы баз данных не производят *подготовку* команд SQL. Оператор qq{} помешает текст в кавычки. Иногда в SQL требуется применять кавычки, а оператор qq позволяет не беспокоиться об удалении кавычек внутри кавычек.

В конце строки 20 можно заметить параметр VALUES (?,?). Вопросительные знаки — это *метки-заполнители,* применять которые очень полезно, так как, когда потребуется выполнить команду SQL, можно будет передать в функцию *execut()* значения для каждой **метки-заполнителя.** Эти значения будут подставлены вместо вопросительных знаков в команду SQL.

Наконец, в строке 21 появляется конструкция or die. Она предназначена для проверки ошибок. Если функция *prepare()* возвращает значение лжи, это говорит о какой-то ошибке, поэтому программа сразу останавливается с выдачей соответствующего сообщения.

```
22:    $sth_insert->execute($Description,$File);
23:    $DBH->disconnect;
24:  }
```

В строке 22 вызывается функция *execute()* для дескриптора `$sth_insert`. Два значения, переданные в нее, — это два поля, которые будут вставлены в базу данных.

Строка 23 производит отключение от базы данных.

Строка 24 завершает подпрограмму *store_Description ()*.

```
25:  sub Get_File_Name {
```

Строка 25 начинает подпрограмму *Get_File_Name()*.

```
26:    if ($ENV{'HTTP_LISER_ACENT'} =~ /win/i) {
27:      fileparse_set_fstype("MSDOS");
28:    }
```

Строка 26 начинает блок if. . . elsif. Здесь проверяется, содержит ли значение, переданное из переменной окружения `HTTP_USER_AGENT`, текст *win.* Если это так, выполняется код внутри блока.

В строке 27 вызывается функция *fileparse_set_fstype()* из модуля File:: Basename и устанавливается тип синтаксического анализа файла `MSDOS`. В результате программа будет считать разделителями файлов и каталогов обратную косую черту.

Строка 28 закрывает первую часть блока `if. . . elsif`.

```
29:     elsif ($ENV{'HTTP_USER_AGENT'} =~ /mac/i)  {
30:       fileparse_set_fstype ("MacOS") ;
31:     }
```

**Строка 29** выполняется, если условие в **строке 26** не выполняется. Эта строка проверяет, содержит ли HTTP_USER_AGENT текст "mac". Если это так, выполняется **строка 30** и устанавливается тип синтаксического анализа файла MacOS для систем Macintosh.

**Строка** 31 закрывает блок if. . .elsif. Раздела else здесь нет. Если оба условия не выполняются, тип синтаксического анализа просто не устанавливается и остается по умолчанию (Unix).

```
32:     my $full_name = shift;
33:     $full_name = basename($full_name);
34:     $full_name =~ s!\s!\_!g;
```

**Строка 32** через функцию *shift* получает значение, переданное в подпрограмму, и сохраняет его в скаляре $full_name.

**Строка 33** вызывает функцию *basename()* из модуля File: :Basename и передает в нее параметр $full_name.

**В строке 34** вычисляется регулярное выражение, которое заменяет все пробелы символами подчеркивания. Это делается затем, чтобы не иметь дела с пробелами в именах файлов. Такой способ удобен, прост и не требует особого труда. В регулярном выражении в качестве разделителя мы используем !, так как для обозначения пробелов нужно применять обратные косые черты (\s). Запись s ! \s ! _ ! g выглядит лучше, чем s/\s/_/g. Говорят, что это синдром "домиков из зубочисток" (строка выглядит, как несколько зубочисток, подпирающих друг друга).

```
35:     return($full_name);
36:   }
```

**Строка 35** возвращает из подпрограммы значение переменной $full_name.

**Строка 36** завершает подпрограмму *Get_File_Name()*.

```
37:  sub Store_Results {
38:     my $data;
39:     my $mime = uploadInfo($File_Name)->{'Content-Type'};
```

**Строка 37** начинает подпрограмму *Store_Results*

**Строка 38** создает переменную $data, которая будет использоваться в этой подпрограмме для временного хранения данных.

В **строке 39 в** скаляр $mime записывается тип содержимого файла, загруженного пользователем, который определяется функцией *uploadlnfo ()* модуля CGI.

```
40:       open (STORAGE, ">$Directory/$File")
41:         or die "Ошибка: SDirectory/$File: $!\n";
```

**Строки 40—41** — это опять один оператор Perl, охватывающий две строки. **В строке 40** открывается новый файл и ему присваивается дескриптор STORAGE. **В строке 41** вызывается функция *die()* и выводится сообщение об ошибке, если при открытии файла возникает проблема.

Так, если $Directory содержит значение /home/brent, а $File содержит address.txt, будет создан и открыт файл /home/brent/address.txt. Дескриптор этого файла — STORAGE.

```
42:     if ($mime !~ /text/) {
43:       binmode ($File_Name);
44:       binmode (STORAGE);
45:     }
```

**В строках 42—45** для двух дескрипторов файлов вызывается функция *binmode ()*, если значение $mime не содержит строку **text.** Это очень простой способ определить, является

ли загруженный файл текстовым или двоичным. Если файл действительно двоичный, и используемый сервер работает не в Unix, следует выполнить *binmode()* для обеих дескрипторов. Если программа всегда будет работать в среде Unix, эти строки можно опустить, так как Unix сама умеет правильно обрабатывать двоичные и текстовые данные.

```
46:     while ( read($File_Name, $data, 1024) ) {
47:       print STORAGE $data;
48:     }
```

**Строки 46—48** выполняют запись файла на диск.

**Строка 46** начинает цикл *while ()*, который продолжается, пока в файле есть данные для чтения. Функция *read()* в цикле читает порции данных. Так как загруженный файл может быть двоичным, нельзя сразу вывести файл построчно, так как двоичные файлы не содержат концы строк, как текстовые. В функцию *read()* передается три параметра: дескриптор читаемого файла, имя переменной, которая будет хранить текущую порцию данных, и, наконец, число байт, читаемое за один раз.

**Строка 47** копирует текущую порцию данных из $data в дескриптор файла STORAGE.

**Строка 48** завершает цикл *while ()*.

```
49:     close STORAGE;
50: }
```

**Строка 49** закрывает дескриптор STORAGE.

**Строка 50** завершает процедуру *Store_Results ()*.

```
51: sub Print_Results {
52:   my $link = "$Url_Path/$File";
```

**Строка 51** начинает подпрограмму *Print_Results()*.

**Строка 52** создает переменную $link, содержащую путь URL и имя файла. Путь URL отличается от реального пути по каталогам сервера, так как URL отражают местоположение в конфигурации Web-сервера и фактически не соответствуют действительному пути на сервере. Это скорее виртуальные пути, которые Web-сервер использует для поиска файла.

```
53:     print header;
54:     print start_html("Пример загрузки файла 3");
```

**В строке 53** с помощью функции *header()* модуля CGI выводится стандартный заголовок HTTP.

**В строке 54** функция этого же модуля *start_html()* выводит начальный код HTML создаваемой страницы. Строка, передаваемая в параметре, становится заголовком страницы.

```
55:     print<<HTML;
56:     <PRE>
57:     <B>Отправленный файл:</B> $File_Name
58:     <B>Имя файла:</B> $File
59:     <B>Ссылка на файл:</B> <A HREF="$link">$link</A>
60:     <A HREF="/cgi-bin/book/viewfiles.cgi">Просмотр файлов</A>
61:     </PRE>
62:     HTML
```

**Строки 55—62** — включенный документ, в котором выводится HTML, отображаемый на получаемой в результате странице. Переменные, которые можно заметить в документе, при отображении у пользователя заменяются на их значения.

```
63:     print end_html;
64: }
```

**В строке 63** функция *end_html()* из модуля CGI выводит заключительный код HTML создаваемой страницы.

**Строка 64** завершает подпрограмму *Print_Results()*.

# Просмотр файлов

Как вы помните, программа viewfiles.cgi должна была динамически создавать страницу, отображающую все файлы и их описания. Сейчас мы рассмотрим эту программу и изучим ее действие. Она невелика и не слишком сложна.

```
01:  #!/usr/bin/perl -wT
02:  # viewfiles.cgi
03:  use strict;
04:  use DBI;
05:  use CGI qw(:standard);
06:  use CGI::Carp qw(fatalsToBrowser);
```

**Строки 1–6** почти одинаковы во всех приложениях. И в этом скрипте они практически не изменяются. Единственное отличие — здесь нам не понадобится модуль File::Basename.

```
07:  my $Web_Directory = "/book/storage";
```

В **строке** 7 создается переменная, содержащая местоположение файлов.

```
08:  Print_Html_Top();
09:  Get_Descnptions();
10:   Print_Html_Bottom();
```

**Строки 8–10** вызывают три функции, образующие эту программу.

```
11:  sub Print_Row {
```

**Строка 11** начинает подпрограмму *Print_Row()*. Эта подпрограмма выводит одну строку данных, содержащую имя файла и его описание.

```
12:     my $rec = shift;
13:     ту $fname = $rec->{FileName};
14:     my $desc = $rec->{Description};
```

**Строка 12** считывает значение, переданное в подпрограмму, и сохраняет его в переменной $rec. Если для считывания используется команда shift без параметров, в переданном параметре производится сдвиг на одно значение, т.е. передается следующее значение параметра.

В **строках 13–14** создаются две переменных и им при помощи оператора стрелки присваиваются значения полей FileName и Description текущей записи.

```
15:     print qq(<TRXTD>);
16:     print qq(<A HREF="$Web_Directory/$fname">$fname</A>);
17:     print qq(</TDXTD>$desc</TDx/TR>) ;
18:  }
```

**Строки 15–17** просто выводят строку данных в таблице **HTML.**

**Строка 16** создает ссылку на файл, который был загружен.

**Строка 18** завершает подпрограмму *Print_Row()*.

```
19:  sub Get_Descriptions {
20:    my $DBH = DBI->connect("DBI:mysql:book","book","addison");
```

**Строка** 19 начинает подпрограмму *Get_Descriptions*.

**Строка** 20 создает подключение к базе данных и сохраняет дескриптор подключения в переменной $DBH.

```
21:     my $sth_fetch =
22:       $DBH->prepare( qq{SELECT * FROM files} )
          or die $DBH->errstr;
```

**Строки 21—22** — один оператор. Здесь создается новый дескриптор $sth_fetch подготовленной команды SQL, которая выбирает все записи из таблицы files базы данных. Если при подготовке команды возникает ошибка, то вызывается функция die и программа завершается.

```
23:     $sth_fetch->execute();
```

**Строка 23** — оператор *execute()*, фактически выполняющий в базе данных команду SQL, для дескриптора которой он вызывается.

```
24:     while( my $ptr = $sth_fetch->fetchrow_hashref ){
25:       Print_Row($ptr);
26:     }
27:   }
```

**Строка 24** начинает цикл *while*. В параметре цикла задано my $ptr = $sth_fetch->fetchrow_hashref. ЭТОТ оператор создает переменную $ptr, которая используется как указатель. Этот указатель заполняется возвращаемым значением функции *fetchrow_hashref()*.

Таким образом, этот цикл извлекает из базы данных по одной строке, пока в ней остаются неизвлеченные данные. Каждый раз ссылка на данные сохраняется в $ptr.

**Строка 25** вызывает функцию *Print_Row()*, которая производит вывод данных, только что принятых из базы.

**Строка 26** завершает цикл *while ()*.

**Строка 27** завершает подпрограмму *Get_Description()*.

```
28:   sub Print_Html_Top {
29:     print header;
```

**Строка** 28 начинает подпрограмму *Print_Html_Top()*. Эта подпрограмма выводит начало генерируемой страницы **HTML.**

**Строка 29** выводит стандартный заголовок HTTP.

```
30:     print<<EOT;
31:     <HTML><HEAD><TITLE>Просмотр файлов</TITLE></HEAD>
32:     <BODY BCCOLOR="#FFFFFF">
33:     <CENTER><H2>Просмотр файлов</H2><P>
34:     <TABLE BORDER=1 CELLSPACING=0>
35:     EOT
36:   }
```

**Строки 30—36** завершают подпрограмму. Здесь во включенном документе выводится начало HTML и начинается таблица.

```
37:   sub Print_Html_Bottom {
38:     print<<EOT;
39:     <TR><TD COLSPAN=2><A HREF="/book/upload3.html">
40:     <CENTER><H3><B>Загрузить файл</B></H3></CENTER></A>
41:     </TD></TR></TABLE>
42:     <P></CENTER>
43:     </BODY></HTML>
44:     EOT
45:   }
```

**Строки 37—45** образуют подпрограмму *Print_Html_Bottom()*, столь же простую, как функция *Print_Html_Top()*. Здесь во включенном документе просто выводится код HTML, необходимый для завершения генерируемой страницы.

# Загрузка нескольких файлов

Теперь у нас есть программа, позволяющая пользователям загружать двоичные или текстовые файлы и сохранять описания к каждому файлу. Мы также можем динамически отображать список файлов и их описаний. Полезность этого Web-приложения трудно переоценить. Но что, если пользователь хочет загрузить много файлов? Загрузка файлов по одному была бы очень медленной, и ваши пользователи, вероятно, не будут этим особенно довольны. В следующем примере, пользуясь существующим кодом, мы добавим в программу возможность загружать несколько файлов одновременно.

На рис. 7.7 показан приблизительный вид интерфейса пользователя из примера 4. Здесь пользователь может загрузить не менее пяти файлов. Впрочем, количество этих файлов — полностью во власти разработчика программы, но здесь для примера мы выбрали пять файлов.

Как только загрузка будет завершена, пользователь увидит сообщение, информирующее о том, что все прошло успешно. На данный момент его реализация уже не должна казаться вам особо сложной. Единственное, что придется сделать заново — это механизм загрузки нескольких файлов. Потребуется приложить немного больше труда, чтобы записать в базу данных несколько файлов и их описания, главным образом потому, что надо будет реализовать какой-то цикл, чтобы собрать всю информацию. Несмотря на это, программа займет не более 100 строк кода — очень неплохо для такого полезного приложения. Этот последний пример загрузки можно легко превратить в полнофункциональную программу для сайта Internet или intranet.

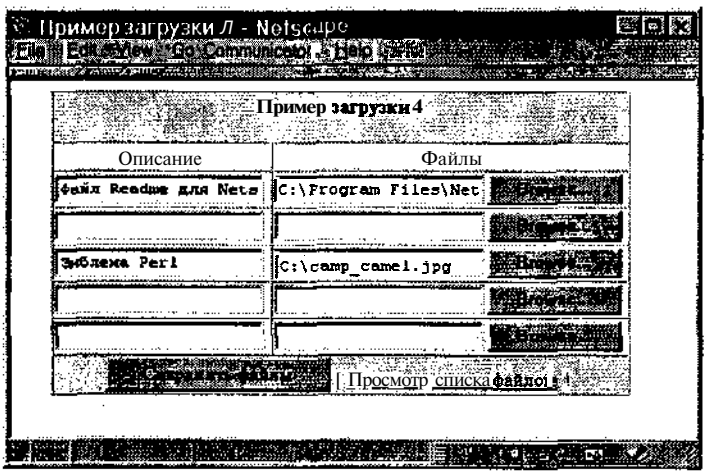



*Рис. 7.7. Экран загрузки нескольких файлов*

```
01:  #!/usr/bin/perl -Tw
02:  # upload4.cgi
03:  use strict;
04:  use DBI;
05:  use File::Basename;
06:  use CGI qw(:standard);
07:  use CGI::Carp qw(fatalsToBrowser);
```

**Строки 1–7** — такие же, как в предыдущем примере. Вероятно, вы уже заметили, что многие из наших программ, которые уже созданы и которые еще будут разработаны, построены на базе других программ. Повторное использование как можно большего объема кода помогает сэкономить время при разработке.

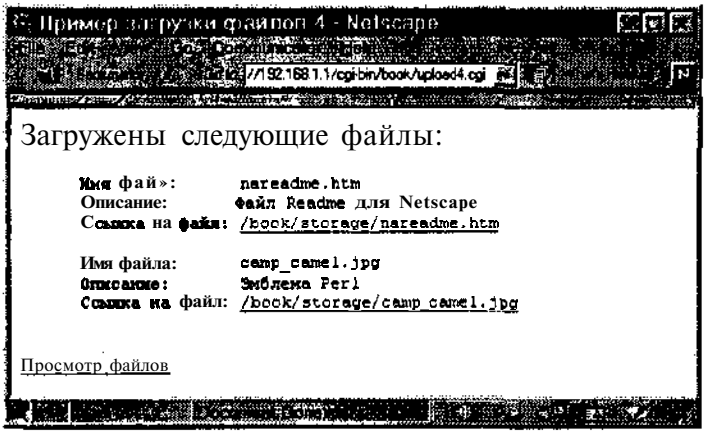



*Рис. 7.8. Результат загрузки нескольких файлов*

```
08: my SDirectory = "/usr/www/html/book/storage";
09: my $Url_Path = "/book/storage";
10: my @File_Names = param('file_name');
11: my ©Descriptions = param('file_desc');
12: my @File_Array = Of
```

**Строки 8—12** создают и инициализируют несколько переменных, которые будут постоянно использоваться в программе.

**Строка** 8 создает *глобальную* переменную SDirectory и записывает в нее полный путь к подкаталогу, в котором мы будем хранить файлы.

**Строка 9** создает *глобальную* переменную `$Url_Path` и записывает в нее путь URL для хранения файлов. Это путь, который необходим Web-серверу для обращения к файлам. Предварительно он должен быть установлен в самом Web-сервере.

**Строка 10** создает *глобальный* массив `@File_Names` и присваивает ему значения, переданные в полях имен файлов из вызывающей формы HTML. Каждое поле загрузки файла в форме называется file_name.

Если в форме HTML есть несколько полей с этим именем, как в данном случае, модуль CGI возвращает массив со всеми значениями, которые были сохранены в этих полях. Благодаря ему анализ форм становится очень простым делом.

**Строка 11** создает *глобальный* массив @Descriptions и записывает в него значения, переданные в полях описаний вызывающей формы HTML.

Строка 12 создает *глобальный* массив `@File_Array`.

Во всех этих случаях, как и раньше, мы используем понятие *глобальный* довольно свободно. Переменная, объявленная с ключевым словом `my`, областью действия имеет блок, в котором она создана. В данном случае мы создаем переменные, не входящие ни в какой блок. Таким образом, их областью действия будет вся программа. Поэтому их можно назвать глобальными.

```
13: $CGI::POST_MAX = 1024 *1500;  # Предел загрузки 1500 Кбайт
```

**Строка 13** сообщает модулю CGI, что объем загрузки не может превышать 1500 килобайт. Это предотвращает попытки загрузить очень большие файлы. В этом примере предельное значение увеличено, так как это общий размер *всех* загружаемых файлов. Здесь пользователь может загрузить целых пять файлов, и поэтому значение `POST_MAX` увеличено в несколько раз. Этот предел программист должен определять для каждого конкретного случая, так как иногда большой размер файла может быть обычным делом.

```
14:  Get_Names();
15:  Store_Descriptions();
16:  Print_Results();
```

**Строки 14—16** вызывают три подпрограммы, которые делают большую часть работы в этой программе. Вся остальная часть программы — код этих подпрограмм.

```
17:  sub Get_Names {
18:    my $counter = 0;
19:    my $full_name;
20:    my $file_name;
```

**В строках 17—20** начинается подпрограмма *Get_Names()* и создаются некоторые переменные, область действия которых ограничена этой подпрограммой. Цель данной подпрограммы — получить все имена файлов, переданные из формы HTML, и сохранить их в базе данных.

```
21:    foreach $full_name (@File_Names) {
22:      my $rec = {};
```

**Строка 21** начинает цикл *foreach (),* который обходит все значения массива `@File_Names` и сохраняет текущее значение в переменной `$full_name`.

**Строка 22** создает указатель `$rec` на хэш и очищает его содержимое. Так как эта переменная объявлена с `my`, ее область действия ограничена блоком *foreach ().*

```
23:    if ($full_name ne "") {
24:      $file_name = Get_File_Name($full_name);
```

**Строка 23** проверяет, *не* пусто ли значение `$full_name`. Если эта переменная содержит текст, выполняется блок *if ().*

**В строке 24** значение, возвращенное подпрограммой *Get_File_Name(),* сохраняется в переменной `$file_name`. Эта подпрограмма удаляет все данные пути и оставляет только имя файла.

```
25:      $rec->{file_name} = $file_name;
26:      $rec->{fun_name} = $full_name;
27:      $rec->{description} = $Descriptions[$counter];
```

В **строке** 25 значение *$file_name* записывается под ключом `file_name` безымянного хэша, на который указывает `$rec`. Такой **способ** может показаться несколько странным, но он прекрасно подходит для хранения элементов данных наподобие записей. Это напоминает использование структур в C.

**Строка 26** сохраняет значение `$full_name` под ключом `full_name` безымянного хэша.

**Строка 27** сохраняет значение элемента массива `$Descriptions` с индексом `$counter` под ключом `description` безымянного хэша.

Теперь у нас есть "запись", содержащая имя файла, полное имя и описание файла. Мы должны сохранить ее в массиве, чтобы с этими данными было легче работать далее в программе.

```
28:      push @File_Array, $rec;
29:      Store_File($fun_name, $file_name);
30:    }
```

**Строка** 28 ссылка на безымянный хэш из `$rec` помещается в следующий элемент `@File_Array`.

**Строка 29** вызывает подпрограмму *Store_File()* и передает в нее значения `$full_name` и `$file_name`. Эта подпрограмма сохраняет информацию в базе данных.

**Строка 30** закрывает блок *if.*

```
31:        $counter ++;
32:      }
33:   }
```

**В строке 31** происходит приращение переменной `$counter`. Это выполняется *вне* блока if, так как счетчик надо увеличивать и в том случае, если значение `$full_name` пусто.

Поля в форме HTML не обязательно должны быть заполнены последовательно; пользователь может поместить данные в первой и третьей строках и оставить вторую строку пустой. Обходя каждый элемент массива, мы удостоверимся, что получены все данные, которые отправил пользователь.

**Строка** 32 завершает цикл *foreach ()*.

**Строка 33** завершает подпрограмму *Get_Names()*.

```
34:  sub Store_Descriptions {
35:     my $temp;
36:     my $DBH = DBI->connect(
          "DBI:mysql:book","book","addison" );
```

**В строке 34** начинается подпрограмма *store_Descriptions ()*.

**Строка 35** объявляет переменную `$temp`. Эта переменная служит для хранения ссылки на хэш, который хранится в переменной `@File_Array`.

**Строка 36** производит подключение к базе данных book.

```
37:     my $sth_insert =
38:       $DBH->prepare( qq{INSERT INTO files
          (Description, FileName) VALUES (?,?)} )
39:       or die $DBH->errstr;
```

**В строке 37** создается скаляр, в который записывается дескриптор команды SQL, созданной в **строке 38.**

**Строка 39** — оператор **die,** который выполняется, если при подготовке инструкции SQL возникает ошибка. Общее назначение трех этих строк — получить команду SQL, которая вставляет данные в подготовленную базу. Когда нам потребуется передать в нее какие-то данные, надо будет только вызвать функцию `execute ()`.

```
40:     foreach $temp (@File_Array) 1
41:       $sth_insert->execute( $temp->{description},
            $temp->{file_name} );
42:     }
```

**Строка** 40 начинает цикл *foreach (),* который обходит каждый элемент массива `@File_Array` и сохраняет текущее значение в переменной $temp.

**В строке 41** вызывается функция *execute()* для дескриптора `$sth_insert`, созданного в **строке 37. В** эту функцию с помощью оператора стрелки передаются имя и описание файла. Этот оператор применяется, так как `$temp` хранит ссылку на хэш, который был создан в строках 25—27. Оператор стрелки позволяет определять ключи значений, к которым надо обратиться.

**Строка 42** завершает цикл *foreach ()*.

```
43:     $DBH->disconnect;
44:  }
```

**Строка** 43 производит отключение от базы данных.

**Строка 44** завершает подпрограмму *Store_Description ()*.

```
45:  sub Get_File_Name {
```

В **строке** 45 начинается подпрограмма *Get_File_Name()*.

```
46:     if ( $ENV{'HTTP_USER_AGENT'} =~ /win/i ) {
47:        fileparse_set_fstype("MSDOS");
48:     }
```

**Строка 46** проверяет, содержит ли переменная окружения HTTP_USER_AGENT текст *win*. Если да, то файл, скорее всего, пришел из системы Windows, поэтому выполняется блок if.

**В строке 47** устанавливается тип файловой системы MSDOS.

**Строка 48** закрывает первую часть блока if...elsif.

```
49:     elsif { $ENV{'HTTP_USER_AGENT'} =~ /mac/i ) {
50:        fileparse_set_fstype("MacOS");
51:     }
```

**Строка 49** выполняется, если условие в **строке** 46 ложно. Теперь мы проверяем, работает ли пользователь в системе Macintosh.

**В строке 50** устанавливается тип файловой системы Macos.

**Строка 51** завершает блок if...elsif.

```
52:     my $full_name = shift;
53:     $full_name = basename($full_name);
54:     $full_name =~ s!\s!\_!g;
```

**Строка** 52 через функцию *shift* получает значение, переданное в подпрограмму, и сохраняет его в скаляре $full_name.

**Строка 53** вызывает функцию *basename ()* из модуля File::Basename и передает в нее параметр $full_name.

**В строке 54** вычисляется регулярное выражение, которое заменяет все пробелы символами подчеркивания. Это делается затем, чтобы не иметь дела с пробелами в именах файлов. Такой способ удобен, прост и не требует особого труда.

```
55:     return($full_name);
56: }
```

**Строка 55** возвращает из подпрограммы значение $full_name.

**Строка 56** завершает подпрограмму *Get_File_Name*.

```
57: sub Store_File{
58:    my $file_handle = shift;
59:    my $file_name = shift;
60:    my $data;
```

**Строка** 57 начинает подпрограмму *store_File()*.

**В строках 58—59** объявляются переменные и через *shift ()* в них считываются значения, переданные в подпрограмму.

**Строка 60** объявляет еше одну переменную, которая будет использоваться для временного хранения данных.

```
61:     my $mime = uploadInfo($file_handle)->{Content-Type};
```

Строка 61 объявляет переменную $mime и записывает в нее тип содержимого загруженного файла, полученный функцией *uploadInf()* из CGI.pm.

```
62:     open (STORAGE, ">$Directory/$file_name")
63:        or die "Ошибка: $!\n";
```

Строки 62—63 — один оператор Perl, занимающий две строки. В **строке 62** мы открываем новый файл и присваиваем ему дескриптор STORAGE. ЕСЛИ при открытии

файла происходит ошибка, в **строке 63** вызывается функция ***die ()*** и выводится соответствующее сообщение.

Например, если $Directory содержит значение /home/brent, а $File содержит address.txt, будет создан и открыт файл /home/brent/address.txt. Дескриптор этого файла — STORAGE.

```
64:     if ($mime !~ /text/) {
65:       binmode ($File_Name);
66:       binmode (STORAGE);
67:     }
```

**В строках 64—67** для двух дескрипторов файлов вызывается функция *binmode()*, если значение $mime не содержит строку text. Это очень простой способ определить, является ли загруженный файл текстовым или двоичным. Если файл действительно двоичный и используемый сервер работает не в Unix, следует выполнить *binmode ()* для обоих дескрипторов. Если программа всегда будет работать в среде Unix, эти строки можно опустить, так как Unix сама умеет правильно обрабатывать двоичные и текстовые данные.

```
68:   while( read($File_Name, $data, 1024) ) {
69:     print STORAGE $data;
70:   }
```

**Строки 68—70** выполняют запись файла на диск.

**Строка 68** начинает цикл *while (),* который продолжается, пока в файле есть данные для чтения. Функция *read()* в цикле читает порции данных. Так как загруженный файл может быть двоичным, нельзя сразу вывести файл построчно, так как двоичные файлы не содержат концы строк, как текстовые. В функцию *read()* передается три параметра: дескриптор читаемого файла, имя переменной, которая будет хранить текущую порцию данных, и, наконец, число байт, читаемое за один раз.

**Строка 69** копирует текущую порцию данных из $data в дескриптор файла STORAGE.

**Строка 70** завершает цикл while *()*.

```
71:   close STORAGE;
72: }
```

**Строка 71** закрывает дескриптор STORAGE.

**Строка 72** завершает процедуру *Store_Results*.

```
73: sub Print_Results {
74:   my $temp;
```

**Строка 73** начинает подпрограмму *Print_Results*.

**Строка 74** объявляет переменную $temp.

```
75:   print header;
76:   print start_html("Пример загрузки файлов 4");
77:   print h2("Загружены следующие файлы:");
```

**В строке 75** с помощью функции header *()* модуля CGI выводится стандартный заголовок HTTP.

**В строке 76** функция этого же модуля *start_html()* выводит начальный код HTML создаваемой страницы. Строка, передаваемая в параметре, становится заголовком страницы.

В **строке 77** функция *h2* CGI.pm выводит верхний колонтитул страницы. Эта функция помещает переданную в нее строку в дескрипторы <H2>.

```
78:     foreach $temp (@File_Array) {
79:       my $link = "$Url_Path/$temp->{file_name}";
```

**В строке 78** начинается цикл *foreach ()*, который обходит каждый элемент в `@File_Array` и сохраняет текущее значение в переменной $temp. Этот цикл служит для вывода информации о каждом файле на генерируемой странице **HTML.**

**Строка 79** создает переменную $link и сохраняет в ней текущий путь URL и имя файла. Эти данные в дальнейшем будут использованы для создания ссылки на файл.

```
80:       print<<HTML;
81:       <PRE>
82:       <B>Имя файла:</B>       $temp->{filename}
83:       <B>Описание:</B>        $temp->{description}
84:       <B>Ссылка на файл:</B> <A HREF="$link">$link</A><P>
85:       </PRE>
86:       HTML
87:     }
```

**Строка 80** начинает "включенный документ", в котором выводится код HTML. Строка `HTML` здесь служит закрывающим дескриптором включенного документа.

**Строки 81—85** — просто HTML, который отображается на созданной странице.

**Строка** 86 завершает включенный документ, а **строка 87** завершает цикл *foreach ()*.

В строках 82 **и** 83 с помощью оператора стрелки выводятся значения в ключах имени файла и описания. Этот оператор нужен для разыменования ссылки на безымянный кэш, в котором хранятся данные.

```
88:     . print qq(\n<A HREF="/cgi-bin/book/viewfiles.cgi">
          Просмотр файлов</A>);
89:       print end_html;
90:   }
```

**Строка 88** выводит ссылку на программу просмотра файлов.

В **строке 89** функция `end_html()`из модуля CGI выводит заключительный код HTML создаваемой страницы.

**Строка 90** завершает подпрограмму *Print_Resul().*

Наш последний пример загрузки файлов закончен. Он построен на базе всего материала, рассмотренного в трех предыдущих примерах. Эта программа, возможно, содержит не все особенности, которые вы бы хотели, но она может послужить вам серьезным подспорьем на пути создания собственных приложений, функции которых включают загрузку файлов. Полученные вами знания позволяют еще более усовершенствовать эти примеры и повысить ваш уровень в программировании на Perl.

# Упражнения

- Спроектируйте более удобный пользовательский интерфейс и напишите программу загрузки файлов для вашего сайта.
- Добавьте функции загрузки файлов в программу электронной почты, приведенную в главе 10.
- Создайте программу для загрузки и сохранения информации о ваших любимых записях MP3.

# Листинги

### Листинг 7.3. upload1.cgi

```
01:  #!/usr/bin/perl -Tw
02:  # upload1.cgi
03:  use strict;
04:  use CGI qw (:standard);
05:  use CGI::Carp qw(fatalsToBrowser);
06:  my $File_Name = param('filename');
07:  my $Mime = uploadInfo($File_Name)->{Content-Type};
08:  $CGI::POST_MAX = 1024 * 250;  # Предел загрузки 250 Кбайт
09:  Print_Results();
10:  sub Print_Results {
11:    print header;
12:    print start_html('Пример загрузки файла 1');
13:    print qq(<PRE><B>Имя файла:</B> $File_Name\n);
14:    print qq(<B>Тип MIME:</B> $Mime\n);
15:    print qq(<B>Содержимое файла:</B><XMP>\n\n);
16:    while (<$File_Name>) { print; }
17:    print qq(</XMP></PRE>);
18:    print end_html;
19:  }
```

### Листинг 7.4. upload2.cgi

```
01:  #!/usr/bin/perl -Tw
02:  # upload2.cgi
03:  use strict;
04:  use CGI qw(:standard);
05:  use CGI::Carp qw(fatalsToBrowser);
06:  $CGI::POST_MAX = 1024 * 250;  # Предел загрузки 250 Кбайт
07:  my $File_Name = param('filename');
08:  my $Mime = uploadInfo($File_Name)->{Content-Type};
09:  print header(-type=>$Mime);
10:  Print_Results();
11:  sub Print_Results {
12:    my $data;
13:    if($Mime !~ /text/) {
14:      binmode($File_Name);
15:      while (read($File_Name, $data, 1024)) { print $data; }
16:    } else {
17:      print start_html ('Пример загрузки файла 2');
18:      print qq(<PRE>);
19:      print qq(<B>Имя файла:</B> $File_Name\n);
20:      print qq(<B>Содержимое файла:</B>\n);
21:      while (<$File_Name>) { print; }
22:      print qq(</PRE>};
23:      print end_html;
24:    }
25:  }
```

**Листинг 7.5. upload3.cgi**

```
01: #!/usr/bin/perl -Tw
02: # upload3.cgi
03: use strict;
04: use DBI;
05: use File::Basename;
06: use CGI qw(:standard);
07: use CGI::Carp qw(fatalsToBrowser);
08: my $Directory = "/usr/www/html/book/storage";
09: my $Url_Path = "/book/storage";
10: my $File_Name = param('filename');
11: my $Description = param('description');
12: 'my $File = Get_File_Name($File_Name);
13: $CGI::POST_MAX = 1024 * 250;  # Предел загрузки 250 Кбайт
14: Store_Results();
15: Store_Description();
16: Print_Results();
17: sub Store_Description {
18:   my $DBH = DBI->connect("DBI:mysql:book","book","addison");
19:   my $sth_insert =
20:     $DBH->prepare( qq{INSERT INTO files
      (Description,FileName) VALUES (?,?)} )
21:   or die $DBH->errstr;
22:  $sth_insert->execute($Description,$File) ;
23:   $DBH->disconnect;
24: }
25: sub Get_File_Name {
26:   if ($ENV{'HTTP_LISER_ACENT'} =~ /win/i) {
27:     fileparse_set_fstype("MSDOS") ;
28:   }
29:   elsif ($ENV{'HTTP_USER_AGENT'} =~ /mac/i) {
30:     fileparse_set_fstype("MacOS");
31:   }
32:   my $full_name = shift;
33:   $full_name = basename ($full_name);
34:   $full_name =~ s!\s!\_!g;  # Пробелы заменяются на _
35:   return($full_name);
36: >
37: sub Store_Results {
38:   my $data;
39:   my $mime = uploadInfo($File_Name)->{'Content-Type'};
40:   open (STORAGE, ">$Directory/$File")
41:     or die "Ошибка: $Directory/$File: $!\n";
42:   if ($mime !~ /text/) {
43:     binmode ($File_Name);
44:     binmode (STORAGE);
45:   }
46:  while ( read($File_Name, $data, 1024) ) {
47:     print STORAGE $data;
48:   }
49:   close STORAGE;
50: }
51: sub Print_Results {
52:   my $link = "$Url_Path/$File";
53:   print header;
54:   print start_html("Пример загрузки файла 3");
55:   print<<HTML;
```

```
56:     <PRE>
57:     <B>Отправленный файл:</B> $File_Name
58:     <B>Имя файла:</B> $File
59:     <B>Ссылка на файл:</B> <A HREF="$link">$link</A>
60:    <A HREF="/cgi-bin/book/viewfiles.cgi">Просмотр файлов</A>
61:     </PRE>
62:     HTML
63:     print end_html;
64: }
```

## Листинг 7.6. viewfiles.cgi

```
01: #!/usr/bin/perl -wT
02: # viewfiles.cgi
03: use strict;
04: use DBI;
05: use CGI qw(:standard);
06: use CGI::Carp qw(fatalsToBrowser);
07: my $Web_Directory = "/book/storage";
08: Print_Html_Top();
09: Get_Descnptions();
10: Print_Html_Bottom();
11: sub Print_Row {
12:   my $rec = shift;
13:   my $fname = $rec->{FileName};
14:   my $desc = $rec->{Description};
15:   print qq(<TR><TD>);
16:    print qq(<A HREF="$Web_Directory/$fname">$fname</A>);
17:    print qq(</TD><TD>$desc</TD></TR>);
18: }
19: sub Get_Descriptions {
20:    my $DBH = DBI->connect("DBI:mysql:book","book","addison");
21:    my $sth_fetch =
22:      $DBH->prepare( qq(SELECT * FROM files) )
       or die $DBH->errstr;
23:    $sth_fetch->execute();
24:    while( my $ptr = $sth_fetch->fetchrow_hashref ){
25:      Print_Row($ptr);
26:    }
27: }
28: sub Print_Html_Top {
29:    print header;
30:    print<<EOT;
31:    <HTML><HEAD><TITLE>Просмотр файлов</TITLE></HEAD>
32:    <BODY BCCOLOR="#FFFFFF">
33:    <CENTER><H2>Просмотр файлов</H2><P>
34:    <TABLE BORDER=1 CELLSPACING=0>
35:    EOT
36: }
37: sub Print_Html_Bottom {
38:    print<<EOT;
39:    <TR><TD COLSPAN=2><A HREF="/book/upload3.html">
40:    <CENTER><H3><B>Загрузить файл</B></H3></CENTER></A>
41:     </TDx/TRx/TABLE>
42:    <Px/CENTER>
43:    </BODY></HTML>
44:   EOT
45: }
```

Листинг 7.7. **upload4.cgi**

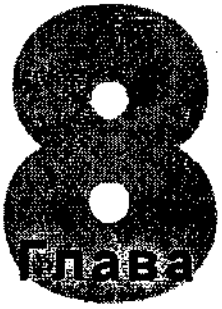
```
01: #!/usr/bin/perl -Tw
02: # upload4.cgi
03 : use strict;
04: use DBI;
05: use File::Basename;
06: use CGI qw(:standard);
07: use CGI::Carp qw(fatalsToBrowser);
08: my $Directory = "/usr/www/html/book/storage";
09: my $Url_Path = "/book/storage";
10: my @File_Names = param('file_name');
11:my @Descriptions =param( 'file_desc ');
12: my @File_Array = {};
13:  $CGI::POST_MAX =1024 *1500;  # Предел загрузки 1500 Кбайт
14:  Get_Names();
15: Store_Descriptions();
16:  Print_Results();
17:  sub Get_Names {
18:    my $counter = 0;
19:    my $full_name;
20:    my $file_name;
21:    foreach $full_name (@File_Names) {
22:      my $rec = {};
23:      if ($full_name ne "") {
24:        $file_name = Get_File_Name($full_name);
25:         $rec->{file_name} = $file_name;
26:         $rec->{fun_name} = $full_name;
27:         $rec->{description} = $Descriptions[$counter];
28:         push @File_Array, $rec;
29:         Store_File($fun_name, $file_name);
30:      }
31:      $counter ++;
32:    }
33:  }
34:  sub Store_Descriptions {
35:    my $temp;
36:    my $DBH = DBI->connect (
       "DBI:mysql:book","book","addison" );
37:    my $sth_insert =
38:      $DBH->prepare( qq{INSERT INTO files
       (Description, FileName) VALUES (?,?)} )
39:      or die $DBH->errstr;
40:    foreach $temp (@File_Array) {
41:      $sth_insert->execute( $temp->{description},
         $temp->{file_name} );
42:    }
43:    $DBH->disconnect;
44:  }
45:  sub Get_File_Name {
46:    if ( $ENV{'HTTP_USER_AGENT'} =~ /win/i ) {
47:      fileparse_set_fstype("MSDOS");
48:    }
49:    elsif { $ENV{'HTTP_USER_AGENT'} =~/mac/i ) {
50:      fileparse_set_fstype("MacOS");
51:    }
52:    my $full_name = shift;
53:    $full_name = basename($full_name);
```

```
54:     $full_name =~ s!\s!\_!g;
55:     return($full_name);
56:   }
57:   sub Store_File(
58:   ray  $file_handle = shift;
59:     my $file_name = shift;
60:   ray  $data;
61:    my $mime = uploadInfo ($file_handle) ->{Content-Type};
62:     open (STORAGE, ">$Directory/$file_name")
63:       or die "Ошибка: $!\n";
64:     if ($mime !~ /text/) {
65:       binmode ($File_Name);
66:       binmode (STORAGE);
67:     }
68:    while( read($File_Name, $data, 1024) ) {
69:       print STORAGE $data;
70:     }
71:     close STORAGE;
72:   }
73:   sub Print_Results {
74:     my $temp;
75:     print header;
76:     print start_html("Пример загрузки файлов 4");
77:     print h2("Загружены следующие файлы:");
78:     foreach $temp (@File_Array) {
79:      my $link = "$Url_Path/$temp->{file_name}";
80:       print<<HTML;
81:       <PRE>
82:       <B>Имя файла:</B>       $temp->{filename}
83:       <B>Описание:</B>        $temp->(description}
84:       <B>Ссылка на файл:</B> <A HREF="$link">$link</A><P>
85:       </PRE>
86:       HTML
87:     }
88:     print qq(\n<A HREF="/cgi-bin/book/viewfiles.cgi">
         Просмотр файлов</A>);
89:     print end_html;
90:   }
```

Листинг 7.8. Файл upload4 . html

```
<HTML>
<HEAD><TITLE>Пример загрузки 4</TITLEx/HEAD>
<BODY BGCOLOR="#FFFFFF">
<FORM NAME="upload" METHOD=POST
   ACTION="/cgi-bin/book/upload4.cgi"
ENCTYPE="multipart/form-data">
<CENTER>
<TABLE BORDER=1 CELLSPACING=0>
<TR BGCOLOR="#E0E0E0"><TD COLSPAN=2>
<CENTER><B>Пример загрузки 4</H2x/CENTER>
</TDx/TR>
<TR><TD>
<CENTER><B>Описание</B></CENTER>
</TD><TD>
<CENTER><B>Файлы</B></CENTER>
</TDx/TR>
```

```
<TR><TD>
<INPUT TYPE="text" NAME="file_desc" MAXLENGTH=40>
</TDxTD>
<INPUT TYPE="file" NAME="file_name" MAXLENGTH=40>
</TDx/TR>
<TR><TD>
<INPUT TYPE="text" NAME="file_desc" MAXLENGTH=40>
</TDxTD>
<INPUT TYPE="file" NAME="file_name" MAXLENGTH=40>
</TDx/TR>
<TR><TD>
<INPUT TYPE="text" NAME="file_desc" MAXLENGTH=40>
</TDxTD>
<INPUT TYPE="file" NAME="file_name" MAXLENGTH=40>
</TDx/TR>
<TR><TD>
<INPUT TYPE="text" NAME="file_desc" MAXLENGTH=40>
</TD><TD>
<INPUT TYPE="file" NAME="file_name" MAXLENGTH=40>
</TDx/TR>
<TR><TD>
<INPUT TYPE="text" NAME="file_desc" MAXLENGTH=40>
</TD><TD>
<INPUT TYPE="file" NAME="file_name" MAXLENGTH=40>
</TDx/TR>
<TR BGCOLOR="#E0E0E0"><TD COLSPAN=2><CENTER>
<INPUT TYPE="SUBMIT" NAME="submit" VALUE="Сохранить файлы">
[ <A HREF="/cgi-bin/book/viewfnes.cgi ">
Просмотр списка файлов</A> ]
</CENTERx/TDx/TR>
</TABLE></CENTERx/FORM>
</BODY></HTML>
```




# Отслеживание щелчков

## Введение

Отслеживание количества щелчков может дать полезную информацию о том, что делают посетители на Web-сайте. *Щелчком* мы называем ситуацию, когда кто-то щелкает на гиперссылке на вашей Web-странице и переходит на страницу, связанную с этой ссылкой. Что означает для вас отслеживание щелчков? Скажем, у вас есть несколько страниц, на которых предлагается программное обеспечение (надо надеяться, с открытыми исходными кодами), и вы хотите знать, какой популярностью пользуется та или иная программа, т.е. сколько раз ее загрузили. Эту задачу легко может выполнить скрипт CGI, который в сущности будет посредником между гиперссылкой и окончательным ее адресатом. С точки зрения разработчика программы, нам надо просто знать, по каким ссылкам переходят посетители с наших Web-страниц. Может быть чрезвычайно полезно знать, в каких программах они наиболее заинтересованы, и на чем вы должны сконцентрировать свои усилия. В данной главе рассматриваются скрипты, имеющие только этот тип функциональных возможностей.

В первом примере этой главы будет показано, как решить эту задачу в самой простой форме отслеживания щелчков на ссылках. Здесь демонстрируется передача URL в скрипт CGI, создание или обновление соответствующей записи в базе данных и сохранение в ней количества щелчков для данного URL. Два других примера расширяют эти функциональные возможности и реализуют рекламное приложение с меняющимися баннерами. Это приложение содержит два скрипта: первый по случайному закону выбирает из базы данных изображение и показывает его, а второй отслеживает щелчки на нем.

В этой главе используются следующие прагмы и модули:

- strict
- DEI
- CGI
- POSIX
- Image::Size

# Пример: простое отслеживание щелчков

Следующий пример — простое приложение, предназначенное для отслеживания щелчков посетителей на ссылке Web-страницы. Чтобы связать это приложение с реальной жизнью, мы применим его для описанной выше ситуации отслеживания количества загружаемых с Web-сайта программ.

Предположим, что на вашей Web-странице есть несколько ссылок на дистрибутивы разных программ. Некоторые из них могут помещаться вместе со страницей, а другие могут храниться на других, удаленных Web-сайтах. Задача этого приложения состоит в том, чтобы определить, сколько раз посетители щелкают на ссылках, соответствующих каждой программе. Учтите, что эта ситуация — только одна из возможных, и данный скрипт можно применять для отслеживания щелчков на любых ссылках. В этом приложении используется один скрипт и одна простая таблица базы данных.

Сначала мы рассмотрим эту таблицу. В листинге 8.1 показана команда SQL, которая ее создает. Имя таблицы — `click_tracker`.

Листинг 8.1. Команда создания таблицы **`click_tracker`**

```
CREATE TABLE click_tracker (
  PAGE varchar(255) NOT NULL,
  COUNT int(11) NOT NULL,
  LAST_ACCESS datetime
);
CREATE UNIQUE INDEX page_index on click_tracker (PAGE);
```

Эта простая таблица содержит лишь три поля. В первом поле PAGE мы будем хранить идентификатор каждой отслеживаемой страницы. Страницы будут динамически добавляться в таблицу при щелчке на гиперссылке, содержащей URL. Параметр NOT NULL для этого поля задан затем, чтобы никто не мог добавлять недопустимые поля и вставлять пустые записи. Например, в нормальной ситуации данный скрипт (`track.cgi`) будет вызываться примерно такой командой:

```
http://www.you.com/cgi-bin/track.cgi?url=downloads/file
```

Но кто-нибудь, считающий себя хакером, может попытаться вызвать ошибку при вставке или вставить пустое значение, задав следующую команду:

```
http://www.you.com/cgi-bin/track.cgi?url=dow'nload's/file
```

Этот URL мог бы привести к вставке в поле PAGE базы данных значения NULL, НО этого не произойдет, так как мы задали для этого поля параметр NOT NULL. Это означает, что новая запись обязательно должна содержать данные в PAGE. В результате попытка вставки записи будет неудачной и пользователь просто будет направлен на несуществующую страницу.

Второе поле COUNT является целочисленным и будет содержать количество щелчков на данном URL. Для этого поля также задано NOT NULL, чтобы гарантировать, что

в нем всегда будут данные. Третье и последнее поле LAST_ACCESS типа datetime будет содержать время последнего щелчка на ссылке.

Наконец, мы создаем уникальный индекс для поля PAGE, что позволяет избежать появления повторяющихся строк с одним URL. В скрипте так или иначе должна быть принята такая мера, но не мешает и дополнительно позаботиться о целостности данных в самой базе.

```
01:  #!/usr/bin/perl -wT
02:  # track.cgi
03:  use strict;
04:  use DBI;
05:  use CCI qw(:standard);
06:  use POSIX;
```

**Строки 1–6** указывают путь к Perl, задают строгий синтаксис и режим проверки на загрязнение и загружают все модули, которые мы будем использовать в этом скрипте.

```
07:  my $dbname = 'book';
08:  my $dbhost = 'localhost';
```

**В строках 7 и 8** указывается имя используемой базы данных и имя компьютера, на котором она располагается.

```
09:  my $dsn = "DBI:mysql:database=$dbname;host=$dbhost";
10:  my $dbh=DBI->connect($dsn,"user","password");
```

**В строках 9 и 10** производится подключение к базе данных. **Строка** 9 определяет источник данных — в данном случае базу данных MySQL под названием book, находящуюся на машине localhost. **В строке 10** выполняется собственно попытка подключения с помощью метода DBI *connect ()*. Результат подключения сохраняется в переменной $dbh.

```
11:  if (!defined($dbh)) {
12:    print header;
13:  print "\nошибка: Не удалось подключиться к базе
       данных MySQL:\n";
14:  print DBI->errmsg;
15:  print "-" x 25;
16:  exit;
17: }
```

**В строках 11–17** выполняется небольшая проверка ошибок. Если подключение прошло неудачно, метод *connect <)* возвращает значение undef, что проверяется в **строке 11.** Если это так, т.е. переменная $dbh не определена, то подключение потерпело неудачу.

**В строках 12–15** выводится сообщение о неудаче подключения, отображается соответствующая страница ошибки, и скрипт завершает работу. Заметьте, что **строка 12** выводит заголовок HTML. Так как перед этим блоком кода не выводится никакой заголовок, это необходимо, чтобы сообщение об ошибке было должным образом отображено в броузере. В реальном, не учебном скрипте может быть целесообразнее не выводить это сообщение и завершать скрипт, а фиксировать ошибку в журнале и направлять броузер на нужный URL. Мы предоставляем сделать это читателю.

```
18:  my $url = param('url');
```

**В строке 18** переменной $url присваивается значение одноименного параметра. Откуда мы получаем это значение? Чтобы этот скрипт мог отслеживать щелчки на определенном URL, в него должно передаваться имя отслеживаемой страницы. Например, если мы отслеживаем загрузку программ, и один из дистрибутивов называется my_program.tar.gz, гиперссылка на него на странице должна выглядеть примерно так:

```
<A HREF="/cgi-bin/track.pl?url=my_program.tar.gz">
My Program v0.1</A>
```

Теперь ясно, откуда появляется параметр url! Значение этого параметра, my_program.tar.gz, сохраняется в $url. В ходе анализа этого примера мы покажем, как отслеживать локальные URL и удаленные URL.

```
19:  my $time = strftime("%Y-%m-%d %I:%M:%S", localtime);
```

В строке 19 данные времени форматируются методом POSIX *strftime ()* в удобный для чтения вид для последующей вставки в базу данных. Дата и время записываются в формате YYYY-MM-DD HH:MM:SS. Большая часть баз данных, включая MySQL, хорошо принимает в поля datetime строки даты/времени формата POSIX.

```
20:  my Squery = qq(update click_tracker set
       COUNT=COUNT+1, LAST_ACCESS='$time' where PAGE='$url');
```

Строка 20 инициализирует переменную Squery запросом SQL, предназначенным для обновления базы данных. При обновлении в поле LAST_ACCESS записи, поле PAGE которой соответствует параметру url, записывается новое время, а значение поля COUNT этой записи увеличивается на 1.

```
21:  my $sth = $dbh->prepare($query);
```

Строка 21 методом prepare() подготавливает запрос SQL и инициализирует переменную дескриптора команды $sth.

```
22:  if ($5th->execute < 1) {
23:    $query = qq(insert into click_tracker (PAGE, COUNT,
         LAST_ACCESS) Values ('$url',1, 'Stime'));
24:    $sth = $dbh->prepare($query);
25:    $sth->execute;
26: }
```

Строки 22—26 выполняются, если запрос на обновление завершается неудачно. Неудача обновления означает, что значению $url еще не соответствует никакая запись в базе данных. В этом случае мы должны добавить в базу новую запись с этим Surl. Этот механизм позволяет добавлять новые URL, щелчки на которых требуется отслеживать, без необходимости обновления базы данных вручную. Если запрос завершается неудачно, мы переопределяем Squery как команду вставки в базу новой записи с этим URL и значением COUNT=1. Затем этот новый запрос готовится (prepare) и выполняется (execute).

```
27:  $dbh->disconnect;
28:  print redirect($url);
```

Строки 26 и 27 завершают скрипт, закрывая подключение к базе данных и перенаправляя броузер на нужный URL.

Теперь мы можем отслеживать отдельные щелчки на гиперссылках. Давайте посмотрим, как этот механизм работает. На рис. 8.1 показана простая Web-страница с шестью ссылками на URL для загрузки программ. Три из числа этих ссылок указывают на местные страницы, а остальные — на удаленные серверы.

В листинге 8.2 приведена часть кода HTML этой страницы, где помещаются гиперссылки.

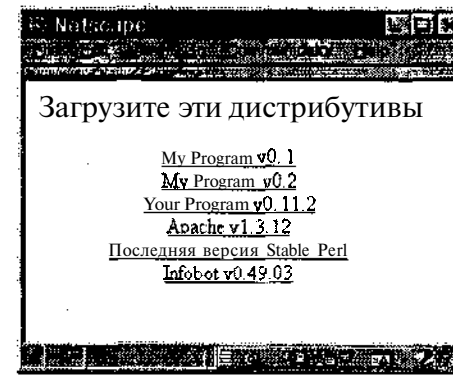



*Рис. 8.1. Экран Web-страницы*

**Листинг 8.2. HTML для рис. 8.1**

```
<A HREF="/cgi-bin/track.cgi?url=software/my_prog-0.1.tar.gz">
My Program v0.1</A>
<BR>
<A HREF="/cgi-bin/track.cgi?url=software/my_prog-0.2.tar.gz">
My Program v0.2</A>
<BR>
<A HREF="/cgi-bin/track.cgi?url=software/your_prog.tar.gz">
Your Program v0.11.2</A>
<BR>
<A HREF="/cgi-bin/track.cgi?url=http://www.apache.org/dist/
apache_1.3.12">Apache v1.3.12</A>
<BR>
<A HREF="/cgi-bin/track.cgi?url=http://www.perl.com/CPAN/src/
stable.tar.gz">Последняя версия Stable Perl</A>
<BR>
<A HREF="/cgi-bin/track.cgi?url=http://www.infobot.org/src/dev/
Infobot-0.49.03.tar.gz">Infobot v0.49.03</A>
```

Примерный результат отслеживания шелчков на этих ссылках показан в листинге 8.3.

**Листинг 8.3. Содержимое таблицы `click_tracker`**

```
mysql> select * from click_tracker
    -> \g
+-------------------------------------+-------+---------------------+
| PAGE                                | COUNT | LAST_ACCESS         |
+-------------------------------------+-------+---------------------+
| software/my_prog-0.1.tar.gz         |    16 | 2000-02-22 09:34:27 |
| software/my_prog-0.2.tar.gz         |    55 | 2000-02-22 09:34:27 |
| http://www.you.com/your_prog.tar.gz |    65 | 2000-02-22 09:49:49 |
| http://www.perl.com/CPAN/src/stable.tar.gz | 116 | 2000-02-23 09:08:08 |
| http://www.apache.org/dist/apache_1.3.12.tar.gz | 39 | 2000-02-24 10:34:39 |
| http://www.infobot.org/src/dev/Infobot-0.49.03.tar.gz | 97 | 2000-02-25 10:05:13 |
+-------------------------------------+-------+---------------------+
6 rows in set (0.00 sec)
```

# Пример: случайные изображения

Другое приложение, широко распространенное в Web — случайно меняющиеся изображения. Наиболее очевидная область применения приложений этого типа — рекламные баннеры. Следующий пример этой главы демонстрирует один из способов показа случайных изображений на Web-странице. Затем мы дополним этот пример и попробуем реализовать отслеживание количества показов (т.е. сколько раз было показано каждое изображение), количества щелчков на гиперссылке изображения и времени последнего показа и последнего щелчка. Эти функции образуют каркас приложения с меняющимися баннерами. Конечно, вы можете включить в это приложение и свои функции.

Сначала рассмотрим схему внутренней базы данных, которую мы будем использовать для этого приложения (листинг 8.4).

### Листинг 8.4. Схема базы данных для приложения с меняющимися баннерами

```
CREATE TABLE image_tracker (
  ID int(11) NOT NULL auto_increment,
  IMAGE varchar(255) NOT NULL,
  URL varchar(255) NOT NULL,
  ALT varchar(255) NOT NULL,
  CLICKS int(11),
  IMPRESSIONS int(11),
  LAST_IMPRESSION datetime,
  LAST_CLICK datetime,
  PRIMARY KEY (ID)
);
```

Эта команда определяет, как должна выглядеть таблица image_tracker. Первое поле ID — это автоинкрементное значение идентификатора, которое у нас будет играть роль уникального первичного ключа. Как используется это поле, будет показано ниже. Следующие Три поля, IMAGE, URL И ALT, будут содержать необходимую информацию для каждого изображения. В поле IMAGE будет храниться путь к изображению, в поле URL — URL, на который указывает гиперссылка, а в ALT — описание изображения, которое будет выводиться в атрибуте ALT дескриптора HTML <IMG>. Все эти поля не могут иметь пустые значения (NOT NULL). ЭТО вызвано тем, что данные этих полей необходимы для правильной работы приложения. Например, нельзя показать "пустое" изображение или создать ссылку на "пустой" URL. Хотя эта информация так или иначе будет помешена в базу данных, целесообразно принять дополнительные меры для сохранения целостности данных.

Следующие два поля, CLICK и IMPRESSIONS, будут использоваться для отслеживания количества щелчков на изображении и количества его показов на Web-странице. Если задача приложения — показывать баннеры, конечно, лучше знать оба этих параметра. "Щелчок" — это ситуация, когда броузер пользователя переходит по гиперссылке, связанной с баннером или изображением. "Показ" происходит каждый раз, когда изображение появляется на Web-странице. Если мы можем узнать, что щелчок на баннере А происходит при 10% его показов, а на баннере Б — при %75 его показов, это дает ценную информацию для маркетинга или просто говорит о том, что баннер А не нравится пользователям.

Далее, база данных содержит два поля даты-времени (datetime), в которых будет сохраняться время последнего показа изображения (LAST_IMPRESSION) И время последнего щелчка на нем (LASTCLICK). Скажем, вы управляете при помощи этого приложения баннерами своего клиента или партнера. Он может пожаловаться, что вы не показываете его баннеры и потому он не получает дохода с вашего сайта. Тогда будет полезна эта информация, так как поле LAST_IMPRESSION немедленно сообщит, отображается ли этот баннер.

В заключение на основе поля ID создается первичный ключ. Это необходимо, чтобы гарантировать, что каждая запись будет уникальной. Как мы увидим далее, это поле — самый существенный элемент приложений со случайным показом баннеров. Теперь, когда у нас есть готовая база данных, мы можем создавать скрипты для показа и отслеживания изображений. Мы начнем со скрипта, который выводит изображения случайным образом. Этот скрипт называется show_banner.cgi и используется как вставка на стороне сервера (SSI). Если в вашем приложении изображения не должны или не могут играть роль гиперссылок, этот скрипт можно изменить так, чтобы выводились только изображения, но не HTML. В результате скрипт можно было бы вызы-

вать из дескриптора HTML <IMG>. Но лучше, чтобы изображения имели ссылки, иначе какой прок в рекламе, если нет доступа к рекламируемому объекту? Скрипт вызывается следующей командой SSI.

```
<!--#include virtual="../cgi-bin/show_banner.cgi"-->
```

Конечно, путь надо будет изменить в соответствии с конфигурацией вашей системы.

```
01:  #!/usr/bin/perl -wT
02:  # show_banner.cgi
03:  use strict;
04:  use DBI;
05:  use CGI qw(:standard);
06:  use POSIX;
07:  use Image::Size qw(html_imgsize);
```

**Строки 1—7** задают путь к Perl и импортируют необходимые модули.

```
08:  my $dbname = 'book';
09:  my $dbhost = 'local host';
```

**Строки 8 и 9** инициализируют $dbname и $dbhost соответственно именем базы данных и ее местоположением.

```
10:  my $dsn = "DBI:mysql:database=$dbname;host=$dbhost";
11:  my $dbh=DBI->connect($dsn,"user","password");
12:  if (!defined($dbh)) {
13:    print header;
14:    print "\nошибка: Не удалось подключиться к базе
         данных MySQL:\n";
15:    print DBI->errmsg;
16:    print "-" x 25;
17:    exit;
18:  }
```

**В строках 10—18** производится попытка подключения к базе данных и при неудаче скрипт завершается с сообщением об ошибке.

```
19:  my $time = strftime("%Y-%m-%d %I:%M:%S", localtime);
```

**Строка 19** инициализирует переменную $time строкой даты и времени в стиле POSIX. Эти данные будут использоваться для обновления времени последнего показа изображения. Их окончательный формат — YYYY-MM-DD HH:MM:SS.

```
20:  my $query = qq(select ID from image_tracker);
```

**Строка 20** записывает в $query первый запрос. Первый элемент информации, который мы должны получить из базы данных — идентификаторы изображений в базе. Эти идентификаторы (значения поля ID) — единственный уникальный признак каждой записи, поэтому только их можно использовать для выбора изображений. Когда я разрабатывал этот скрипт, первой мыслью было получить максимальное значение идентификатора (функцией max(ID)) или общее количество записей (count(*)) и сгенерировать на его основе случайное число в диапазоне от 0 до самого этого значения (сейчас случайное число генерируется в **строке 27,** но другим способом). Недостаток этого метода в том, что, когда удаляется запись, поля ID остальных записей не обновляются автоматически и поэтому уже не образуют непрерывной последовательности чисел от 1 до максимального. Как мы увидим ниже, это создает опасность, что сгенерированному случайному числу может не найтись соответствия среди записей. Чтобы устранить эту проблему, я решил выбрать из базы все значения ID и поместить их в массив, из которого затем будет выбран элемент со случайным индексом.

```
21:  my $sth = $dbh->prepare($query);
22:  $sth->execute;
```

**Строки 21 и 22** подготавливают запрос и выполняют его. В переменной `$sth` сохраняется дескриптор команды.

```
23:  my @num;
```

**В строке 23** объявляется массив `@num`, в который будут помещены значения ID, возвращенные запросом.

```
24:  while (my $val = $sth->fetchrow_array) {
25:    push(@num, $val);
26:  }
```

**В строках 24—26** в массив `@num` помещаются данные, возвращенные запросом.

```
27:  my $rand = int(rand(@num));
```

**Строка 27** генерирует случайное число и присваивает это значение переменной $rand. Случайное число генерируется в диапазоне от 0 до количества элементов в `@num`. Массив `@num`, используемый как скаляр, возвращает количество элементов в нем. Если обратиться к нему в форме $#num, будет получен максимальный индекс элемента в массиве, причем счет начинается с 0. Более выразительным надо считать обращение к массиву как к скаляру, так как при втором способе, чтобы получить количество элементов, надо еще прибавить к $#num единицу.

```
28:  Squery = "select IMAGE, URL, ALT from image_tracker
       where ID=" . $num[$rand];
29:  $sth = $dbh->prepare($query);
30:  $sth->execute;
```

**В строках 28—30** обрабатывается запрос, который возвращает из базы информацию об изображении, которое надо показать. В определении запроса в **строке 27** извлекаются поля IMAGE, URL и ALT из записи, поле ID которой равно значению элемента массива со случайным **индексом**, $num[$rand]. Затем этот запрос готовится и выполняется.

```
31:  my ($image, $url, Salt) = $sth->fetchrow_array;
```

**В строке 31** возвращенные значения извлекаются в контексте массива и присваиваются переменным $image, $url и $alt.

```
32:  $query = qq(update image_tracker set
       IMPRESSIONS=IMPRESSIONS+1,
       LAST_IMPRESSION='$time' where IMAGE='$image');
33:  $sth = $dbh->prepare($query);
34:  $sth->execute;
35:  $dbh->disconnect;
```

**В строках 32—35** определяется и выполняется последний запрос. Этот запрос должен увеличить значение поля IMPRESSIONS на 1 и обновить поле `LAST_IMPRESSION` значением текущей даты и времени, которое было сохранено в `$time` в **строке** 19. Наконец, **строка 35** закрывает подключение к базе данных.

```
36:  my Ssize = html_imgsize($image);
37:   print qq(<A HREF="track_banner.cgi?url=$url&img=$image">
     <IMG SRC="$image" $size ALT="$alt"x/A>) ;
```

**Строки 36 и 37** завершают скрипт. **Строка 36** методом *`html_imgsi()`* модуля `Image::`Size определяет высоту и ширину изображения Simage. Этот метод возвращает строку, пригодную для помещения в дескриптор HTML <IMG>, которая сохраняется в Ssize. **Строка 37** выполняет последнюю операцию, выводя изображение в бро-

узер. Можно заметить, что созданная гиперссылка указывает на track_banner.cgi — следующий скрипт, который мы рассмотрим в этой главе. Эта гиперссылка также передает в track_banner.cgi два параметра: url — URL, на который будет направлен броузер при щелчке, и img — показываемое изображение. На основе этих значений скрипт track_banner. cgi обновляет запись для данного изображения.

Предыдущий скрипт может быть очень полезен и сам по себе. В нем демонстрируется хранение информации изображения в простой таблице базы данных и отображение этой информации. Также здесь представлен хороший метод получения из базы случайной записи. Но в следующем разделе мы увидим, как дополнить этот скрипт другим скриптом. Вместе они образуют более законченное и работоспособное приложение.

# Пример: отслеживание щелчков (повторение)

Следующий пример почти точно повторяет первый скрипт этой главы. Он принимает данные, переданные из show_banner.cgi, обновляет базу данных и перенаправляет броузер на URL, который хотел получить пользователь.

```
01:  #!/usr/bin/perl -wT
02:  # track_banner.cgi
03:  use strict;
04:  use DBI;
05:  use CGI qw(:standard);
06:  use POSIX;
07:  my $dbname = 'book';
08:  my $dbhost = 'localhost';
09:  my $dsn = "DBI:mysql:database=$dbname;host=$dbhost";
10:  my $dbh=DBI->connect($dsn,"user","password");
11:  #if (!defined($dbh)} {
12:    print header;
13:    print "\nошибка: Не удалось подключиться к базе
         данных MySQL:\n";
14:    print DBI->errmsg;
15:    print "-" x 25;
16:    exit;
17:  }
18:  my $url = param('url');
19:  my $image = param('img');
20:  my $time = strftime("%Y-%m-%d %I:%M:%S", localtime);
```

**Строки 1–20** выполняют действия, о которых мы уже говорили выше.

```
21:  my $query = qq(update image_tracker set
       CLICKS=CLICKS+1, LAST_CLICK='$time'
       where IMAGE='$image' and URL='$url');
```

**Строка 21** определяет команду SQL, которая обновляет соответствующую запись, увеличивая значение поля COUNT на 1 и записывая в поле LAST_CLICK новую дату и время.

```
22:  my $sth = $dbh->prepare($query);
23:  $sth->execute;
24:  $dbh->disconnect;
25:  print redirect($url);
```

**Строки 22–25** готовят запрос, определенный в **строке 21,** выполняют этот запрос, производят отключение от базы данных и, наконец, перенаправляют броузер на нужный URL.

Сочетание двух этих скриптов дает нам Web-приложение, которое отображает периодически меняющиеся баннеры, а также отслеживает количество их показов и щелчков. Его можно использовать как основу для дополнительных функциональных возможностей, которые могут вам понадобиться.

# Упражнения

- В этом приложении отсутствует одна важная функциональная возможность — добавление пользователем новых изображений и фиксация информации о них в базе данных. Создайте Web-страницу, позволяющую записывать в базу данных всю необходимую информацию о новых изображениях. Эта страница должна давать возможность как загружать на сервер файлы новых изображений, так и добавлять изображения, которые уже есть на сервере. Пользователь также должен иметь возможность добавить URL для гиперссылки и описание для атрибута ALT.
- Закончив это упражнение, добавьте в него функцию редактирования данных об изображениях. Это будет полезно для внесения изменений в URL, а также для исправления опечаток, которые могут возникнуть при вводе записи.
- Сделайте так, чтобы при ошибке подключения к базе данных пользователь все равно направлялся на требуемый URL.
- Наконец, добавьте функцию удаления записи из базы данных.

# Листинги

Листинг 8,5. Полный текст **track.cgi**

```
01: #!/usr/bin/perl -wT
02: # track.cgi
03: use strict;
04: use DBI;
05: use CCI qw(:standard);
06: use POSIX;
07: my $dbname = 'book';
08: my $dbhost = 'localhost';
09: my $dsn = "DBI:mysql:database=$dbname;host=$dbhost";
10: my $dbh=DBI->connect($dsn,"user","password");
11: if (!defined($dbh)) {
12:   print header;
13: print "\nошибка: Не удалось подключиться к базе
     данных MySQL:\n";
14: print DBI->errmsg;
15: print "-" x 25;
16: exit;
17: }
18: my $url = param('url');
19: ray $time = strftime("%Y-%m-%d %I:%M:%S", localtime);
20: my $query = qq(update click_tracker set
     COUNT=COUNT+1, LAST_ACCESS='$time' where PAGE='$url');
21: my $sth = $dbh->prepare($query);
22: if ($sth->execute < 1) {
23:   $query = qq(insert into click_tracker (PAGE, COUNT,
       LAST_ACCESS) Values ('$url', 1, '$time'));
24:   $sth = $dbh->prepare($query);
```

```
25:     $sth->execute;
26:   }
27:   $dbh->disconnect;
28:   print redirect($url);
```

**ЛИСТИНГ 8.6. Полный текст show_banner.cgi**

```
01:  #!/usr/bin/perl -wT
02:  # show_banner.cgi
03:  use strict;
04:  use DBI;
05:  use CGI qw(:standard);
06:  use POSIX;
07:  use Image::Size qw(html_imgsize);
08:  my $dbname = 'book';
09:  my $dbhost = 'local host';
10:  my $dsn = "DBI:mysql:database=$dbname;host=$dbhost";
11:  my $dbh=DBI->connect($dsn,"user","password");
12:  if (!defined($dbh)) {
13:    print header;
14:    print "\пошибка: Не удалось подключиться к базе
         данных MySQL:\n";
15:    print DBI->errmsg;
16:    print "-" x 25;
17:    exit;
18:  }
19:  my $time = strftime("%Y-%m-%d %I:%M:%S", localtime);
20:  my Squery = qq(select ID from image_tracker);
21:  my $sth = $dbh->prepare($query);
22:  $sth->execute;
23:  my @num;
24:  while (my $val = $sth->fetchrow_array) {
25:    push(@num, $val);
26:  }
27:  my $rand = int(rand(@num));
28:  Squery = "select IMAGE, URL, ALT from image_tracker
       where ID=" . $num[$rand];
29:  Ssth = $dbh->prepare($query);
30:  $sth->execute;
31:  my ($image, Surl, Salt) = $sth->fetchrow_array;
32:  Squery = qq(update image_tracker set
       IMPRESSIONS=IMPRESSIONS+1,
       LAST_IMPRESSION='$time' where IMAGE='$image');
33:  Ssth = $dbh->prepare($query);
34:  Ssth->execute;
35:  $dbh->disconnect;
36:  my Ssize = html_imgsize($image);
37:  print qq(<A HREF="track_banner.cgi?url=$url&img=$image">
     <IMG SRC="$image" Ssize ALT="$alt"></A>);
```

**Листинг 8.7. Полный текст track_banner.cgi**

```
01:  #!/usr/bin/perl -wT
02:  # track_banner.cgi
03:  use strict;
04:  use DBI;
```

```
05:  use CGI qw(:standard);
06:  use POSIX;
07:  my $dbname = 'book';
08:  my $dbhost = 'localhost';
09:  my $dsn = "DBI:mysql:database=$dbname;host=$dbhost";
10: my $dbh=DBI->connect ($dsn, "user", "password") ;
11:  #if (!defined($dbh)) {
12:    print header;
13:    print "\nошибка: Не удалось подключиться к базе
        данных MySQL:\n";
14:    print DBI->errmsg;
15:    print "-" x 25;
16:    exit;
17:  }
18:  my $url = param('url');
19:  my $image = param('img');
20:  my $time = strftime("%Y-%m-%d %I:%M:%S", localtime);
21:  my $query = qq(update image_tracker set
      CLICKS=CLICKS+1, LAST_CLICK='$time'
      where IMAGE='$image' and URL='$url');
22:  my $sth = $dbh->prepare($query);
23:  $sth->execute;
24:  $dbh->disconnect;
25:  print redirect($url);
```


# Использование mod_perl

## Что такое mod_perl

mod_perl — это модуль Apache, который позволяет Web-серверу непосредственно выполнять код Perl. Иными словами, в Apache фактически встраивается интерпретатор Perl, так что скрипты на Perl можно выполнять без необходимости запускать внешний процесс (интерпретатор Perl). Обычно, когда Web-сервер должен выполнить скрипт CGI, он запускает скрипт в оболочке (shell) и принимает его вывод. Это ограничивает скорость выполнения скриптов, так как они по сути воспроизводятся в самой системе. mod_perl решает эту проблему, избегая необходимости каждый раз начинать новый процесс интерпретатора Perl при выполнении CGI. Это осуществляется за счет компоновки библиотеки выполнения Perl с сервером Apache, что позволяет не только повысить скорость CGI, но и предоставляет разработчикам объектно-ориентированный способ взаимодействия с API серверов Apache. Возможность работать с API Apache дает разработчику доступ почти ко всем фазам запроса HTTP.

Самое большое преимущество mod_perl — скорость. Так как при обработке кода CGI не возникает дополнительной задержки, связанной с запуском процесса, результаты возвращаются более оперативно. Как мы говорили выше, разработчик может вмешаться в любую фазу запроса HTTP: регистрацию, идентификацию, подготовку ответа и передачу URI. Все эти возможности открывают перед программистом целый новый "континент" CGI.

Конечно, использование mod_perl имеет и свои неприятные стороны, самая важная из которых — увеличение размера процесса HTTPD. Это происходит потому, что в HTTPD включается интерпретатор Perl. Это проявление компромисса между скоростью и размером процесса. Существует несколько способов устранить эту проблему, один из которых — простая модернизация ОЗУ вашей системы. Если это неприемлемо, Web-сервер можно сконфигурировать так, чтобы неактивных дочерних процессов было возможно меньше, либо ограничить число клиентов, которые одновременно могут подклю-

чаться к серверу. Документация к Web-серверу — лучший способ узнать самые новые и действенные методы решения этой задачи. Другой путь уменьшения размера процесса — написать стартовый скрипт для предварительной загрузки наиболее часто используемых модулей Perl. Таким образом, эти модули будут компилироваться только один раз, при запуске сервера, а не при каждом появлении в последующих скриптах оператора *use ()*. Пример стартового скрипта будет показан в следующем разделе.

Настоятельно рекомендуется прочесть документацию к mod_perl — как ту, что поставляется с дистрибутивом, так и документацию и страницы FAQ, содержащиеся на Web-сайте mod_perl по адресу http://perl.apache.org. В документации можно найти различные способы конфигурации и применения mod_perl, а также решить на ее основании, действительно ли mod_perl — то, что вам нужно.

Некоторые из модулей, используемых в этой главе, не входят в состав mod_perl, но могут быть найдены в CPAN. Модули устанавливаются так же, как любые другие модули Perl, так как они ничего другого собой не представляют. Важно, чтобы вы прочли документацию по модулю и научились конфигурировать его и применять.

Ниже перечислены прагмы и модули, используемые в этой главе.

- strict
- lib
- Apache.pm
- Apache :Constants
- Apache::Registry
- Apache::Album
- Apache::AuthDBI
- Apache::MyLog
- Apache::Sandwich
- CGI.pm
- Untaint.pm
- Image::Magick

# Конфигурация mod_perl

Существует несколько способов конфигурации mod_perl, о которых можно узнать в документации. Базовая конфигурация в основном проста и естественна. В листинге 9.1 показан примерный раздел конфигурации, который можно вставить в httpd.conf.

Листинг 9.1. Пример базовой конфигурации mod_perl

```
PerlTaintCheck On
PerlRequire /usr/local/apache/perllib/startup.perl
<Directory/usr/local/apache/htdocs/perl-cgi>
  SetHandler perl-script
  PerlHandler Apache::Registry
  PerlSendHeader On
  AllowOverride None
  Options +ExecCGI
</Directory>
```

Пока пропустим строки PerlRequire и PerlTaintCheck. В остальной части этого примера демонстрируется разрешение использовать каталог для mod_perl. Данные конфигурации содержатся в директиве Directory; название каталога —

/usr/local/apache/htdocs/perl-cgi. Директива SetHandler установлена на значение perl-script, что означает, что Apache должен привлекать raodperl для выполнения запросов к /usr/local/apache/htdocs/perl-cgi. Директива PerlHandler сообщает, что mod_perl должен использовать модуль Apache: :Registry для обработки запроса, прежде чем ответ будет направлен в броузер. Сейчас мы не будем останавливаться на работе модуля, указанного в PerlHandler. Достаточно запомнить, что этот модуль (или модули) обслуживает фазу запроса HTTP.

В листинге 9.1 также упоминается Apache: :Registry. Этот модуль помогает выполнять скрипты CGI Perl под mod_perl, не внося в них изменений (надо надеяться). Как указано в листинге 9.1, при каждом запросе к http://www.you.com/perl-cgi/script.cgi скрипт будет скомпилирован *по одному разу для каждого* дочернего процесса HTTPD и кэширован модулем Apache: :Registry. Это может значительно ускорить работу CGI, потому что для каждого запроса к скрипту, который приходит после того, как этот скрипт был скомпилирован и кэширован, результаты выдаются из кэша, а компилирование заново не требуется. Код перекомпилируется, только если скрипт изменился. Более подробно мы рассмотрим модуль Apache : : Registry ниже.

Директива PerlSendHeader указывает серверу Apache, что он должен самостоятельно выводить заголовки HTTP. Вообще, это хорошая идея. Когда эта директива включена, Web-сервер, на котором работает mod_perl, проверяет, передают ли скрипты заголовок HTTP. Если это так, он сам выполняет это действие.

Теперь перейдем к директиве PerlTaintCheck. Когда Apache: :Registry просматривает строку #!, он распознает только переключатель -w. Если в этой строке включены предупреждения, переменной $^W модуля присваивается значение 1. Но не существует подобной специальной переменной Perl, в которой бы фиксировался режим проверки на загрязнение. Поэтому, чтобы включить проверку на загрязнение (что следует сделать), разработчик должен применить директиву PerlTaintCheck. Когда она включена (On), mod_perl узнает, что CGI надо выполнять с проверкой на загрязнение, и предпринимает соответствующие действия. Если в строке #! задан переключатель -т, но директива PerlTaintCheck не включена, в журнале Web-сервера появится ошибка, но скрипт будет выполнен, хотя и без проверки на загрязнение.

Нам осталось рассмотреть директиву PerlRequire. Когда Web-сервер с mod_perl запускается, он загружает стартовый файл, если такой указан в директиве PerlRequire. Фактически при этом стартовый файл вызывается командой *require ()*, и компилируются те модули, которые импортированы в нем командой use *()*. После этого любой скрипт CGI, в котором используется mod_perl, может вызвать эти модули командой *use (),* и они не потребуют нового компилирования. Листинг 9.2 — пример стартового файла startup.perl.

### Листинг 9.2. Пример стартового скрипта

```
#!/usr/bin/perl -w
use strict;
use lib "/usr/local/apache/perllib";
use Apache::Registry ();
use CGI ();
CGI->compile(':all');
use CCI::Carp ();
use DBI ();
use DBD::mysql ();
use Untaint ();
1;
```

Этот скрипт вызывает все модули, которые в нем указаны, и помешает скомпилированный код этих модулей в процесс HTTPD. Он не импортирует из этих модулей конкретные методы, поэтому разработчик не избавляется от необходимости указывать модули после команды use *()* в своих скриптах. Однако Perl не будет перекомпилировать код этих модулей, так как это уже сделано. Скомпилированный код появится и в каждом дочернем процессе HTTPD.

Еще одно полезное действие, как можно заметить в этом скрипте, выполняет строка use lib *'/путь'*. Ее присутствие означает, что нам не нужно указывать use lib *'/путь'* в каждом файле, как мы делали для `Untaint.pm`. В результате для вызова дополнительных модулей из другого каталога надо только указать их в *use()*. Это может быть особенно полезно при разработке новых модулей, так как в этот каталог можно помешать их новые редакции для проверки.

# Apache::Registry

Выше в этой главе мы кратко коснулись модуля Apache: `:Registry`, но полезнее будет поговорить о нем более подробно. Этот модуль входит в состав дистрибутива **mod** perl и чрезвычайно полезен при разработке CGI. Одно из достоинств этого модуля — то, что он моделирует окружение CGI. Иными словами, при компилировании исходного кода этот модуль создает окружение, подобное реальному окружению CGI, в котором будет выполняться скрипт. Apache: `:Registry` полезен еще и тем, что кэширует скомпилированный код, благодаря чему его не надо перекомпилировать при каждом вызове скрипта. Этот модуль просто сравнивает время кэширования и время последнего изменения файла. Если скрипт изменился с момента последнего кэширования, он компилируется и кэшируется заново. Кроме того, Apache: `:Registry` позволяет программисту выбирать тип создаваемого скрипта. Скрипты CGI можно писать обычным способом или с использованием Apache Perl API. Можно применять и оба способа.

Разработчик сам определяет, какой стиль наиболее подходит для него. Если приложение интенсивно использует Apache Perl API, это может уменьшить его мобильность, так как многое придется изменять при переносе скрипта на Web-сервер без mod_perl. Если такое перемещение маловероятно, это дает разработчикам большие преимущества, так как с помощью API можно реализовать многое, что недоступно в обычном CGI, например, регистрацию и управление запросом до отправки ответа. Для примера в листинге 9.3 показан скрипт, написанный как обычный CGI, а в листинге 9.4 — тот же скрипт, но с использованием Apache Perl API.

**Листинг 9.3. Пример использования `Apache::Registry` в скрипте CGI**

```
#!/usr/bin/perl -wT
use strict;
print "Content-type: text/html\n\n";
my $request_time = scalar localtime;
print<<HTML;
  <HTML>
  <HEAD>
  <TITLE>Пример Apache::Registry</TITLE>
  </HEAD>
  <BODY>
    Здравствуйте, $ENV{"REMOTE_ADDR"}. В $request_time
    вы просили, чтобы я послал вам $ENV{"REQUEST_URI"}
    по протоколу $ENV{"SERVER_PROTOCOL"}. Вот он.
    Спасибо за развлечение.
  </BODY>
  </HTML>
HTML
```

Как можно видеть, этот скрипт похож на любой другой CGI. Единственное отличие — то, что он управляется модулем Apache: :Registry (считая, что это разрешено в конфигурации вашего сервера Apache).

### Листинг 9.4. Пример использования Apache Perl API в скрипте

```
01:  #!/usr/bin/perl -wT
02:  use strict;
03:  use Apache::Constants qw(OK);
04:  my $r = Apache->request;
05:  $r->content_type('text/html');
06:  $r->send_http_header;
07:  return OK if $r->header_only;
08:  my $remote_host = $r->get_remote_host;
09:  my $request_time = scalar localtime $r->request_time;
10:  my $uri = $r->uri;
11:  my $protocol = $r->protocol;
12:  $r->print(<<HTML);
13:    <HTML>
14:    <HEAD>
15:    <TITLE>Пример Apache::Registry</TITLE>
16:    </HEAD>
17:    <BODY>
18:    Здравствуйте, $remote_host. В $request_time
       вы просили, чтобы я послал вам $uri
19:    по протоколу $protocol. Вот он.
       Спасибо за развлечение.
20:    </BODY>
21:    </HTML>
22:  HTML
```

В этом скрипте вводятся некоторые новые концепции, так что его стоит рассмотреть построчно.

**Строки** 1 и 2 сообщают, где находится интерпретатор Perl, и включают предупреждения компилятора и прагму `strict`.

В **строке** 3 вызывается модуль Apache: `:Constants` и из него импортируется константа OK. Модуль `Apache::Constants` содержит константы, используемые в файлах заголовка Apache, что позволяет программисту легко возвращать коды ответа сервера, такие как OK, DECLINED, FORBIDDEN и т.д. Для этого скрипта нужна только константа OK.

**Строка 4** присваивает переменной `$r` ссылку на объект запроса Apache. Модуль `Apache.pm` содержит объект, в котором хранится вся информация, нужная серверу для обработки запроса. Метод `request()` возвращает ссылку на этот объект; в данном случае она сохраняется в *$r*. Такое применение *$r* обычно во многих программах Apache Perl API. Использование буквы `r` для ссылки на объект — хороший способ запомнить самому и сообщить другим, что она обозначает объект *запроса* (request). Аналогично, в некоторых скриптах Ss представляет объект *сервера,* но пока мы не будем углубляться в этот вопрос. Имея ссылку на объект запроса, разработчик может обращаться ко всей информации, которую этот объект содержит.

**Строка 5** вызывает метод *content_type ()* объекта запроса и задает тип содержимого документа.

**Строка 6** посылает в броузер заголовок ответа при помощи метода *send_http_header()*. Можно заметить одну особенность Apache Perl API: названия методов в основном отражают их действие.

**Строка 7** возвращает в броузер ответ сервера ок (200), если броузеру нужен только заголовок, без содержимого. В этом случае после отправки ответа скрипт завершается.

В **строке** 8 методом *get_remote_host()* определяется имя DNS машины, на которой находится броузер. Если имя машины неизвестно, возвращается undef. Если в конфигурации Apache не включена директива HostNameLookups, возвращается IP-адрес клиента. Если же эта директива включена, возвращается доменное имя клиента. Часто эту директиву не включают (по умолчанию она также не включена), так как это может привести к снижению производительности, когда сервер будет пытаться выполнить обратный просмотр DNS для всех запрашивающих клиентов. В любом случае полученное значение сохраняется в $remote_host.

**Строка 9** записывает в $request_time скалярное представление времени, когда был сделан запрос на сервер.

**Строка 10** методом *uri ()* извлекает URI из объекта запроса. Полученное значение сохраняется в $uri.

**Строка 11** вызывает метод *protocol ()*, который определяет протокол HTTP, примененный для запроса этого документа. Значение сохраняется в $protocol.

**Строки 12—22** при помощи метода *print ()* объекта запроса Apache *request ()* передают текст в броузер. Этот метод заменяет встроенную функцию Perl *print ()*. Метод Apache: :print устанавливает время ожидания, прежде чем данные фактически будут отправлены клиенту. Встроенная функция *print ()* этого не делает.

Рекомендуется сначала ознакомиться с методами Apache.pm, внимательно прочитав документацию этого модуля. Работая с mod_perl, вы, конечно, должны будете знать все, на что он способен. Это особенно верно, если вам нужно написать собственный обработчик содержимого, что мы и сделаем позже в этой главе.

Следует запомнить, что Apache: :Registry должен применяться только для исполняемых файлов. Вообще, если файл — не скрипт Perl, не следует, чтобы Apache::Registry работал с ним. Это вызвано тем, что данный модуль считает все запрашиваемые файлы подпрограммами Perl, т.е., произойдет ошибка, если он попытается скомпилировать, например, файл HTML. Этого избежать нетрудно: надо только сконфигурировать Apache так, чтобы Apache: :Registry был обработчиком только для файлов с определенным расширением или в определенном каталоге скриптов CGI. В листинге 9.5 показан такой раздел конфигурации, который разрешает обработку в Apache::Registry для всех файлов в дереве Web, имеющих расширение .rpl. Конечно, для этого файлы .rpl должны быть выполняемыми скриптами Perl. В листинге 9.1 обработка в Apache: : Registry была разрешена для всех файлов в каталоге.

**ЛИСТИНГ 9.5. Пример раздела конфигурации ДЛЯ Apache::Registry**

```
<Files *.rpl$>
  SetHandler perl-script
  PerlHandler Apache::Registry
  PerlSendHeader On
  Options +ExecCGI
</Files>
```

В этом разделе вы узнали, что делает Apache: :Registry и как использовать этот модуль. Также вы научились работать с некоторыми методами Apache Perl API, что позволяет ускорить выполнение кода. В следующих разделах мы расскажем о других модулях и областях применения mod_perl.

# Автоматические колонтитулы с использованием Apache: : Sandwich

Большинство профессиональных Web-сайтов, если не все, имеют цельный, согласованный дизайн, нацеленный на то, чтобы доставить удовольствие пользователям и создать впечатление фирменного стиля. Согласованность дизайна страниц достигается цветовыми схемами, сходством графики, а также верхними и нижними колонтитулами. Как добиться этой согласованности и сохранить ее — важная проблема при разработке Web-сайта. Должна быть продумана легкая в обращении система добавления и изменения таких элементов страницы, как колонтитулы. Ниже мы попробуем реализовать такую систему с использованием mod_perl и Apache: : Sandwich.

Верхним колонтитулом мы называем верхнюю часть Web-страницы, которая может содержать эмблему компании, информацию раздела <BODY>, баннеры и т.д. Нижний колонтитул — это нижняя часть страницы, которая может содержать текстовые ссылки и сведения об авторском праве. Лучший способ легко добавить эту информацию на каждую Web-страницу заключается в том, чтобы просто поместить соответствующий текст на каждой странице сайта. Это может быть довольно легким делом, если имеется шаблон, по которому создаются все страницы. Действительно, такой шаблон позволяет быстро сгенерировать сотни страниц, но что делать, если требуется добавить в нижний колонтитул новую ссылку? Возможные варианты — исправить каждую страницу вручную, написать скрипт, который будет изменять каждую страницу, или применить приложение, которое позволяет делать групповой поиск и замену. Все эти варианты требуют долгого времени и не очень дружественны к разработчику. Мало того, что этот процесс утомителен, очень вероятно, что после него придется самостоятельно просмотреть многие (если не все) страницы, чтобы убедиться, что скрипт или приложение не сделали ошибок. Не знаю, как вы, но я считаю такой вариант нереальным.

Другой способ вставки колонтитулов на каждую страницу состоит в использовании SSI. SSI можно поместить в начале и в конце каждой Web-страницы. Это решение напоминает предыдущее, однако теперь нас будут заботить лишь несколько строк текста. Но что, если наш SSI отображает эмблему компании, а мы решили вставить под ним баннер? Снова придется изменять каждую страницу, на которой он должен быть отображен. Это связано с теми же проблемами, что и формирование колонтитулов в коде HTML каждой страницы. Возможно, этот вариант лучше, но он также нежизнеспособен.

Самое надежное решение — использовать mod_perl и Apache: : Sandwich. Модуль Apache: : Sandwich, легкий в использовании, вставляет верхний и/или нижний колонтитул на все Web-страницы автоматически. Он способен на это, так как выполняет функцию обработчика содержимого для Apache и может изменить содержимое файла до того, как он будет отправлен в броузер в фазе ответа HTTP. Это одна из наиболее интересных возможностей mod_perl. Он позволяет модулю Perl "вклиниваться" в запрос между клиентом и Web-сервером. Иными словами, Apache: : Sandwich действует как посредник и создает "сандвич" из содержимого HTML и заданных верхнего и нижнего колонтитулов.

Посмотрим, как реализовать этот способ. Во-первых, в файле конфигурации httpd.conf должен быть раздел, указывающий Web-серверу, что надо использовать Apache: : Sandwich как обработчик запросов на файлы определенного типа. В нашем примере мы используем файлы .html, но можно указать и другие типы файлов и каталоги в зависимости от конкретных потребностей.

**Листинг 9.5. Пример раздела конфигурации для Apache::Sandwich**

```
<FilesMatch "\.html$">,
  SetHandler perl-script
  PerlHandler Apache::Sandwich
  PerlSetVar HEADER "/pbj/sandwich/header.txt"
  PerlSetVar FOOTER "/pbj/sandwich/footer.txt"
</FilesMatch>
```

Эти директивы заставляют Apache, получив запрос на файл с расширением .html, вызывать для обработки этого запроса Apache::Sandwich. Вначале значение perl-script директивы SetHandler сообщает серверу, что управление запросом надо передать в mod_perl. Затем директива PerlHandler сообщает mod_perl, что запрос должен обрабатывать Apache::Sandwich. Две строки PerlSetVar предназначены для модуля Apache::Sandwich. Они определяют две переменных, HEADER и FOOTER, значения которых — пути к соответствующим файлам. Это не полные пути на сервере, а относительные пути в корневом каталоге документов. Как можно заметить, содержимое верхнего и нижнего колонтитулов — просто текстовые файлы.

**Листинг 9.7. Содержимое header.txt**

```
<HTML>
<HEAD>
  <TITLE>Пример Apache::Sandwich</TITLE>
</HEAD>
<BODY>
<CENTER>
<H1>Привет! Я верхний колонтитул!</H1>
</CENTER>
```

**Листинг 9.8. Содержимое footer.txt**

```
<CENTER>
<H3>Пока! Я нижний колонтитул!</H3>
Copyright &copy; 19100, ЫЫЫ Инкорпорейтед Лтд Корпорейшен
</CENTER>
</BODY>
</HTML>
```

В листингах 9.7 и 9.8 приведено содержимое файлов header.txt и footer.txt. Как можно видеть, это простые файлы HTML. Если установлена описанная выше конфигурация и присутствуют эти файлы, каждый файл с расширением .html будет иметь колонтитулы с их содержимым. В листинге 9.9 показан файл index.html, который при запросе "вкладывается" между колонтитулами. Результат показан на рис. 9.1.

**Листинг 9.9. Содержимое index.html**

```
<HR NOSHADE>
<P>
<CENTER>
Вас приветствует основная часть страницы.
</CENTER>
<P>
<HR NOSHADE>
```

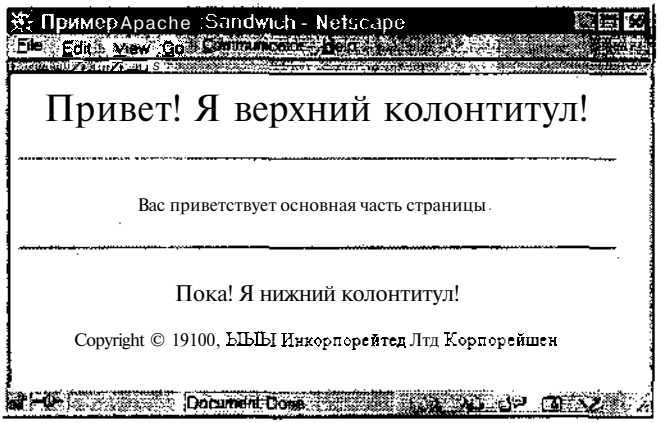



*Рис. 9.1. Наш "сандвич"*

Модуль Apache: :Sandwich также позволяет *собирать* колонтитулы. Это означает, что верхний или нижний колонтитулы могут состоять из последовательности файлов. Предположим, например, что мы хотим поместить на каждую страницу меняющийся баннер. Теперь это можно сделать, просто добавив в верхний колонтитул еще один файл. В данном случае можно применить скрипт с баннерами show_banner.cgi из главы 8. Чтобы добавить новый файл, конфигурацию придется изменить примерно так, как показано в листинге 9.10.

## Листинг 9.10. Два файла верхнего колонтитула, собранные вместе

```
<FilesMatch "\.html$">
  SetHandler perl-script
  PerlHandler Apache::Sandwich
  PerlSetVar HEADER "/pbj/sandwich/header.txt
  /cgi-bin/show_banner.cgi"
  PerlSetVar FOOTER "/pbj/sandwich/footer.txt"
</FilesMatch>
```

Если верхний или нижний колонтитул состоит из нескольких файлов, они отображаются или выполняются в порядке, в котором перечислены в конфигурации. Следовательно, листинг 9.10 даст примерно такой результат, как показано на рис. 9.2. В show_banner.cgi было внесено единственное изменение: теперь он выводит соответствующую строку.

Единственное замечание к Apache: :Sandwich — если попытаться использовать его со скриптами CGI, он не будет работать правильно. Причина в том, что в модуле, чтобы был отображен верхний колонтитул, должен быть выведен заголовок **HTTP** text/html, т.е., если телом страницы будет скрипт CGI, он просто отобразится в броузере, а не выполнится. По умолчанию модуль считает, что тело страницы — только статический текст. Однако начиная с версии 2.04 в Apache: : Sandwich появился вспомогательный метод *insert_parts* который можно применять для вставки колонтитулов. Единственное замечание — этот метод должен быть вызван в каждом скрипте. Но, хотя он работает не так хорошо, как для неисполняемых файлов, скрипты CG1 все же могут иметь возможность вставки колонтитулов, а также использования Apache: :Registry. В листинге 9.11 показан раздел конфигурации, позволяющий модулю Apache::Registry обрабатывать запросы **HTTP** на файлы с расширением .cgi. Здесь также определяются колонтитулы (HEADER и FOOTER) для Apache: : Sandwich. Присутствие в колонтитулах файлов с расширениями . cgi также может привести к странным результатам.

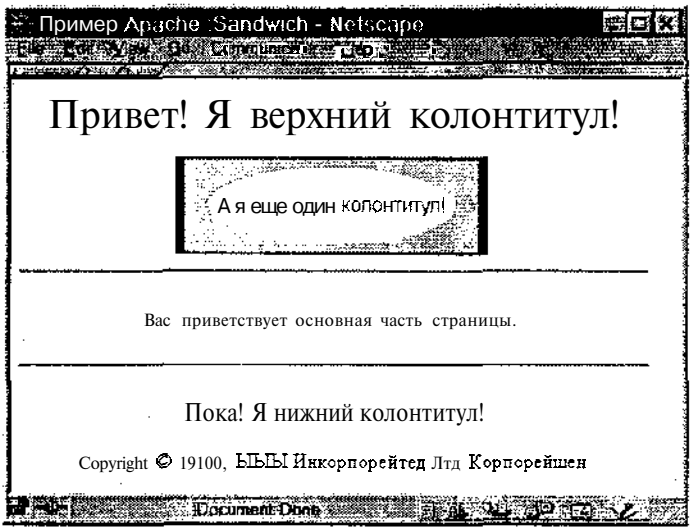



*Рис. 9.2. Двойной сандвич*

**ЛИСТИНГ 9.11.** Как вызвать **Apache::Registry** ДЛЯ файлов .cgi

```
<FilesMatch "*\.cgi">
  SetHandler perl-script
  PerlHandler Apache::Registry
  PerlSetVar HEADER "/book/c10/header.txt"
  PerlSetVar FOOTER "/book/c10/footer.txt"
</FilesMatch>
```

В листинге 912 показан пример использования метода *Apache::Sandwich::insert_parts()* в скрипте CGI.

**Листинг 9.12.** Пример скрипта CGI с методом **insert_parts()**

```
01:  #!/usr/bin/perl -wT
02:  use strict;
03:  use CGI qw(:standard);
04:  use Apache::Sandwich;
05:  print header;
06:  Apache::Sandwich::insert_parts('HEADER');
07:  print hr;
08:  print p({align=>'center'}, "Hello, world!");
09:  print hr;
10:  Apache::Sandwich::insert_parts('FOOTER');
```

**В строках 6 и 10** вызывается метод *insert_pa().s* Перед метолом указывается модуль, так как в операторе use Apache::Sandwich этот метод не импортируется непосредственно. Для применения этого метода скрипты должны загружать модуль Apache::Sandwich, а также выполняться в окружении mod_perl.

Теперь вы узнали, как можно быстро поместить колонтитулы на все Web-страниы и во все скрипты CGI на вашем сайте. Это позволяет разработчикам сэкономить много времени при создании сайта, а также сделать его более удобным в сопровождении.

# Фотоальбом с использованием Apache::Album

Многие крупные и мелкие компании создают для своих отделов маркетинга библиотеки изображений, которыми могут также пользоваться разработчики Web-сайта компании и маркетинговых материалов. Кроме того, многие люди выкладывают в Web картинки для своих родных и друзей. Одна из проблем, которые при этом возникают, — организация этих изображений таким образом, чтобы их было удобно просматривать. Мне приходилось видеть компании, у которых буквально десятки каталогов заполнены этими изображениями, но чтобы просмотреть их, приходится открывать каждое изображение отдельно или привлекать сторонние программы для просмотра эскизов. И то, и другое неудЬбно. Я слышал жалобы многих дизайнеров и разработчиков на то, что нет простого способа просмотра изображений.

В решении этой проблемы может помочь модуль Apache: : Album. Этот модуль позволяет разработчику создавать из каталогов с файлами изображений своего рода фотоальбомы. Он выполняет показ изображений, создание как эскизов (с помощью ImageMagick), так и изображений разной величины для различных размеров экрана, и содержит два метода обработки подписей к изображениям. Этот модуль по сути обладает функциями отдельного приложения, которое дает пользователю возможность создавать новые альбомы и загружать изображения через Web. Но хватит расписывать его возможности! Лучше посмотрим, как этот модуль реализован.

Модуль Apache: : Album имеет значительное количество опций конфигурации. Мы настоятельно рекомендуем: если вы собираетесь работать с этим модулем (как, впрочем, и со всеми другими модулями), прочтите документацию, чтобы узнать, какие новые возможности и параметры конфигурации будут в вашем распоряжении. В этом разделе мы рассмотрим базовую конфигурацию, которая позволяет показывать альбомы изображений, создавать и отображать эскизы и формировать эскизы для разных разрешений экрана. Самое интересное, что для этого не нужно никакого программирования. Надо только создать соответствующий раздел в конфигурации Web-сервера и структуру каталогов.

В первую очередь мы создадим структуру каталогов с альбомами изображений. Посмотрите на структуру, показанную в листинге 9.13.

Листинг 9.13. Пример структуры каталога для каталога Apache: :Album

```
/usr/local/apache/htdocs/albums
                              /kyla
                                   /albums_loc
                                              /vol1
                                              /vol2
```

Каталоги vol1 и vol2 — это фотоальбомы, которые непосредственно содержат файлы изображений. Эти каталоги должны разрешать доступ для записи, так как Apache: : Album создает в каждом каталоге фотоальбома отдельный подкаталог для эскизов. По умолчанию этот каталог называется thumbs, и мы не будем менять это название (если, конечно, ваш фотоальбом сам не посвящен исключительно большим пальцам (thumbs) — тогда для каталога эскизов придется подыскать другое название). В листинге 9.14 показана конфигурация, которую мы будем использовать.

**Листинг 9.14. Раздел httpd.conf для Apache: :Album**

```
01: Alias /kyla /usr/local/apache/htdocs/albums/kyla/
02: <Location /kyla/albums>
03:   SetHandler perl-script
```

```
04:    AllowOverride None
05:    Options None
06:    PerlHandler Apache::Album
07:    PerlSetVar AlbumDir /kyla/albums_loc
08:    PerlSetVar OutsideTableBorder 1
09:    PerlSetVar InsideTablesBorder 1
10:    PerlSetVar BodyArgs BGCOLOR=white
11:    PerlSetVar AllowFinalResize 1
12:    PerlSetVar FinalResizeDir thumbs
13: </Location>
```

**В строке 1** создается псевдоним для каталога /usr/local/apache/htdocs/albums/kyla. Теперь, если обратиться к http://www.me.com/kyla, мы увидим содержимое каталога albums/kyla/.

**В строке** 2 начинается раздел конфигурации для /kyla/albums. Этот путь указывается в директиве Location. В результате Web-сервер узнает, что, когда приходит запрос на http://www.me.com/kyla/albums, надо обратиться к данным этой директивы, и действует соответственно.

**Строка** 3 указывает серверу использовать для управления этим запросом (в данном случае всем, что запрашивается в каталоге /kyla/albums) модуль mod_perl.

**Строка 4** устанавливает, что никакие подкаталоги не могут отменять параметры, заданные в этом разделе конфигурации. Мы полагаемся на Apache::Album, который сам управляет всем, что нам понадобится сделать в этом дереве каталогов.

В **строке** 5 устанавливается параметр Options None, что означает, что сервер запрещает выполнение из этого местоположения CGI, индексации каталогов и т.д. Необходимости в опциях действительно нет, так как это дерево каталогов будет содержать только файлы изображений и необязательный текстовый файл, о чем мы расскажем позже.

**Строка 6** указывает mod_perl использовать модуль Apache::Album как обработчик для поступающих запросов.

В **строке** 7 переменной AlbumDir присваивается физическое местоположение (от корня документов Web-сервера) каталога, содержащего подкаталоги альбома. Это нужно модулю Apache::Album, чтобы знать, где находится физический каталог, содержащий фотоальбомы. PerlSetVar входит в состав Apache Perl API, который позволяет программисту легко передавать переменные из файла конфигурации Web-сервера в скрипт CGI. Более наглядно мы покажем это позже, когда будем изучать создание обработчика mod_perl.

**Строка 8** также устанавливает переменную для Apache::Album. Эта установка означает, что внешняя таблица HTML в окончательном отображении в броузере будет иметь границу.

**Строка 9** аналогична **строке** 8, только она задает границу для внутренних таблиц HTML. Результат можно увидеть на рис. 9.4.

**Строка 10** задает для альбома белый цвет фона.

В **строке** 11 устанавливается переменная, которая сообщает модулю Apache::Album, что должно быть разрешено создавать изображения разного размера. Обычно генерируются только эскизы, но эта опция позволяет генерировать изображения для разных разрешений экрана. Если необходимо, могут быть созданы изображения для разрешения 640x480, 800x600 или 1024x758. Например, если ширина изображения составляет 900 точек, Apache::Album создаст как эскиз, так и изображения, пригодные для показа на экранах размером 640x480 и 800x600. В готовой странице щелчком на эскизе можно будет выбрать показ и полноразмерного изображения, и всех изображений для разных размеров экрана.

**В строке 12** устанавливается переменная, которая указывает модулю, в каком каталоге должны быть созданы изображения альтернативных размеров. В данном случае задан каталог thumbs, т.е. они будут находиться в том же каталоге, что и эскизы.

**Строка 13** завершает этот раздел конфигурации.

Вот и все! Теперь, если пользователь направит свой броузер по адресу `http://www.me.com/kyla/albums`, он увидит список имеющихся фотоальбомов, как на рис. 9.3. Если он выберет один из этих альбомов, будет показана таблица с эскизами изображений этого каталога, как на рис. 9.4. И, наконец, если пользователь щелкнет на одном из этих эскизов, будет показана полноразмерная версия изображения, как на рис. 9.5.

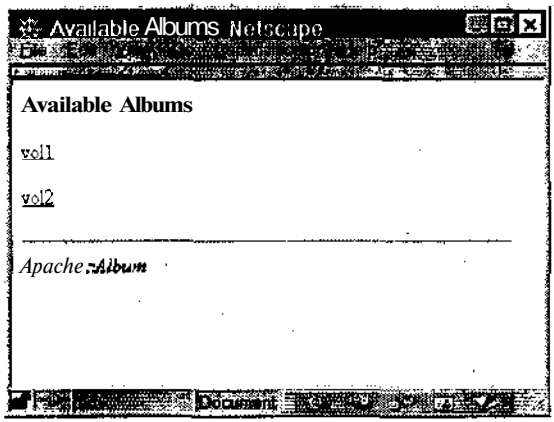



*Рис. 9.3. Список фотоальбомов*


*Рис. 9.4. Просмотр эскизов*

Пока все идет хорошо. Вы теперь знаете, как сделать из каталогов с файлами изображений фотоальбомы с эскизами и изображениями для разных размеров экрана. Возможно, вы также заметили, что подпись к каждой фотографии — слегка измененная версия имени ее файла. Это очень полезная особенность Apache: `:Album`. Однако файлы могут иметь не такие наглядные имена, так что может оказаться более благоразумным самостоятельно составить подписи к некоторым или ко всем фотографиям. Также не помешает дополнить каждую страницу фотоальбома каким-то комментари-

ем, к примеру о том, где были сделаны фотофафии наподобие "Мой день рождения", или об их жанре, например, "Писходелические образы". Apache: `:Album` позволяет сделать это очень простым способом— создав текстовый файл `caption.txt`. Если поместить этот файл в каталог с соответствующими фотографиями, в результате мы получим специально созданные подписи. В листинге 9.15 показан файл `caption.txt`, который можно поместить в каталог `vol2`.


*Рис. 9.5. Просмотр полноразмерного изображения*

Листинг 9.15. Файл **`caption.t x t`** файла для каталога **`vol2`**

```
<CENTER><H2>Кайла - том 2</H2x/CENTER>
_ END_
cool_cat.jpg: Посмотри на меня
kev.jpg: Папа
kyla_smile.jpg: Кайла улыбается
```

Как можно видеть, составить этот файл очень просто. Все, что находится перед маркером ___END___, будет помешено в заголовке страницы. Далее следуют имя файла изображения, двоеточие и желаемая подпись. Не обязательно составлять подпись для каждого изображения в каталоге. Если строки для данного файла в `caption.txt` не будет, в подписи будет просто использовано имя файла, как раньше. Когда при.веденный выше файл будет помешен в каталог `vol2`, фотоальбом этого каталога будет выглядеть в Web так, как показано на рис. 9.6.

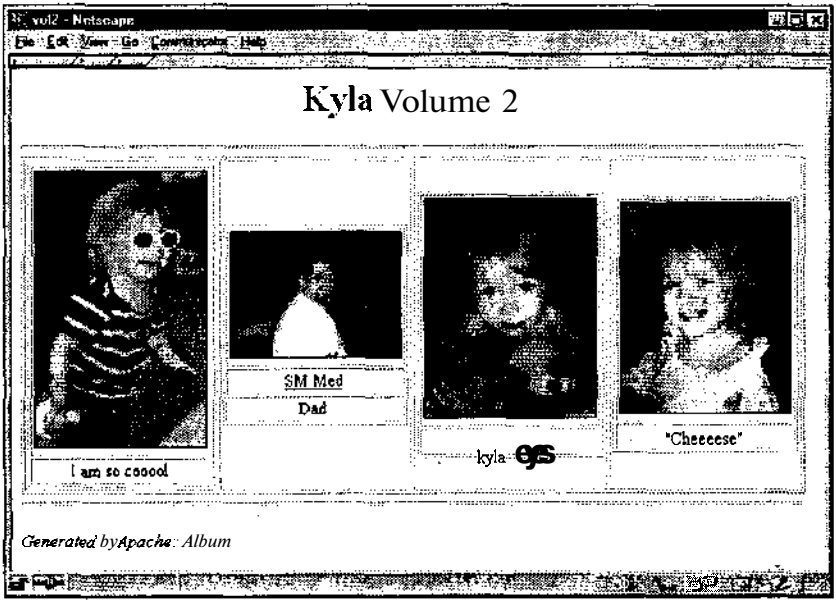



*Рис. 9.6. Фотоальбом с подписями*

Теперь, когда фотоальбомы можно просматривать, нам остается реализовать простой и удобный способ создания новых альбомов и загрузки изображений. И это позволяет сделать Apache: `:Album` — по сути, целое приложение, замаскированное под модуль. Чтобы включить в этом модуле *режим правки*, надо добавить в раздел конфигурации одну строку.

```
PerlSetVar EditMode 1
```

Теперь после перезапуска сервера пользователи смогут создавать на главной странице, на которой помещен список доступных альбомов (см. рис. 9.3), новые альбомы и загружать в них новые изображения из своего броузера. Но вряд ли вы действительно захотите дать эту возможность всем и каждому. Если просто позволить всем пользователям создавать каталоги и загружать файлы, это будет противоречить требованиям безопасности. Чтобы гарантировать, что эта возможность будет предоставлена только тем, кому вы доверяете, надо добавить какую-то операцию по идентификации пользователей. Мы не можем просто защитить каталог `/kyla/albums` паролем, так как те, кто не знает пароля, не смогут просмотреть альбом. Так Apache: `:Album` работает с любым каталогом, который задан в переменной `AlbumDir`, мы можем создать другой раздел конфигурации с указанием другого каталога и реализовать в нем базовую идентификацию пользователей. Таким образом, широкие массы пользователей смогут просматривать изображения, но право модифицировать альбомы будет предоставлено только избранным. Новый раздел конфигурации показан в листинге 9.16.

**Листинг 9.16. Раздел конфигурации с идентификацией**

```
' <Location /kyla/secure>
   SetHandler perl-script
   AllowOverride None
   AuthName "Edit Albums"
   AuthType Basic
   AuthUserFile /usr/local/apache/htdocs/albums/
   .kyla_access
```

```
07:     <Limit GET POST PUT>
08:       require valid-user
09:     </Limit>
10:     PerlSetVar EditMode 1
11:     Options None
12:     PerlHandler Apache::Album
13:     PerlSetVar AlbumDir /kyla/albums_loc
14:     PerlSetVar OutsideTableBorder 1
15:     PerlSetVar InsideTableBorder 1
16:     PerlSetVar BodyArgs BGCOLOR=white
17:     PerlSetVar AllowFinalResize 1
18:     PerlSetVar FinalResizeDir thumbs
19:   </Location>
```

В этом листинге по сравнению с предыдущей конфигурацией добавлены только **строки 4–10**, поэтому мы сосредоточимся на них. Как можно заметить, строка 1 указывает уже на другой каталог — `/kyla/secure`. Следовательно, этот раздел начнет действовать, когда кто-нибудь вызовет URL `http://www.me.com/kyla/secure`.

**В строке 4** устанавливается имя для нашей защищенной области. Это имя будет отображено в диалоговом окне, которое выводится при попытке входа в этот раздел без идентификации. Пример показан на рис. 9.7.

**В строке 5** устанавливается тип идентификации — базовый.(Basic).

**В строке 6** в директиве `AuthUserFile` указывается, где Apache должен искать комбинации имен пользователей и паролей (в файле `/usr/local/apache/htdocs/albums/. kyla_access`).

**Строки 7–9** — директива Limit, в которой запросы типа GET, POST и PUT допускаются только для разрешенных пользователей. Разрешенным пользователем считается тот, кто успешно прошел идентификацию.

На рис. 9.8 показан список альбомов, который отображается после идентификации в новом защищенном разделе. Обратите внимание на текстовое поле, которое позволяет создавать новые альбомы. При создании альбома действительно создается подкаталог в каталоге, указанном в `AlbumDir`. Рекомендуется включать проверку на загрязнения (`PerlTaintCheck On`), если предоставляется режим правки. Включен при предоставлении режима редактирования. Apache: `:Album` сам выполняет очистку имен каталогов, но режим проверки на загрязнение никогда не мешает при работе в CGI.

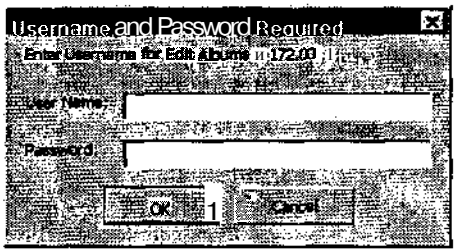



*Рис. 9.7. Диалоговое окно идентификации*

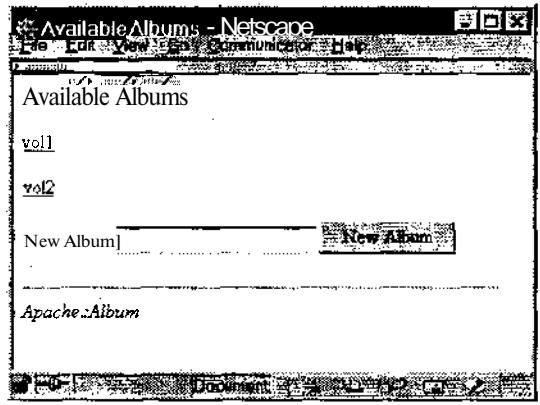



*Рис. 9.8. Список альбомов в режиме правки*

Итак, не написав ни единой строки скрипта, мы получили работоспособное приложение, которое дает доступ к фотоальбомам вашим друзьям, родным и сотрудникам. Этим приложением можно легко управлять (создавать альбомы и загружать изображения) через Web откуда угодно. Мы еще раз рекомендуем вам перечитать документацию, чтобы узнать, какие еще возможности конфигурации могут быть полезны для вас.

# Идентификация с помощью Apache::AuthDBI

В предыдущем примере мы защитили каталог паролем, что не требует применения mod_perl. Однако mod_perl дает возможность реализовать идентификацию с хранением имен пользователей и паролей в базе данных, а не в простом файле. Этого позволяет добиться модуль Apache::AuthDBI — еще один обработчик modperl, который обслуживает идентификацию и проверку подлинности. В данном разделе мы применим его для защиты каталога для правки фотоальбомов.

В первую очередь нам понадобится таблица базы данных, в которой будут храниться имена пользователей и пароли. Эта таблица будет называться album_users и будет включать только две колонки. Для ваших приложений структура этой таблицы может быть какой угодно. В ней можно хранить не только имя пользователя и пароль, но и какие-нибудь сведения о пользователях. Схема таблицы показана в листинге 9.17.

**ЛИСТИНГ 9.17. Схема простой таблицы ДЛЯ Apache::AuthDBI**

```
CREATE TABLE album_users (
  user varchar(50),
  password varchar(50)
)
```

Следующий этап — сконфигурировать Web-сервер так, чтобы он использовал Apache::AuthDBI для идентификации. Как и в случае с Apache:Album, для этого не нужно никакого скрипта, все выполняется в файле конфигурации. В листинге 9.18 показан раздел конфигурации, который мы будем использовать.

**Листинг 9.18. Раздел конфигурации ДЛЯ Apache::AuthDBI**

```
01:  PerlModule Apache::AuthDBI
02:   <Locafion /kyla/secure>
03:     SetHandler perl-script
04:    AllowOverride None
05:    AuthName "Edit Albums"
06:    AuthType Basic
07:     PerlAuthenHandler Apache::AuthDBI::authen
08:     PerlSetVar Auth_DBI_data_source
       dbi:mysql:database=book;host=localhost
09:     PerlSetVar Auth_DBI_username       public
10:     PerlSetVar Auth_DBI_password       foobar
11:     PerlSetVar Auth_DBI_pwd_table      album_users
12:     PerlSetVar Auth_DBI_uid_field      user
13:     PerlSetVar Auth_DBI_pwd_field      password
14:     PerlSetVar Auth_DBI_placeholder    on
15:     PerlSetVar Auth_DBI_nopasswd       on
16:     require valid-user
17:    Options None
18:     PerlHandler Apache::Album
```

```
19:    PerlSetVar EditMode 1
20:    PerlSetVar AlbumDir /kyla/albums_loc
21:    PerlSetVar OutsideTableBorder 1
22:    PerlSetVar InsideTablesBorder 1
23: </Location>
```

В этом разделе конфигурации вновь появились только **строки** 1 и **7—15,** которые заставляют Apache использовать Apache: ;AuthDBI и конфигурируют сам модуль.

**Строка** 1 указывает, что Apache должен загружать модуль Apache: :AuthDBI при своем запуске. Это также можно сделать, поместив строку use `Apache::AuthDBI` в `startup.perl`, но для наглядности и чтобы лучше запомнить, мы задали загрузку Apache: `:AuthDBI` в конфигурации.

**В строке 7** директива PerlAuthenHandler заставляет Apache использовать Apache: `:AuthDBI::authen` для обработки всех поступающих запросов на каталог, защищенный паролем. Так как эта директива стоит внутри директивы Location, она будет применяться, только если запросы относятся к указанному в Location дереву каталогов.

**Строка 8** инициализирует переменную `Auth_DBI_data_source`, по которой Apache: `:AuthDBI` определяет, к какой базе данных надо подключиться. Указанное значение — это DSN, который `DBI.pm` будет использовать для подключения.

**В строке 9** устанавливается переменная `Auth_DBI_username`, которая содержит имя пользователя для подключения через `DBI.pm` к базе данных.

**Строка 10** устанавливает пароль (`Auth_DBI_password`) для подключения к базе данных. Здесь *не следует* применять имя и пароль учетной записи, которая имеет полные привилегии для базы данных. Поскольку файл `httpd.conf` может прочитать любой, у кого есть учетная запись на сервере, надо добиться, чтобы подключение к базе имело минимум привилегий. В этой ситуации пользователю нужен только доступ для чтения, так как `AuthDBI.pm` используется только для чтения. Если требуется, чтобы `AuthDBI.pm` изменял какие-то поля (что не рассматривается в этой книге, но в документации к модулю рассказано, как модифицировать поле для регистрации), надо будет только придать учетной записи соответствующие привилегии. В документации по Apache: `:AuthDBI` также показано, как получить и задать `Auth_DBI_username` и `Auth_DBI_password` без отображения этих значений в открытом тексте.

**Строка 11** определяет `Auth_DBI_pwd_table` — таблицу базы данных, в которой модуль будет искать имена пользователей и пароли.

В **строках** 12 и 13 определяются переменные `Auth_DBI_uid_field` и `Auth_DBI_pwd_field`, значения которых — поля базы данных, в которых содержатся соответственно имена пользователей и пароли.

В **строке** 14 включается опция `Auth_DBI_placeholder` для `AuthDBI.pm`. Если база данных поддерживает метки-заполнители, как, например, MySQL, включение этой опции может повысить производительность. В результате Apache: `:AuthDBI` будет использовать в запросе на извлечение пароля метку-заполнитель вместо имени пользователя. Таким образом, этот запрос можно подготовить однажды, а не формировать каждый раз заново, что экономит время при проверке подлинности пользователей.

**Строка** 15 не особенно подходит для данного приложения (фотоальбома), но нам кажется, что важно продемонстрировать эту особенность `AuthDBI.pm`. Если опция Auth_DBI_nopasswd включена (on), то идентификацию может пройти любое имя пользователя, введенное без пароля. Это может быть полезно, если требуется открыть защищенное приложение или каталог для гостевого доступа. Конечно, программист может ограничить доступ и на основе подлинного имени пользователя (из `$ENV['REMOTE_USER']`), но эта возможность также может быть полезной.

Apache: :AuthDBI проверяет пароли, зашифрованные функцией Perl *crypt ()*. По умолчанию ключом {вторым параметром, "солью" шифрования) для *crypt ()* является имя пользователя, т.е., когда при идентификации вводятся имя пользователя и пароль, происходит последовательность событий, показанная в листинге 9.19.

**Листинг 9.19. Ход идентификации**

```
01:  SELECT password FROM table WHERE username = имя_пользователя
02:  если password = crypt ("пароль", "имя_пользователя")
     то идентифицировать
03:  иначе запретить доступ.
```

В модуле имеется еще две опции: первая предоставляет имени пользователя приоритет над паролем, а вторая разрешает использование паролей в открытом тексте. Первая опция устанавливается следующей строкой в разделе конфигурации.

```
PerlSetVar Auth_DBI_encryption_salt userid
```

Если нужно, чтобы пароли не шифровались, добавьте следующую строку.

```
PerlSetVar Auth_DBI_encrypted of.f
```

В нашем примере пароли будут шифроваться и ключом для шифрования будет имя пользователя. Мы предоставляем читателю создать Web-приложение или программу для командной строки, позволяющую добавлять имена пользователей в базу данных. Подробнее об этом мы поговорим в конце главы.

# Создание обработчика **mod_perl**

Вы уже научились задавать базовую конфигурацию mod_perl, а также использовать некоторые модули Apache как обработчики. В этом разделе мы расскажем, как написать на Perl собственный обработчик mod_perl. Создание обработчика mod_perl особенно не отличается от создания других модулей Perl, с тем исключением, что обработчики могут непосредственно использовать Apache Perl API и главный метод обычно называется *handler()*. По умолчанию, если модуль выполняет функцию обработчика (*Perl*Handler*), метод *handler()* вызывается для управления запросом или действием. Другие особенности касаются переменных с лексической и глобальной областью действия. Глобальные переменные в модуле Apache используются совместно, пока существует процесс HTTPD. Иными словами, если какой-то дочерний процесс должен управлять 40 запросами и использует глобальную переменную Sglobal, то, когда один из запросов изменяет ее, это значение изменяется для всех запросов к данному дочернему процессу. Пока вы на собственном опыте не прочувствуете эту концепцию, лучше свести количество изменяемых глобальных переменных к минимуму.

Обработчик mod_perl, который мы рассмотрим, будет управлять фазой регистрации запроса. В журнале доступа Web-сервера содержится масса полезной информации, но, чтобы придать ей легко доступную форму, как правило, нужна какая-то дополнительная программа. Если поместить информацию, которая записывается в журнал доступа, в базу данных, к ней можно будет обращаться легко и быстро. Именно это будет делать наш модуль Apache: :MyLog. В первую очередь надо решить, какая информация должна фиксироваться. В данном случае мы будем записывать имя удаленного узла, имя пользователя (если он прошел проверку), штамп времени, запрашиваемый URI, состояние запроса (200, 404 и т.д), число байт, переданных клиенту, метод запроса (GET, POST И Т.Д.), ИСТОЧНИК ССЫЛКИ И ТИП броузера. Затем, зная, какая информация должна быть записана, мы создадим таблицу базы данных, которая будет ее хранить (листинг 9.20).

Листинг 9.20. Команда SQL, создающая таблицу **request_log** для **Apache::MyLog**

```
CREATE TABLE request_log (
  remote_host varchar(255) DEFAULT '' NOT NULL,
  user varchar(50),
  time_stamp datetime DEFAULT '0000-00-00 00:00:00' NOT NULL,
  requested varchar(255) DEFAULT '' NOT NULL,
  status smallint(3) DEFAULT '0',
  bytes int(8),
  method varchar(8) DEFAULT '' NOT NULL,
  referer varchar(255),
  browser varchar(255)
)
```

Эта таблица называется request_log и имеет соответствующие поля для данных, которые должны быть сохранены. Кроме того, необходим сам модуль. Сейчас мы проанализируем код этого модуля, в котором используются некоторые элементы Apache Perl API. Но прежде следует показать раздел конфигурации, который инициализирует этот модуль для обработки фазы регистрации. Это помогает понять контекст, в котором модуль будет работать. Конфигурация показана в листинге 9.21.

**ЛИСТИНГ 9.21. Раздел конфигурации ДЛЯ Apache::MyLog**

```
PerlLogHandler Apache::MyLog
PerlSetVar MyLog_dsn dbi:mysql:database=book;host=localhost
PerlSetVar MyLog_user public
PerlSetVar MyLog_pwd foobar
PerlSetVar MyLog_table request_log
```

Так как этот модуль должен управлять фазой регистрации запроса, мы должны информировать mod_perl и Apache об этом, задав директиву PerlLogHandler. Текст после этой директивы уже должен быть вам знаком. Как и для Apache::AuthDBI, здесь устанавливаются переменные конфигурации, из которых модуль получает информацию DSN, информацию для входа а базу данных и имя нужной таблицы. Эти сведения не фиксируются в самом модуле, что позволяет ему подключаться к разным базам данных на разных серверах или использовать в базе данных различные таблицы для различных каталогов. Этот блок должен быть помещен в директиве Location или Directory или записан глобально. А теперь перейдем к коду.

```
01: package Apache::MyLog;
```

**Строка 1** определяет имя пакета. Модули Apache, используемые как обработчики, обычно располагаются в пространстве имен Apache::*; имя самого модуля — MyLog.pm. В пространстве имен Apache::* находятся не все модули для mod_perl, например, HTML::Mason, но те из них, которые специфичны для Apache или mod_perl, располагаются там (или должны располагаться).

```
02: use strict;
03: use Apache ();
04: use DBI ();
05: use Apache::Constants qw(:common);
06: use POSIX;
```

**В строках 2–6** загружаются модули, необходимые для остальной части MyLog.pm. Модуль Apache.pm импортирует методы Perl Apache **API;** DBI.pm позволяет выполнять подключения к базе данных; Apache::Constants импортирует обычно используемые константы HTTP (OK, DECLINED и т.д.), а модуль POSIX.pm позволит нам форматировать дату и время.

```
07:  $Apache::MyLog::VERSION = '0.01';
```

Строка 7 определяет скалярную переменную, которая содержит номер версии этого модуля. Если модуль предназначается для распространения, модуль `MakeMaker.pm` создает make-файл для инсталляции его дистрибутива и при этом определяет по скаляру `$Package::VERSION` версию предназначенного для инсталляции модуля. Для наглядности мы непосредственно задали этот скаляр в его пространстве имен, а не объявили глобальную переменную `$VERSION` с прагмой var, как обычно делается.

```
08:  sub handler {
09:     my $r = shift;
```

В **строках** 8 **и** 9 начинается метод `handler()` и инициализируется скаляр `$r` как ссылка на объект запроса. Объект запроса всегда передается в метод *handler()* первым параметром. Это все равно, что задать $self в объектно-ориентированных модулях Perl. Через переменную `$r` программист сможет обращаться к Apache Perl API.

```
10:     return DECLINED unless ($r->is_main);
```

**В строке 10** метод Apache Perl API *is_main()* определяет, относится ли `$r` к основным запросам HTTP (это может быть и подзапрос, например, перенаправление или включенный файл). Нас интересует только основной запрос, и, если это не так, метод возвращает значение DECLINED, и обработка запроса продолжается в другом месте.

```
11:     my %Config = ('dsn'=> $r->dir_config('MyLog_dsn'),
12:                   'user'=> $r->dir_config('MyLog_user'),
13:                      'pwd'=> $r->dir_config('MyLog_pwd'),
14:                     'table'=> $r->dir_config('MyLog_table')
15:                     );
```

**Строки 11—15** инициализируют хэш `%Config`. Этот хэш содержит четыре клавиши, которые будут затем использоваться в модуле. Значения в этот хэш записываются из переменных, заданных в разделе конфигурации. Чтобы обратиться к этим переменным, надо вызвать метод *dir_config()* передать в него имя нужной переменной конфигурации.

```
16:     if (!$Config{dsn} ||
            !$Config{user} ||
            !$Config{pwd}||
        !$Config{table}) {
17:        $r->log_error("MyLog.pm сконфигурирован неправильно.");
18:        return DECLINED;
19:     )
```

**В строках 16—19** выполняется "санитарная проверка", действительно ли переданы все ожидаемые переменные конфигурации. Если какое-то из этих четырех значений не было установлено в файле конфигурации, возвращается DECLINED, и модуль прекращает работу.

```
20:  ray  $dbh = DBI->connect($Config{dsn},
        $Config{user}, $Config{pwd});
```

**В строке 20** производится подключение к базе данных с использованием сведений, полученных из файла конфигурации.

```
21:     if (!$dbh) {
22:       $r->log_error("Не удалось подключиться к базе
           данных. DBI::errstr");
23:       return DECLINED;
24:     }
```

**Строки 21-24** — проверка, прошло ли подключение к базе данных успешно. Если это не так, в журнал ошибок Web-сервера делается соответствующая запись, в которую включается сообщение об ошибке, полученное от модуля `DBI.pm`. Затем **строка** 23 возвращает DECLINED и модуль прекращает работу, не регистрируя никакой информации.

```
25:  ray  @INFO = ($r->get_remote_host,
26:                $r->connection->user,
27:               strftime("%Y-%m-%d %I:%M:%S", localtime),
28:                $r->uri,
29:                $r->status,
30:                $r->bytes_sent,
31:                $r->method,
32:                $r->header_in('Referer'),
33:               $r->header_in( 'User-Agent')
34:              ) ;
```

**В строках 25—34** создается массив @INFO. Этот массив содержит всю информацию, которая будет вставлена в базу данных в порядке, соответствующем расположению полей таблицы. Все элементы массива, кроме одного, инициализируются и использованием Apache Perl API. **Строка 25** методом *get_remote_host()* получает адрес, с которого подключается клиент. Это то же значение, которое в скрипте CGI можно получить из $ENV[ 'REMOTE_HOST' ]. **Строка 26** и метод connection->user производят действие, эквивалентное вызову $ENV['REMOTE_USER']. Значение здесь возвращается, только если клиент прошел проверку в защищенной паролем области. Затем, **в строке 27** методом *POSIX::strftime* форматируется строка даты и времени, которая будет помещена в поле базы данных time_stamp. **В строке 28** методом *uri()* возвращается значение запрашиваемого URI. Состояние запроса определяется в **строке 29** методом *status ()*. Это состояние — численный эквивалент констант OK, NOT_FOUND И Т.Д. Например, при успешной обработке документа возвращается значение состояния 200. Полный список кодов состояния можно найти в приложении А. Поскольку фаза очистки следует после фазы регистрации, клиент уже получил требуемый документ, и состояние запроса стало доступным. **В строке 30** определяется количество байтов, переданных клиенту. Может показаться неожиданным, когда в этом поле базы данных вы увидите значение 0, так как файл всегда имеет какой-то размер. Это происходит, когда запрашиваемый документ выдается из кэша, тогда файл физически не передается клиенту с сервера. Метод *method()* в **строке 31** возвращает метод, примененный при запросе клиента. Как правило, это GET, POST, HEAD ИЛИ PUT. **В строке 32** метод *header_in()*, в который передается параметр Referer (страница ссылки), возвращает URL Web-страницы, на которой находился клиент, когда сделал запрос. Не существует страницы ссылки и в базу данных записывается NULL, если клиент перешел к запрашиваемой странице непосредственно с закладки или введя адрес в поле URL броузера. Наконец, параметр *User-Agent,* переданный в метод *header_in()*, возвращает информацию о броузере. Вообще, метод *header_in()* служит для извлечения информации из заголовка запроса и в него передается имя нужного поля заголовка. Теперь можно сказать, что формирование массива @INFO закончено.

```
35:   my $query = "insert into $Config{table} (
                  remote_host, user,
                  time_stamp, requested, status,
36:               bytes, method, referer,
                  browser) values (" . (join(",", ("?") x
                  @INFO)) . ")";
```

**В строках 35 и 36** в переменную Squery записывается команда SQL, которая вставляет запись в базу данных. Таблица для вставки указывается выражением $Config{table I, значение которого получено из файла конфигурации в **строке 14. В** этой команде SQL используются метки-заполнители: в **строке 36** указывается соответствующее количество вопросительных знаков, которые через запятые объединяются в одну строку функцией *join ()*.

```
37:    my $sth;
38:    unless ($sth = $dbh->prepare($query)) {
39:      $r->log_error("Нельзя подготовить команду:
```

```
                        $DBI::errstr");
40:       $dbh->disconnect;
41:       return DECLINED;
42:    }
```

**В строках 37—42** запрос SQL подготавливается (методом *prepare()*). Если *prepare()* завершается неудачно, в журнал ошибок Web-сервера записывается сообщение, включающее текст, выдаваемый функцией DBI: :errstr. Метод *log_error()* дописывает в журнал ошибок Web-сервера любую строку, переданную ему как параметр. Затем **строка 40** закрывает подключение к базе данных, а **строка 41** возвращает DECLINED.

```
43:    $sth->execute(@INFO);
44:    $dbh->disconnect;
45:    return OK;
46: }
47: 1;
48:__END__
```

**Строки 43—48** завершают модуль. **В строке 43** подготовленный запрос выполняется методом *execute()*, в который передается массив @INFO. Его значения подставляются вместо меток-заполнителей в подготовленной команде. **Строка** 44 производит отключение от базы данных, а **строка** 46 возвращает состояние OK. Метод *handler()* завершается в **строке 46,** а **строка 47** возвращает значение истины туда, куда этот модуль импортируется командой *use()*. **Строка 48** — маркер__END__, за которым должна следовать документация.

В этой главе вы узнали, что такое mod_perl и как выполнить его базовую конфигурацию. Вы также научились применять различные модули Apache для кэширования, создания колонтитулов, идентификации через базу данных, составления фотоальбомов и научились создавать собственные обработчики mod_perl. Все это позволяет применять mod_perl при создании Web-приложений CGI на Perl.

# Упражнения

- Посетите CPAN и просмотрите обработчики mod_perl, которые там имеются. Найдите обработчик, который кажется вам интересным и полезным, и установите его.
- По принципу Apache: : Sandwich напишите обработчик, который будет показывать случайные меняющиеся изображения. За основу можно взять скрипты из этой главы.
- На основе Apache: :DBI дополните MyLog.pm так, чтобы подключение к базе данных было постоянным.

# Листинги

### Листинг 9.22. MyLog.pm

```
01: package Apache::MyLog;
02: use strict;
03: use Apache ();
04: use DBI ();
05: use Apache::Constants qw(:common);
06: use POSIX;
07: $Apache::MyLog::VERSION = '0.01';
08: sub handler {
09: my $r = shift;
```

```
10:    return DECLINED unless ($r->is_main);
11:    my %Config = ('dsn'=> $r->dir_config('MyLog_dsn'),
12:                  'user'=> $r->dir_config('MyLog_user'),
13:                  'pwd'=> $r->dir_config('MyLog_pwd'),
14:                  'table'=> $r->dir_config('MyLog_table')
15:                 );
16:   if (!$Config{dsn} || |
          !$Config{user} | j
          !$Config{pwd} ||
      !$Config{table}) {
17:      $r->log_error("MyLog.pm сконфигурирован неправильно,");
18:      return DECLINED;
19:    }
20:    my Sdbh = DBI->connect($Config{dsn},
        $Config{user}, $Config{pwd});
21:   if (!$dbh) (
22:      $r->log_error("Не удалось подключиться к базе
          данных. DBI::errstr");
23:      return DECLINED;
24:    }
25:   my @INFO = ($r->get_remote_host,
26:                $r->connection->user,
27:               strftime("%Y-%m-%d %I:%M:%S", localtime),
28:                $r->uri,
29:                $r->status,
30:    •           $r->bytes_sent,
31:                $r->method,
32:                $r->header_in('Referer'),
33:               $r->header_in( 'User-Agent')
34:              );
35:   my $query = "insert into $Config{table} {
                   remote_host, user,
                   time_stamp, requested, status,
36:                bytes, method, referer,
                   browser) values (" . (join(",", ("?") x
                   @INFO)) . " ) " ;
37:   my $sth;
38:   unless ($sth = $dbh->prepare($query)) {
39:      $r->log_error("Нельзя подготовить команду:
                        $DBI::errstr");
40:      $dbh->disconnect;
41:      return DECLINED;
42:    }
43:    $sth->execute(@INFO):
44:    $dbh->disconnect;
45:    return OK;
46: }
47: 1;
48: __END__
```


# Электронная почта на базе Web

## Введение

Похоже, что в наше время каждый хочет иметь доступ к своей электронной почте через Web. Возможность проверять электронную почту с любого компьютера, на котором есть доступ в Internet, и удобна и полезна. Такие службы, как Hotmail, Yahoo!, Lycos и множество других, предоставляют для этого бесплатный интерфейс, но эти службы имеют свои недостатки. Один из них состоит в том, что эти службы должны хранить данные вашей учетной записи POP (имя пользователя и пароль), чтобы иметь возможность проверять почтовый ящик. В 1999 г. одна популярная бесплатная почтовая служба была скомпрометирована "дырой" в своем программном обеспечении, которая давала возможность просматривать какую угодно почту, зная только имя учетной записи этого пользователя. Другой недостаток заключается в том, что вы не можете *посылать* почту со своей учетной записи POP, а только из этой "бесплатной службы". Сегодня, когда через такие службы идет весь спам, отправка сообщения с бесплатной учетной записи выглядит несколько непрофессионально. Когда ваш предполагаемый работодатель, начальник или просто знакомый посылает вам письмо на `you@you.com`, часто бывает более разумно направить ответ с you.com, а не с одной из этих бесплатных учетных записей. Не будем отрицать, эти службы действительно имеют свои достоинства.[1] Но не лучше ли будет, если проверять почту на учетной записи POP и отправлять с нее сообщения станет легче и проще? Разве неплохо пред-

---

[1] *Они обладают многими функциями, которые не включены в это базовое приложение. Тем не менее, используя знания, полученные из этой книги, вы сможете добавить их сами.*

ложить вашим клиентам возможность проверять свою почту через Web? Как раз это и позволяет сделать настоящая глава.

В этой главе вы научитесь использовать Perl для создания Web-приложения, которое позволяет проверять электронную почту, читать сообщения, отвечать на них и составлять новые. Это приложение также дает возможность присоединять к сообщениям вложения (attachments), а также просматривать и загружать вложения в почте, пришедшей на вашу учетную запись POP.

Также в этой главе мы введем некоторые новые модули, и вы сможете ознакомиться с ними, чтобы применять в дальнейшем. Эти модули выполняют все операции соединения с серверами POP и SMTP, а также отправку и получение вложений и производят всю рутинную работу, которая не должна вас заботить. Мы будем использовать следующие прагмы и модули.

- `strict`
- `lib`
- `CGI`
- `Mail::POP3Client`
- `Untaint`
- `URI::Escape`
- `Net::SMTP`
- `CGI::Carp`
- `MIME::Parser`
- `Mail::Address`

# Пример: проверка почты POP3 через Web

В первую очередь мы должны предоставить пользователям путь, которым они смогут войти в "почтовую систему". В ходе этого процесса входа в систему мы должны будем выяснить три вещи: имя пользователя, пароль и сервер POP, с которого надо получить почту. Каким образом лучше всего получить эти данные? Разумеется, в первую очередь возникает мысль об использовании механизма базовой идентификации. Но цель этой идентификации — проверить подлинность пользователя для просмотра Web-страниц, что нам не требуется; в нашем случае пользователь должен только войти на сервер POP. Кроме того, при базовой идентификации можно передать в скрипт только имя пользователя и пароль, но не имя сервера POP. Возможный выход — сформировать имена пользователей наподобие `user@pop_server`, а в качестве пароля передавать пароль учетной записи POP. Но если вы меняете свой пароль учетной записи POP (ведь все пароли принято регулярно менять, не так ли?), вам придется воспользоваться утилитой htpasswd или интерфейсом к ней, чтобы изменить пароль в самой системе. Кроме того, если у вас несколько учетных записей POP, надо будет создать для этого приложения эквивалентное количество пользователей. Конечно, все это несложно реализовать в Perl, но в данном случае мы не хотели бы вообще сохранять никакой информации о пользователе. Таким образом, это приложение должно работать в единственном экземпляре и, при необходимости, к нему могут совместно обращаться несколько пользователей. Так как же тогда нам получать имя пользователя POP, пароль и сведения о сервере и где временно хранить это в Web-клиенте? Конечно, через Web-форму и в cookie!

## Примечание

Если какая-то операция связана с передачей паролей через Internet открытым текстом, следует принять дополнительные меры предосторожности. Рекомендуется использовать SSL или шифровать пароль, а затем расшифровывать его в скрипте. Мы предоставляем это читателю в качестве упражнения, чтобы он сам выбрал наилучший способ защиты подобной информации. Мы рекомендуем применять для шифрования пароля функцию Perl `crypt()` или ту схему шифрования, которую вы предпочитаете.

Если создать форму, в которой будут запрашиваться три эти элемента информации, вся работа по идентификации фактически будет производиться сервером POP, а клиент будет сохранять эту информацию. Вплоть до конца "сеанса" пользователя в этой системе ему не нужно заполнять форму входа в систему — конечно, если с самого начала была представлена правильная информация. Это также облегчает открытие нескольких учетных записей POP из одного и того же набора скриптов.

В листинге 10.1 (`pop-login.html`) показан код HTML для Web-формы, которую мы будем использовать. Для этого достаточно обычного файла HTML — нет необходимости генерировать его динамически скриптом Perl.

### Листинг 10.1. HTML Web-формы для входа в систему

```
<HTML>
<HEAD>
<TITLE>Вход в систему электронной почты</TITLE>
</HEAD>
<BODY>
<CENTER>
<TABLE CELLPADDING=4 CELLSPACING=4>
<TD ALIGN=center COLSPAN=2>
<B>Регистрация пользователя</B></TD><TR>
<FORM METHOD=POST ACTION=index.cgi">
<TD ALIGN=right>Имя пользователя:</TD>
<TD ALIGN=left>
<INPUT TYPE="text" NAME="username" si2e=25x/TD><TR>
<TD ALIGN=right>Пароль :</TD>
<TD ALIGN=left>
<INPUT TYPE="password" NAME="password" size=25x/TD><TR>
<TD ALIGN=right>Сервер:</TD>
<TD ALIGN=left>
<INPUT TYPE="text" NAME="server" size=25></TD><TR>
<TD ALIGN=center COLSPAN=2>
<INPUT TYPE=submit VALUE="Войти"></TD>
</FORM>
</TABLE>
</CENTER>
</BODY>
</HTML>
```

Все, что надо помнить об этой форме — то, что она передает три значения: имя пользователя, пароль и сервер. На рис. 10.1 показано, как она будет выглядеть. А теперь перейдем к более интересным вещам.

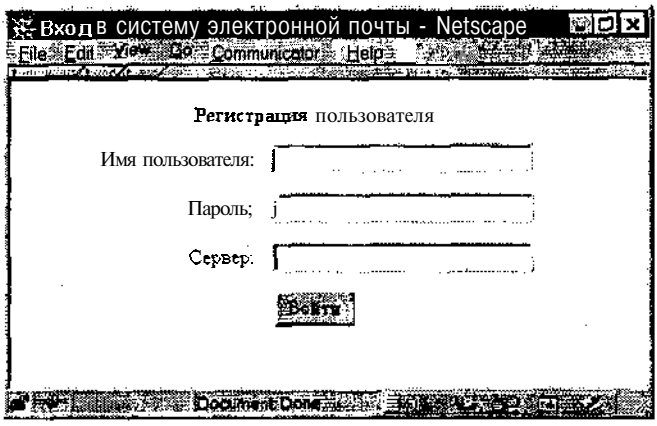



*Рис. 10.1. Форма для входа в систему*

Первый скрипт — `index.cgi` — выполняет подключение к серверу. **POP** при помощи модуля Mail: `:POP3Client`, который анализирует заголовки всех сообщений на сервере **POP** и отображает список сообщений, имеющихся на сервере.

```
01:  #!/usr/bin/perl -wT
02:  # index.cgi
03:  $|=1;
04:  use strict;
05;  use Mail::POP3Client;
06:  use CGI" qw(:standard :netscape *table);
07:  use CGI::Carp qw(fatalsToBrowser);
08:  use Untaint;
```

**Строки 1—8** вам хорошо знакомы. Здесь мы включаем предупреждения, проверку на загрязнения, автосброс буфера вывода и импортируем модули, которые будем использовать. Mail: `:POP3Client` — единственный новый модуль в этом скрипте, и мы будем подробно говорить о нем в ходе изложения.

```
09:  my $cookie = cookie('WebMail') || 0;
```

В **строке 9** методом `CGI.pm` *cookie ()* проверяется, существует ли cookie под названием `WebMail`. Если это так, значение cookie присваивается переменной `$cookie`. Если же оно не существует, то левая часть логического "или" (`||`) не выполнится, и Scookie получит значение 0. Это важно, так как из cookie мы получаем данные пользователя, необходимые для подключения к серверу **POP.** К концу главы в почтовую систему будет добавлена функция "выхода", когда устанавливается значение $cookie О.[2] Поэтому мы сейчас присваиваем Scookie нулевое значение, чтобы в случае, если cookie не существует, это было равносильно тому, что пользователь не вошел в систему.

```
10:  my $user = param('username') || 0;
11:  my Spass = param('password');
12:  my $server = param('server');
```

**В строках 10—12** опрашиваются данные формы. Вспомните, что форма для входа в систему содержит три элемента: username, password и server, соответственно для имени пользователя, пароля, и сервера. Значения, переданные в этих элементах фор-

[2] *Единственная проблема в этом скрипте может возникнуть, если какой-то пользователь имеет пароль "0". Но гораздо большую проблему это создаст для данного пользователя. Это все равно, что иметь код кредитной карточки наподобие "1234".*

мы, присваиваются соответственно переменным $user, $pass и $server. Если из формы не передано никаких данных (пользователь пришел не со страницы входа в систему), переменные не будут иметь значений.

```
13:  if ((!$user || !$pass || !$server) && !$cookie) {
14:    print redirect('pop-login.html"); exit;
15:  }
```

**В строках 13—15** проверяется наличие данных, введенных из формы, или существование cookie. **Строка 13** начинается с условного выражения, которое выполняется, если не существует ("не" передается знаком !) значение $user или (логическое "или" — ||) $pass, или Sserver. Три этих операнда заключены в круглые скобки, так что сначала вычисляется часть выражения с ними, а результат используется в последующем выражении. Если этот результат — ложное значение, что означает, что в $user, $pass и $server есть ненулевые значения, то все условное выражение не выполняется, и скрипт продолжает работу. Если оказывается, что одна из этих переменных не имеет значения, то проверяется, нет ли также (логическое "и" — &&) значения для cookie. Если всех данных фермы не имеется, это означает, что пользователь неправильно заполнил форму; тогда он перенаправляется в нее опять методом CGI.pm *redirect ()*. Если в форме есть данные, пользователь возможно, ранее уже входил в систему, и это проверяется наличием cookie. Если cookie также не существует, то пользователь — новичок для этой системы, и он перенаправляется на страницу входа. Затем скрипт завершается (exit *()* в **строке 14).** Эта команда помешена на ту же строку, чтобы подчеркнуть, что после перенаправления скрипт должен завершить работу. В противном случае будут выполняться следующие операторы с неправильными данными и бессмысленным результатом.

```
16:  if (!$cookie) {
17:    $cookie = cookie(-name => 'WebMail',
18:                     -value => "$user $pass Sserver",
                         -path => '/'
19:    );
20:  print header(-cookie=>$cookie);
```

**Строки 16—20** — проверка существования cookie и действия, которые выполняются, если пользователь еще не имеет своих данных cookie. Условие в **строке 16** выполняется в ситуации, когда пользователь пришел со страницы входа. Если это действительно так, то устанавливается cookie сеанса. Имя этого cookie — WebMail, а значение — введенные в форму имя пользователя, пароль и сервер. Эти три значения объединяются пробелами в одно и присваиваются в строке 18. Также в этой строке устанавливается путь для cookie. Броузер при запросе cookie проверяет URL запрашивающего скрипта. Если URL не содержит путь, заданный для cookie, то броузер не отправит cookie на сервер. Это cookie — временное, так что когда сеанс броузера кончается (пользователь закрывает броузер), cookie перестает существовать.

```
21:  } else {
22:    print header;
23:    ($user, $pass, Sserver) = split(" ",$cookie);
24:  }
```

**Строки 21—24** выполняются, если предыдущее условие в **строках 16—20** не выполняется. Это происходит, если сеанс пользователя уже начался раньше. Тогда методом CGI.pm *header ()* выводится заголовок **HTTP,** а данные пользователя извлекаются из $cookie. Это делается путем простого разбиения cookie по разделяющим пробелам на составные части, которые присваиваются соответствующим переменным.

```
25:  $server = untaint(qr(.*), $server);
26:  $user = untaint(qr(^\w{1,8}$), $user);
27:  $pass = untaint(qr(^\w{1,8}$), $pass);
```

**В строках 25, 26 и 27** очищаются переменные, полученные из cookie или из Web-формы. Независимо от того, откуда были получены данные, они считаются загрязненными, так как пришли извне нашей программы, и должны быть очищены. Очистка производится методом `Untaint.pm` *untaint ()*. Можно заметить, что образец для имени пользователя и пароля, передаваемый в `untaint()`, соответствует любой строке из 1–8 букв и цифр. Для имени сервера принят образец, которому будет соответствовать все. Это сделано с целью примера; вам же придется установить образцы в соответствии с конкретными потребностями.

```
28:  system('/bin/rm","-rf","./$user~$server") unless
       $user !~ /^\w/;                    # UNIX
28:  system("delete","$user-$server");  # Windows
```

**Строка 28** — "безопасный" вызов *system (),* который удаляет все, что находится в каталоге пользователя. Вы можете спросить: "Что за каталог пользователя?" В следующем примере, когда мы перейдем к скрипту, который будет анализировать сообщения электронной почты для просмотра,будет рассмотрен модуль MIME;: Parser. Этот модуль, анализируя сообщение, помещает во временный каталог все его составные части, такие как тело сообщения и вложения. Имя этого каталога состоит из имени текущего пользователя, дефиса и имени сервера POP. Вот откуда появляется каталог пользователя. Мы можем удалить все содержимое этого каталога, поскольку знаем, что пользователь в данный момент не просматривает сообщение и не открывает вложения. Целесообразно удалить все файлы, которые могут остаться в этом каталоге, чтобы никто не мог прочесть их. Более подробно об этом каталогебудет рассказано позже в этой главе. **Строка 28** представлена в двух вариантах — для систем Win32 и для UNIX.

```
29:  ray $pop = new Mail::POP3Client(HOST => $server,
                                     AUTH_MODE => 'PASS');
```

**Строка 29** — вот где становится по-настоящему интересно. Здесь создается новый объект `Mail::POP3Client` и в нем сохраняется информация о сервере. Для обращения к объекту мы будем использовать переменную **$pop.** При формировании объекта задаются два параметра: HOST и AUTH_MODE. HOST — это сервер, на котором мы проверяем почту, а `AUTH_MODE` — тип авторизации, применяемый на сервере POP. По умолчанию принимается тип PASS (хотя здесь мы задаем его явно для ясности), но можно также присвоить значение APOP для авторизации APOP (`MD5`).

```
30:  $pop->User($user);
31:  $pop->Pass($pass);
```

**В строках 30 и 31** методами *user()* и *Pass()* в объект `Mail::POP3Client` передаются соответственно имя пользователя и пароль, указанные пользователем. Теперь этот объект имеет всю информацию для подключения.

```
32:  $pop->Connect || error($pop->Message);
```

Строка 32 выполняет фактическое подключение к серверу POP. Эта строка — по сути дела условный оператор. Его действие можно описать так: "Установить подключение к серверу POP. В противном случае вызвать подпрограмму *error()*". Если метод *Connect()* завершается неудачно, это означает, что подключение к серверу POP не установлено. В этом случае в нашу подпрограмму `error()`, о которой мы расскажем чуть позже, передается сообщение о неудаче. Если подключение прошло успешно, на что надо надеяться, скрипт продолжает работу. Один из недостатков этого метода состоит в том, что невозможно (на момент написания этого скрипта) определить, из-за чего не удалось подключение: из-за недоступности сервера в течение тайм-аута или из-за неверных входных данных.

```
33: ray Scount = $pop->Count;
```

**В строке 33** методом *Count()* определяется количество сообщений на сервере и это значение присваивается переменной `$count`. Это количество мы позже будем использовать при создании гиперссылок на сообщения для их просмотра или удаления.

```
34:  my %deleted;
```

**В строке 34** объявляется хэш `%deleted`, который поможет нам не включать в список удаляемые сообщения. Этот процесс мы подробно объясним несколькими строками ниже.

```
35: if (param('delete') ) {
36:     ($pop->Delete($_) && $deleted{$_}++)
        foreach (param('mess'));
37: }
```

**Строки 35-37** выполняют удаление сообщений. Скоро вы увидите, что, когда пользователь хочет удалить сообщение, в скрипт передаются две пары имя-значение: delete=1 и mess=номер сообщения. Если предпринимается удаление, параметр delete имеет значение ], что и проверяется в строке 35. Если это условие истинно, выполняется **строка 36. В** ней производятся два необходимых действия: во-первых, данное сообщение методом Delete() удаляется с сервера. Собственно говоря, это небольшое преувеличение. Функция *Delete()* на самом деле только помечает сообщение как удаленное, а настоящее удаление происходит, когда подключение к серверу POP закрывается. В этой главе мы еще расскажем об этом подробнее. Во вторых, в хэш `%deleted` добавляется новый элемент с ключом, равным номеру сообщения, которое удаляется. Значение хэша — просто 1. Конечно, здесь можно применить массив, но поиск в хэше не только быстрее и легче, но и хорошо вписывается в последующую подпрограмму. **Строка 36** выполняется в цикле по всем значениям mess, причем текущее значение сохраняется в $_. Одна маленькая строка может сделать столько работы!

**Строку** 36 можно лучше объяснить, если записать ее менее "компактным" способом. Ведь она выполняет много действий и может показаться непонятной, даже если вы знаете, что она делает. Следующий фрагмент кода — **строка 36** в более подробном формате.

```
my @messages = param('mess');
for (@messages) {
  $pop->Delete($_);
  $deleted{$_}++;
}
```

Во многих случаях для лучшей читаемости вы можете предпочесть этот стиль, но важно, чтобы вы поняли, что этот код имеет те же самые функциональные возможности, что и одна строка кода — **строка** 36. Теперь становится понятнее, что в ней происходит. Возвращаемое значение функции param( 'mess ') — массив, потому что его значения получены из списка выбранных флажков в форме. В цикле перебирается каждый элемент этого массива; значение текущего элемента передается в метод *Delete()* и обновляется хэш `%deleted`, который отслеживает, какие сообщения помечены для удаления.

```
38:  print start_html {{-title=>
        'Пример почтового клиента на базе Web',
39:    -BGCOLOR=>'white'}),
40:  p({-align=>'center'},
41:     start_table({-border=>1,
42:                  -cellpadding=>4,
43:                  -cellspacing=>4}),
44:     Tr,
45:     td({-align=>'CENTER',-colspan=>4},
           [strong("Пример почты")]),
```

```
46:     Tr,
47:     th({-align=>'CENTER'}, ['Удалить', 'От',
                                'Тема','Дата']),
48:     Tr,
49:     start_form({-method=>'POST',-action=>'index.cgi'}),
50:     hidden({-name=>'delete',-value=>'1'}))
51:   ;
```

**В строках 38—51** начинается код **HTML** страницы, на которой будут перечислены сообщения на сервере. При формировании страницы активно используются методы `CGI.pm`. В **строках** 38 **и** 39 выводится начало HTML, включая заголовок и цвет фона. В строке 40 начинается форматирование `<P ALIGN='CENTER'>`, которое выравнивает по центру последующую таблицу. **Строки 41—48** — начало таблицы, в которой будет отображена информация о сообщении. Затем начинается форма, которая отправляет свои данные в `index.cgi` (т.е. в сам этот скрипт), и в нее добавляется скрытый элемент с именем `delete` и значением 1. Вскоре вы увидите, что рядом с информацией о каждом сообщении в форме отображается флажок. Этот флажок входит в состав формы и служит для удаления сообщений. Как мы уже видели в **строке 35,** действие удаления начинается, когда скрипт "видит" в форме элемент с именем `delete` и значением 1, который мы и установили в **строке 50.**

```
52: my $c = 1;
```

**В строке 52** переменной $c присваивается значение 1. В последующем цикле эта переменная используется для отслеживания анализируемого сообщения. Но при первом взгляде на этот цикл непонятно, зачем нужен дополнительный счетчик. Неужели нельзя использовать `$count` или текущее значение $i? Все дело в удаляемых сообщениях. Когда этот скрипт проверяет электронную почту на сервере, он видит X сообщений, пронумерованных от A до X. Но когда вы пометите какие-то сообщения для удаления, скрипт удалит их, но не проверит заново текущий список почты на сервере. Почему? Ведь когда вызывается `Delete()`, сообщение только помечается как удаленное, а не удаляется на самом деле. Оно действительно удаляется, когда скрипт отключается от сервера. Поэтому не имеет особого смысла подключаться к серверу, проверять почту, делать удаление и проверять сервер снова. Это объяснение выглядит несколько запутанным, поэтому приведем небольшой пример.

Допустим, что на нашем сервере лежит пять сообщений, и вы пытаетесь удалить 2-е и 4-е сообщения. Так как один и тот же скрипт выполняет как отображение списка сообщений, так и их удаление, нам нужно, чтобы, когда происходит удаление, эти сообщения перенумеровывались заново. Когда вызывается скрипт `index.cgi`, он сначала получает с сервера список и текущее количество сообщений. В данном случае количество будет равно 5. Затем мы удаляем сообщения 2 и 4 методом `Delete()`. Как мы уже говорили, фактического удаления не происходит, но два эти сообщения теперь уже помечены как удаленные. Со всех практических точек зрения можно считать их удаленными. Затем мы снова отображаем список сообщений в броузере, но это уже должен быть новый список, без удаленных сообщений 2 и 4. Это означает, что сообщения 3 и 5 должны быть сдвинуты в отображаемом списке сообщений вверх, чтобы отразить состояние фактического списка, которое устанавливается при закрытии подключения и фактическом удалении сообщений с сервера. Нумерация сообщений на каждом из трех этих этапов иллюстрируется в табл. **10.1.**

**Таблица 10.1. Ход удаления сообщений**

| Первоначально | В ходе сеанса | После отключения от сервера |
|---|---|---|
| Сообщение 1 | Сообщение 1 | Сообщение 1 |
| Сообщение 2 | УДАЛЕНО | Сообщение 2 (было 3) |
| Сообщение 3 | Сообщение 3 | Сообщение 3 (было 5) |
| Сообщение 4 | УДАЛЕНО | |
| Сообщение 5 | Сообщение 5 | |

Как много времени нам пришлось потратить на объяснение инициализации одного-единственного скаляра! Теперь мы можем перейти к описанному выше циклу и ко всем его действиям.

```
53:   for my $i (1..$count-1) {
```

**Строка** 53 начинает цикл. Мы инициализируем переменную $i, которая будет счетчиком цикла. Значение $i будет номером сообщения, по которому его можно будет найти.

```
54:     next if exists $deleted{$i};
```

**Строка 54** проверяет, есть ли в хэше %deleted элемент с номером текущего сообщения. Как можно вспомнить из **строки 36,** в которой удалялось сообщение, в хэш %deleted записаны номера удаленных сообщений. Если такой номер совпадает с номером текущего сообщения, мы знаем, что оно удалено и не должно отображаться на экране, поэтому сразу переходим к следующему сообщению в цикле.

```
55:     my @head;
```

**В строке 55** инициализируется массив @head. Для чего он используется, вы вскоре увидите.

```
56:     foreach($pop->Head($i)) {
```

**Строка 56** — начало вложенного цикла, в котором из сообщения методом MIME::POP3Client *Head()* извлекаются элементы заголовка. Как можно видеть, в этот метод передается $i, поэтому заголовок извлекается из текущего сообщения. Цикл продолжается.

```
57:       if(s/^From:\s+//i) {
             $head[0] = qq(<td align=center>
             <input type="checkbox" name="mess" value="$c">
             </td>\n<td align=center>
             <a href="pop-view.cgi?mess=$c">$_</a></td>\n);
          }
58:       if (s/^Subject:\s+//i) {
             $head[1] = qq(<td align=center>
             <a href="pop-view.cgi?mess=$c">$_</a></td>\n);
          }
59:       if (/^Date:\s+/i && $_ =~ s/^Date:\s+(.*?)
             (?:\s+[+-]\d+(?:\s+.*?)?)?$/$1/i) {
             $head[2] = qq(<td align=center>$_</td><tr>\n);
60:     } }
```

**В строках 57—60** в цикле анализируются элементы заголовка. В данном примере нас интересуют элементы From, Subject и Data. В **строках** 57 **и** 58 с помощью подстановки проверяется, соответствует ли текущая строка заголовка сообщения (хранящаяся в $_) образцу данного элемента (т.е. формату отправителя и темы сообщения). Если подстановка выполняется успешно, полученный текст отображается. **В**

**строке 59** строка даты проверяется на соответствие образцу. Если такое соответствие наблюдается, выполняется необходимая подстановка. Эти три элемента заголовка могут выглядеть, например, так:

```
From: "Мельтцер, Кевин" <kevin@xxxxx.xxx>
Subject: Это тема сообщения!
Date: Mon, 10 Jan 2000 10:01:02 -0500
```

При отображении списка сообщений в броузере название строки заголовка (т.е. From:, Subject:) выводить не нужно. Для этого мы исключаем из строк слова "From", "Subject" и "Date". Следующий этап обработки строки — генерация кода HTML для вывода этих данных в одной строке таблицы, состоящей из четырех ячеек. Перед первой ячейкой дополнительно добавляется флажок, предназначенный для удаления этого сообщения. Код HTML в соответствующем порядке записывается в элементы массива @head. В нулевом элементе этого массива будут данные отправителя, в первом — тема сообщения и во втором — дата. Этот массив необходим потому, что некоторые почтовые клиенты отправляют сообщения с элементами заголовка, расположенными в другом порядке. Этот неприятный факт мы обнаружили при тестировании скрипта. В следующем скрипте pop-view.cgi будет показано, как анализировать заголовок сообщения с помощью MIME: : Parser, но мы посчитали, что важно будет также показать вам, как это делается вручную.

К этому времени цикл уже обработал заголовок, и массив @head должен выглядеть примерно так:

```
@head = ('"<td align=center><input type="checkbox" name="mess"
  value="l"></td>\n<td align=center><a href="pop-
  view.cgi?mess=1">Мельтцер, Кевин" <kevin@xxxxx.xxxx/a>
  </td>\n"',
  '"<td align=centerxa href="pop-view.cgi?mess=1">Это
  тема сообщения!</a></td>\n"',
  '"<td align=center>Mon, 10 Jan 2000 10:01:02</td><tr>\n'"
);
```

Затем этот внутренний цикл ненадолго завершается и мы возвращаемся во внешний цикл.

```
61:     $c++;
```

**В строке 61** происходит приращение счетчика, т.е. переход к номеру следующего сообщения. Как можно было заметить в **строках 57—60,** эта переменная служит для генерирования гиперссылок на скрипты просмотра и удаления сообщений.

```
62:     print @head;
63: }
```

**Строка 62** выводит в клиент данные заголовка последнего сообщения. Затем внешний цикл переходит к следующему элементу. **Строка 63** завершает внешний цикл.

```
64: $c--;
```

**В строке 64** значение $c уменьшается из-за особенностей поведения предшествующего цикла. Когда этот цикл приходит к завершению, приращение $c происходит еще раз, и поэтому мы должны скомпенсировать его, чтобы получить правильное значение счетчика.

```
65:  print td({-colspan=>4,-align=>'LEFT'},
66:            submit(-value=>'Удалить') . " В вашем
               почтовом ящике находится <b>$c</b> сообщений"),
67:  end_form,
68:  end_table,
69:  end_html;
```

**Строки 65—69—** один большой оператор `print()`, который разбит на несколько строк для наглядности. В нем сначала выводится еще одна ячейка таблицы (<TD>), В которой в **строке 65** помещается кнопка отправки формы и некоторый текст. Если щелкнуть на этой кнопке, все номера сообщений будут переданы в скрипт для удаления. Эти номера определяются из значений флажков, которые были установлены. В **строках 66,** 67 и 68 выводятся закрывающие дескрипторы формы, таблицы и HTML соответственно. Теперь все данные отображены у пользователя.

```
70:  $p.op->Close;
```

**Строка 70** закрывает подключение к серверу POP. При этом серверу неявно направляется команда QUIT. В этот момент все сообщения, помеченные как удаленные, удаляются физически и сокет закрывается.

```
71:  sub error {
72:    my $err = shift;
73:    print h3("Произошла ошибка при подключении к
                 серверу($err). Попробуйте подключиться
                 снова"),p,
74:    a({-href=>'index.cgi'},"Назад");
75:    exit;
76:  }
```

**Строки 71—76** — подпрограмма сообщения об ошибке. Она вызывается в строке 32, если подключиться к серверу POP не удается. Эта подпрограмма сообщает пользователю о любом сбое при подключении, используя ответ сервера. После вывода сообщения отображается также ссылка для возврата в скрипт `index.cgi`. Затем выдается команда `exit`, вследствие чего скрипт останавливается и не пытается отображать отсутствующие сообщения. На рис. 10.2 показан пример этой ошибки, когда скрипт не может соединиться с сервером POP.

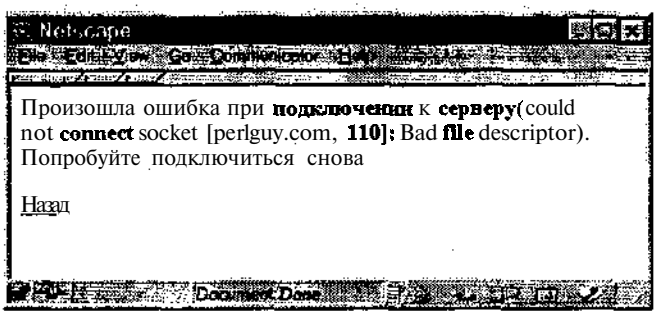



*Рис. 10.2. Экран при неудачном подключении к серверу POP*

Теперь у нас есть основа для приложения, которая позволяет входить на сервер POP и показывать сообщения электронной почты на этом сервере. Мы также имеем возможность удалять ненужные сообщения. На рис. 10.3 показан примерный вид экрана при отображении сообщений.

Как видно из этого рисунка, мы получили изрядное количество спама! Также обратите внимание, что мы уже установили флажок удаления на одном из сообщений (сообщении 7), которое, скорее всего, относится к спаму. Мы хотим избавиться от этого сообщения, и, так как оно уже отмечено и готово к удалению, мы сейчас это и сделаем. На рис. 10.4 показан экран после удаления сообщения 7. Вспомните также, что при этом происходит неявно. Как мы говорили, эта страница отображает не фактическую нумерацию сообщений в данный момент, а ту, которая установится, когда помеченное сообщение (сообщение 7 в данный момент) будет физически удалено.

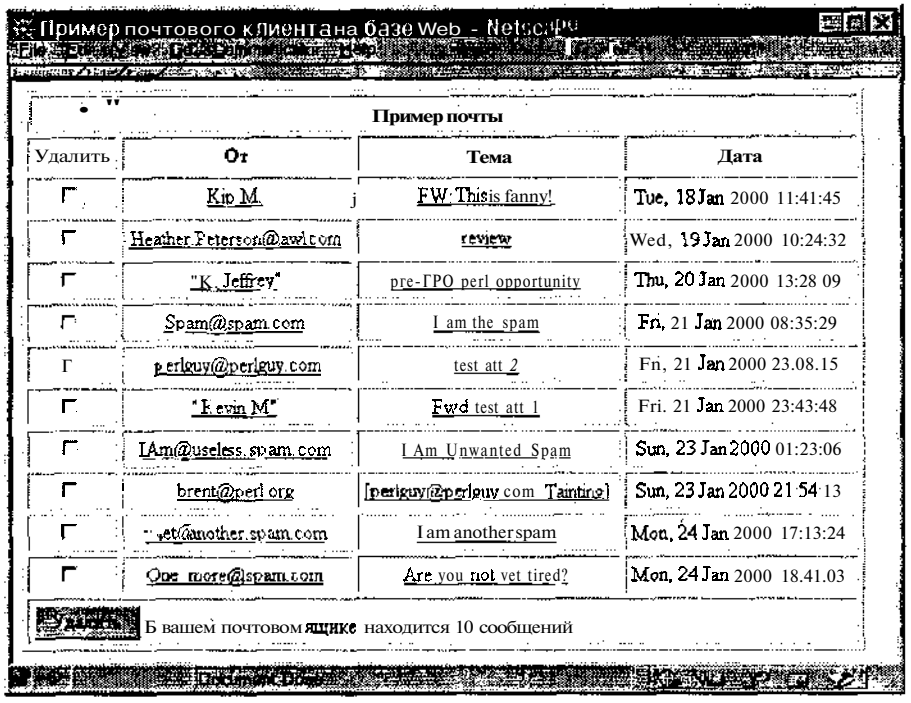



*Рис. 10.3, Экран при проверке электронной почты*

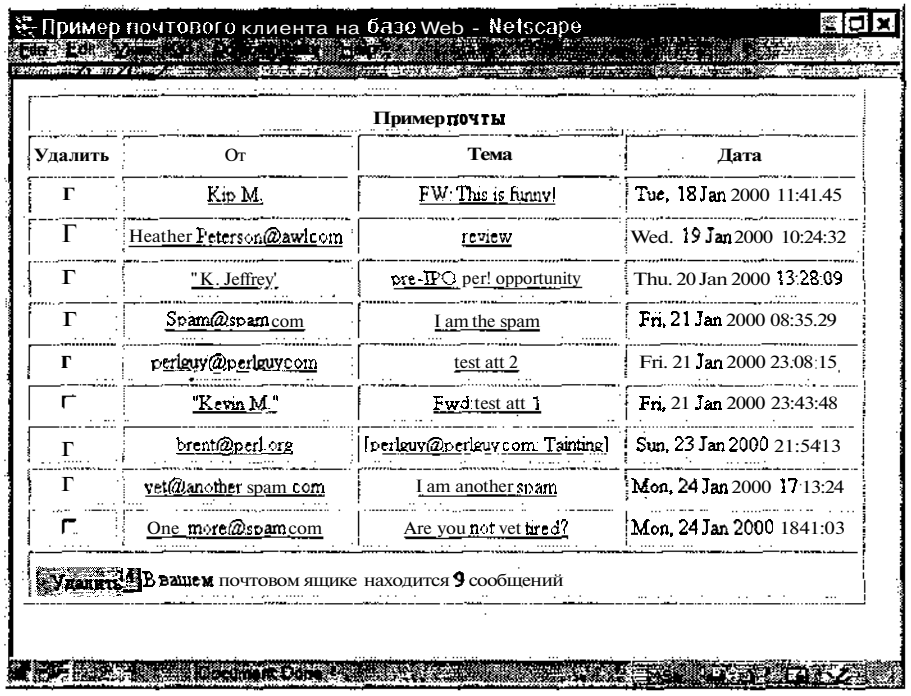



*Рис. 10.4. Спам удален!*

# Пример: чтение электронной почты через Web

Все пока идет хорошо, но нам еще нужно прочесть полученную почту. Мы уже можем просматривать список сообщений и удалять их, но от электронной почты гораздо больше пользы, когда ее можно читать! Как вы заметили, поля От и Тема на экране имеют ссылки на скрипт `pop-view.cgi`. Для сообщения 5, например, эта ссылка выглядит так.

```
<a href="pop-view.cgi?mess=5">
```

Давайте посмотрим на этот скрипт.

```
01:  #!/usr/bin/perl -wT
02:  # pop-view.cgi
03:  $|=1;
04:  use strict;
05:  use Mail::POP3Client;
06:  use CGI qw(:standard);
07:  use CGI::Carp qw(fatalsToBrowser)
08:  use MIME::Parser;
09:  use URI::Escape;
10:  use Untaint;
```

**Строки 1—10** — традиционное вступление.

```
11:  my $cookie = cookie('WebMail') || 0;
12:  my ($user, $pass, $server);
13:  if (!$cookie) {
14:    print redirect('pop-login.html"); exit;
15:  } else {
16:    print header;
17:    ($user, $pass, Sserver) = split(" ",$cookie);
18:  }
```

**Строки 11—18** очень похожи на **строки 9—24** в скрипте `index.cgi`. Мы проверяем, существует ли cookie WebMail. Если это не так, пользователь направляется на страницу входа. Если же такое cookie есть, скрипт продолжается.

```
19:  my $message = param('mess');
```

**Строка 19** методом `CGI.pm` *`param()`* получает номер сообщения, который передается в URL при запросе на Web-сервер из формы. Как можно вспомнить, ссылки на этот скрипт включают пару имя-значение, причем в данном случае имя — mess, а значение — номер сообщения. По этому значению мы будем выбирать определенное сообщение на сервере POP.

```
20:  Sserver = untaint(qr(.*), Sserver);
21:  $user = untaint (qr(^\w{1,8}$), $user);
22:  $pass = untaint(qr(^\w{1,8}$), Spass);
```

В **строках** 20, 21 и 22 выполняется очистка. Обратите внимание, что мы задали здесь для соответствия очень нестрогий образец. В этой ситуации у пользователя должно быть правильное cookie, установленное в `index.cgi`, но лучше проявить чуть больше подозрительности и сделать образцы построже. Теперь наши данные очищены и мы можем продолжать.

```
23:  my $pop = new Mail::POP3Client(HOST => $server);
24:  $pop->User($user);
```

```
25:  $pop->Pass($pass);
26:  $pop->Connect || error($pop->Message);
```

**В строках 23—26** создается новый объект Mail: : POP3Client, в него передается информация подключения и устанавливается подключение. Эти действия точно такие же, как и в предыдущем скрипте index.cgi.

```
27:  my Sparser = new MIME::Parser;
```

**Строка 27** создает новый объект MIME: : Parser в переменной Sparser. Этот объект поможет нам обращаться к методам MIME: : Parser при анализе сообщения.

```
28:  if ('(-d "./$user-$server")) {
29:  system('/bin/mkdir", "-m","0777", "$user-$server"); # UNIX
29:  system("mkdir", "$user-$server"); #Windows
30:  }
```

**В строках 28—30** создается каталог для хранения файлов, в которые MIME: : Parser будет выводить свои результаты. Помните **строку** 28 в index.cgi, где мы удаляли все содержимое этого каталога? А здесь мы создаем его, если этот каталог не существует. Существование каталога проверяется оператором -d. Если каталог не найден, он создается безопасным вызовом *system()* (см. главу 2 по безопасному применению *system()* и разрешениям Unix). Когда MIME: : Parser анализирует сообщение, он создает файлы, например, файл для тела сообщения и по файлу для каждого вложения. Если один и тот же скрипт применяется для нескольких пользователей или для нескольких учетных записей одного пользователя, объединение идентификатора пользователя с именем сервера помогает избежать возможной записи сообщений для двух пользователей в один и тот же каталог. Сочетание имени пользователя и имени сервера будет уникальным, так как на каждом сервере может быть только один пользователь с определенным именем.

```
31:  $parser->output_dir("./$user-$server");
```

**Строка 31** сообщает объекту MIME:: Parser, что все выходные данные будут помещены в указанный каталог. Обычно файлы, создаваемые MIME: : Parser, после использования удаляются, но вложения не удаляются. Мы не будем удалять файлы вложений, пока пользователь не оставит страницу (и не вернется в index.cgi, который и производит удаление), на тот случай, если пользователь захочет просмотреть файл вложения..

```
32:  my $mail = $pop->Retrieve($message);
```

В **строке 32** методом Mail: : POP3Client: : Retrieve (здесь Spop — это объект Mail: : POP3Client, который имеет метод *Retrieve ())* с сервера загружается нужное сообщение. Численное значение Smessage, которое мы получили из параметра mess формы, указывает, какое сообщение надо получить с сервера. Затем весь текст сообщения, включая заголовок, тело и закодированные приложения, записывается в переменную $mail.

```
33:  my Sentity = $parser->parse_data($mail);
```

Строка **33** неявно выполняет некоторые важные действия. Давайте проанализируем эту строку справа налево. Как можно заметить, скаляр Smail (содержащий полный текст сообщения) передается как параметр в метод *parse_data()*. Метод *parse_data()* входит в состав объекта $parser (MIME: : Parser). Фактически это метод базового класса MIME: : ParserBase, подклассом которого является MIME: : Parser, но сейчас нас не должны заботить такие подробности. Он возвращает и сохраняет в переменной $entity объект MIME: :Entity— одиночный объект сообщения или объект сложного сообщения с произвольной вложенностью. Собственно говоря, мы получаем в Sentity структуру данных, из которой можно легко выделить и использовать составные части сообщения.

```
34:  my $head = $entity->head;
```

В **строке** 34 из объекта `$entity` извлекается заголовок. Теперь в переменной `$head` содержится структура данных, подобная объекту `$entity`, но являющаяся его подмножеством. Это объект, заполненный информацией заголовка сообщения.

```
35:  print qq(Кому: ) . text_to_html($head->get('To',0));
36:  print qq(Копия: ) . text_to_html($head->get('Cc',0))
       if $head->get('Cc',0);
37:  print qq(Тема: ) . text_to_html ($head->get('Subject',0));
38:  print qq(От: ) . text_to_html($head->get('From',0));
39:  print qq(Дата: ) . $head->get('Date',0) . qq(<br>\n);
```

**В строках 35—39** обрабатываются элементы заголовка сообщения[3]. Значения выдаются методом *get ()* объекта $head (это объект Mail: `:Header`) и выводятся в клиент. Каждый элемент дополнительно обрабатывается подпрограммой *text_to_html()*, которую мы рассмотрим ниже в этой главе.

```
40:  my $parts = $entity->parts;
```

**В строке 40** определяется количество частей в объекте сообщения. Если это сообщение не имеет никаких частей, т.е. вложений, значение `$parts` будет 0.

```
41:  print p;
```

**Строка 41** просто выводит дескриптор HTML <P> одноименным методом CGI.pm. Поскольку мы импортировали все стандартные методы CGI.pm, сюда вошло множество методов генерирования HTML, включая метод *p ()*, который выводит этот дескриптор.

```
42:  if (!$parts) {
43:   print text_to_html ($entity->body_as_string);
44:    $entity->purge;
```

**В строках 42—44** проверяется, обнаружены ли в **строке 40** какие-то части сообщения. Если никаких частей нет (значение `$parts` — не истина), то мы получаем методом *body_as_string()* нашего объекта `$entity` тело сообщения в виде строки и выводим его. Дополнительно эта строка обрабатывается подпрограммой *text_to_html()*, которую мы рассмотрим ниже. Наконец, в **строке 44** все внешние части тела сообщения (на диске) удаляются методом `purge` *()*. Как было сказано выше, когда `MIME::Parser` анализирует сообщение, он разбивает его на отдельные файлы для тела сообщения и остальных частей (например, вложений). При вызове `purge` *()* эти файлы удаляются. Так как эти строки находятся в условном операторе, который выполняется, если сообщение состоит только из тела, то удаляется только оно.

```
45:  } else {
46:    print text_to_html (
         $entity->parts(0)->bodyhandle->as_string);
```

**В строках 45 и 46** начинается вторая часть условного оператора. Она выполняется, если в сообщении обнаружено несколько частей. Как и **строка 43, строка 46** выделяет часть тела объекта сообщения и передает ее в подпрограмму *text_to_html()* для последующего вывода. Так как в метод *parts ()* передается число 0, то извлекается нулевой элемент сообщения, т.е. его тело.

---

[3] *Целесообразно добавить функцию автоматического определения кодировки сообщений, для чего следует извлекать из заголовка элемент 'Content-Type', а из него - значение атрибута 'charset', которое устанавливать в заголовке страницы (естественно, для этого придется изменить структуру программы) — Прим. ред.*

```
47:     print h3("Вложения: ");
```

**Строка 47** просто выводит заголовок уровня 3 с текстом "Вложения".

```
48:   for (my $i=1; $i<$entity->parts;$i++) {
```

**В строке 48** начинается цикл, который обходит все части объекта сообщения. Как можно видеть, цикл начинается со значения счетчика `$i` 1, так как нулевая часть — тело сообщения, которое мы только вывели.

```
49:       (my $temp_file_name =
          $entity->parts($i)->bodyhandle->path)
          =~ s!^.*$user-$server/!!;
50:       my $un_file = un_escape($temp_file_name);
```

**В строках 49 и 50** вводится "временный" файл — фактически файл вложения. **Строка 49** получает имя файла вложения, исключая все символы, кроме имени файла, из строки, полученной выражением *$entity->parts ($i) ->bodyhandle->path*, которая содержит путь к файлу вложения на сервере. По правилам, принятым в наших примерах, пути для всех файлов вложений имеют вид *./пользователь-сервер/вложение*.ext, что и является значением *$entity->parts ($±) ->bodyhandle->path()*. Мы должны исключить отсюда часть *./пользователь-сервер/*. Почему? Поскольку в следующих строках мы будем отображать это имя файла в клиенте, желательно, чтобы оно не содержало лишней информации. Кроме того, если файл вложения — это изображение, нам нужно будет показать его имя в подписи. **В строке 50** имя вложения кодируется в URI, так как мы будем делать гиперссылку на него, а кодировка URI нужна, чтобы можно было обработать пробелы и нестандартные символы. Имя файла вложения записывается в `$temp_file_name`, а это же имя в кодировке **URI** — в `$uri_file`.

```
51:       if ($temp_file_name=~/\. (gif|jpg|png)$/i) {
52:         print qq(Файл: img src=
            "$user-$server/$temp_file_name">);
```

**В строках 51 и 52** мы проверяем, содержит ли имя файла вложения `$temp_file_name` расширение графических файлов GIF, JPG или PNG. Если вложение — графический файл одного из этих форматов, в **строке** 52 выводится код HTML, отображающий этот файл в броузере. Так как броузер всегда может обрабатывать файлы этих типов, имеет смысл просто показать эти изображения, а не предлагать пользователю сначала загрузить их.

```
53:       } else {
54:         print qq(Файл: <a href="view_att.cgi?att=$uri_file">
            $temp_file_name</a><BR>);
```

**Строки 53 и 54** выполняются, если вложение не является графическим файлом. **В строке 54** выводится **HTML** для ссылки на вложение. Заметьте, что в этой ссылке вызывается скрипт `view_att.cgi` и в него как параметр `att` передается `$uri_file`. Скрипт `view_att.cgi`, который отображает вложение, мы рассмотрим ниже в этой главе. Вы поймете, почему мы создаем отдельный скрипт для показа вложений, а не просто даем ссылку на файл вложения.

```
55:       }
56:     }
57:   }
```

**Строки 55—57** закрывают условный оператор (проверку, является ли вложение графическим файлом), цикл (обход частей сообщения) и внешний условный оператор, который проверяет, есть ли в этом сообщении другие части, кроме тела.

```
58:  print p(-{-align=>'center'},
59:  a({-href=>"index.cgi?delete=1&mess=$message"},
       "Удалить сообщение"), " | ",
60:  a({-href=>index.cgi"}, "Вернуться к почте"), " i ",
61:  a({-href=>"pop-logout.cgi"}, "Выход"));
```

**В строках 58—61** выводится информация "нижнего колонтитула". В данном случае он включает три ссылки: для удаления сообщения, для возврата к списку сообщений и для выхода из почтовой системы.

```
62:  $pop->Close;
```

**Строка 62** закрывает подключение к серверу **POP.**

```
63:  sub text_to_html {
64:    my $raw = shift;
```

**В строках 63 и 64** начинается подпрограмма *text_to_html* Эта подпрограмма выполняет над переданной в нее строкой, которая сохраняется в переменной $raw, четыре действия.

```
65:      $raw =~ s!([<>])!($1 eq '<')? "&lt;": "&gt\;"!eg;
```

**Строка 65** заменяет все символы "больше" (>) соответствующим кодом **HTML** sgt;, а все символы "меньше" (>) — кодом <. Это можно выполнить в двух строках, но мы объединили оба эти действия в одном регулярном выражении. В левой части подстановки указан символьный класс, состоящий из двух символов — "меньше" и "больше". Этот символьный класс заключается в круглые скобки, так что можно обращаться к совпадающему символу в правой части подстановки. Далее, в правой части мы имеем условное выражение. Если значение переменной $1, которая содержит совпадающий символ из левой части, равно символу "меньше", то это значение заменяется его соответствием в коде HTML— **<**. Если же это условие не выполняется, то переменная содержит символ "больше", и он заменяется на другой код, **sgt;.** Модификатор !e в конце этой строки позволяет вычислять условное выражение в правой части подстановки. Эта строка нужна для того, чтобы символы "больше" и "меньше", которые могут встречаться в сообщении, не рассматривались как часть дескрипторов HTML и правильно отображались в броузере.

```
66:      $raw =~ s!  ((ht|f)tps?://)([\w-]*)((\..[^\s]*)+)!
           <a href="$1$3$4" target="external">$1$3$4</a>!g;
```

**Строка** *66* переводит все вхождения ссылок http, https и ftp в тексте сообщения в гиперссылки. Поскольку в этом регулярном выражении есть несколько символов /, для наглядности мы использовали в качестве разделителя частей подстановки восклицательный знак. Это регулярное выражение начинается с поиска текста http://, https:// или ftp://. Если какой-то из этих образцов находит соответствие, найденный текст сохраняется в переменной $1. Это результат выполнения внешней пары круглых скобок. Внутренняя пара круглых скобок ((ht|f)) дает соответствие ht или f, и это значение сохраняется в $2, хотя использовать его мы не будем. Следующая пара круглых скобок — символьный класс, соответствующий любым буквам и цифрам, а также дефису. Это обычные и допустимые символы для доменного имени. Весь найденный текст вплоть до первой точки сохраняется в $з. Последняя пара круглых скобок задает соответствие точке и любому ненулевому количеству любых символов, отличных от пробела. Найденное соответствие сохраняется в $4. Как можно видеть, в правой части подстановки на базе этих значений создается гиперссылка. Всю эту механику довольно трудно усвоить сразу, особенно если вы не очень хорошо разбираетесь в регулярных выражениях, поэтому мы подробно проанализируем, что происходит в этой строке.

Для примера разберем текстовую строку "http://www.geekstuff.com - тенниски с эмблемой Perl Mongers". Первой паре круглых скобок ((ht|f)tps?://) соответствует текст "http://", который сохраняется в $1. Поскольку соответствие внутренним круглым скобкам записывается в специальной переменной со следующим номером, в $2 будет сохранена строка "ht". Пока все идет прекрасно. Следующий образец ([\w-]*) ищет все буквенно-цифровые символы (заданные выражением \w) и дефисы. В найденное соответствие записываются все последующие символы, пока не встретится символ, не принадлежащий к этому символьному классу, например, точка. В данном случае это соответствие будет равно "www" и будет сохранено в $3. Последний образец ((\.[^\s]*)+) находит в строке точку, за которой следует некоторое количество любых символов, отличных от пробела. Знак + задает многократное повторение поиска по этому образцу. В данной строке для этого образца будут найдены два соответствия: ".geekstuff" и ".com". Последнее значение сохраняется в $4, и анализ текста на этом завершается.

```
67:     $raw =~ s!([\w\-.]+)\@([\w-]+)((\.[\w-]+)*)!
          <a href=\"pop-compose.cgi?to=$1\@$2$3\">$1\@$2$3</a>!g;
```

**Строка 67** напоминает **строку** 66 с тем отличием, что в ней выявляется в тексте сообщения и преобразуется в гиперссылку mailto адрес электронной почты[4]. В правой части подстановки мы ищем любые буквы и цифры, а также дефис, которые встречаются любое количество раз вплоть до вхождения символа @. Найденный текст соответствует первой части (имени пользователя) адреса электронной почты и сохраняется в $1. Две следующие пары круглых скобок после @ задают соответствие для остальной части адреса. Как и в **строке 66,** первый образец находит последовательность букв, цифр и дефисов после символа @ и перед точкой, и найденный текст сохраняется в $2. Затем в строке отыскиваются все последовательности "точка, буквы или цифры или дефис". Результат сохраняется в $3. Все! Но возникает вопрос — почему третий образец в **строке 66** и в этой строке не один и тот же? В конце концов, доменное имя URL и адреса электронной почты должны совпадать, не так ли? Не совсем так. Допустим, некто послал вам письмо, содержащее URL со строкой запроса; например, http://search.cpan.org/search?mode=module&query=Apache. Ясно видно, что текст после http://search.cpan.org/search содержит символы, которые не являются ни буквами, ни цифрами, и, установив образец, как в этой строке, мы не получим желаемого результата. Напротив, адрес электронной почты не будет содержать таких символов, поэтому мы задаем для образца только буквы, цифры и дефис, а не любые символы вплоть до пробела.

```
68:     $raw =~ s/\n/<bi->/g;
69:     return $raw;
70: }
```

**В строках 68—70** вначале все концы строк заменяются дескрипторами HTML <BR>. Так как в Web-клиентах не предусмотрен символ конца строки, мы должны сделать так, чтобы строки прерывались там, где нужно. Затем в **строке 69** мы возвращаем из подпрограммы сообщение, преобразованное в **HTML,** и в **строке 70** завершаем саму подпрограмму *text_to_html()*. Любой текст, переданный в эту подпрограмму, теперь будет красиво выглядеть в броузере.

```
71:  sub error {
72:    my $err = shift;
73:    print h3("Произошла ошибка при подключении к
                 серверу($err). Попробуйте подключиться
```

[4] *Это решение подходит для большинства адресов электронной почты.*

```
                снова"),p,
74:     a({-href=>'index.cgi'},"Назад");
75:     exit;
76:   }
```

**Строки 71–76** завершают скрипт. Эта подпрограмма полностью повторяет подпрограмму `error()` из `index.cgi`. Вы можете подумать; "Почему бы не выделить эту подпрограмму в модуль и не использовать ее для всех скриптов?" Не хочу портить сюрприза, но именно это я хотел предложить вам в упражнениях в конце главы. Результат работы скрипта показан на рис. 10.5.

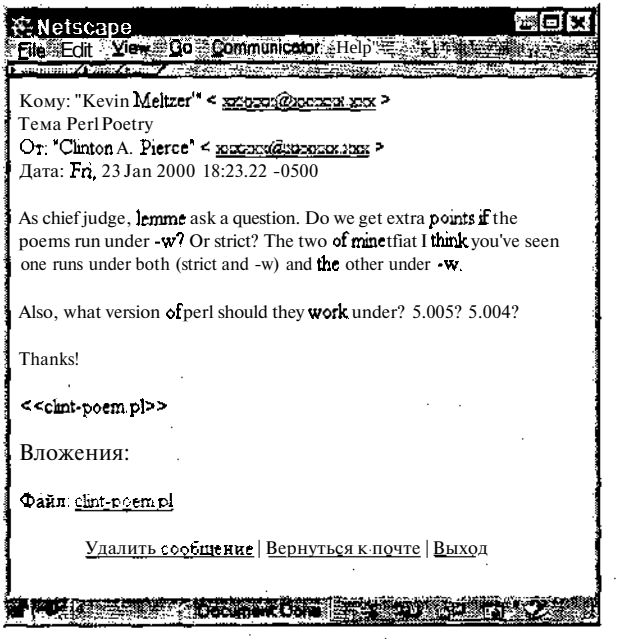



*Рис. 10.5. Просмотр сообщения*

Попробуем подвести итог. Теперь мы имеем возможность входить на сервер POP, забирать с него сообщения и отображать их список. Кроме того, мы можем удалять сообщения, читать их, а также просматривать вложения. Но на самом деле это еще не так. Мы можем увидеть, что в сообщении имеется вложение, и сделать ссылку для его просмотра, но сам просмотр нам пока недоступен. Поэтому сейчас мы займемся этой функцией.

# Пример: показ вложений

Скрипт `view_att.cgi`, приведенный в листинге 10.2, передает вложение в клиент. Этот скрипт невелик и в основном не требует пояснений. Он принимает как параметр имя файла вложения, затем, как и другие скрипты, проверяет информацию cookie. Если cookie не существует, то пользователь, опять же, как в других скриптах, направляется на страницу входа. особое внимание здесь следует обратить на **строку 7** и **строки 11–17.**

### Листинг 10.2. Скрипт для показа вложений

```
01: #!/usr/bin/perl -wT
02: # view_att.cgi
03: use CGI qw(:cgi);
```

```
04:  use strict;
05:  use URI::Escape;
06:  use File::Basename;
07:  my $id = basename(param('att'));
08:  my $cookie = cookie('WebMail') || 0;
09:  #if ($cookie) {
10:    my ($user, $pass, Sserver) = split(" ",$cookie);
11:    print "Content-Type: application/x-unknown\n";
12:    print "Content-Disposition: attachment;
         filename=$id\n\n";
13:    open(FILE,"./$user-$server/$id");
14:      local $/ = undef;
15:     my $content = <FILE>;
16:    close FILE;
17:    return print $content;
18:  } else {
19:    print redirect("pop-login.html"); exit;
20:  }
```

**В строке 7** вводится одна мера предосторожности. Как можно увидеть далее, в следующих строках предполагается наличие временных файлов, которые MIME::Parser создает в пользовательском каталоге на Web-сервере. В этой строке вызывается метод *basename()*, импортированный из модуля File::Basename в **строке** 5. Этот метод возвращает в переменную $id только имя файла, исключая путь. Благодаря этому пользователь не сможет передать в этот скрипт относительный путь и загрузить какой-либо другой файл, который ему загружать не положено. Первейшая мера предосторожности состоит в том, что ни в коем случае нельзя автоматически выдавать клиенту любые файлы, которые он потребует в параметрах скрипта. В данном примере мы знаем, что от клиента должно быть получено только имя файла, так как каталог уже был определен ранее. Поэтому все, что кроме имени файла передается в скрипт, отбрасывается.

**В строке 11** в броузер передается заголовок, который указывает, что содержимое имеет неизвестный тип файла. В результате броузер открывает диалоговое окно, в котором запрашивает, что сделать с поступающими данными (сохранить, открыть и т.д.).

**Строка 12** сообщает броузеру, что поступает вложенный файл, а также имя этого файла. Когда Web-сервер передает клиенту любой файл, он передает в заголовке тип MIME, как бы говоря ему: "Я посылаю тебе файл, а это тип данных, которые он содержит". На основе этой информации клиент определяет, как он должен обработать поступающий файл. В данном случае мы сообщаем клиенту, что тип файла нам неизвестен. Если клиент не знает, какой тип у поступающего файла, то он не знает, как его обработать, и поэтому выводит окно, спрашивая пользователя, что делать. Здесь пользователь может выбрать, сохранить ли этот файл или открыть его в соответствующем приложении.

**В строках 13–16** файл открывается, все его содержимое считывается в скаляр (для этого разделителю записей *$/* присваивают неопределенное значение), затем дескриптор файла закрывается.

**Строка 17** выводит все содержимое файла в клиент.[5]

Теперь, когда вы видите, "как", позвольте мне объяснить, "зачем". Очень легко было бы срздать ссылку непосредственно на файл вложения и предоставить Web-серверу самому выполнять действия по ней. Однако по соображениям безопасности может понадобиться, чтобы временный каталог, с которым работает MIME::Parser, не входил в дерево каталогов Web-сервера. В данном примере мы для простоты отка-

[5] *В системах Unix текстовые и двоичные файлы не различаются. В других операционных системах надо указать нужный тип файла командой* `binmode()`.

зались от этой меры, но вы можете применить ее, заменив `./$user-$server` на что-нибудь вроде `/tmp/$user-$server` и слегка изменив регулярное выражение в `pop-view.cgi`. Опять обратимся к вопросу "почему". Во-первых, этот скрипт проверяет, предоставил ли клиент необходимое cookie. Если это не так, получить вложение невозможно. Также здесь предусматривается мера безопасности, описанная выше — вынос каталога за пределы Web-сервера. Если каталог находится не в дереве каталогов Web-сервера, попытка просто переадресовать броузер на файл командой `redirect()` не удастся. Но если файл открывается, как в этом скрипте, он может находиться *где угодно* на сервере (т.е. на машине, на которой работает Web-сервер) — все равно он будет открыт и его содержимое передано клиенту. Еще один хитрый прием позволяет гарантировать, что данный файл будет передан конкретному клиенту, которому разрешено его получить. Это происходит благодаря тому, что имя временного каталога определяется на основе данных cookie и состоит из имени учетной записи и имени сервера POP, т.е. является уникальным.

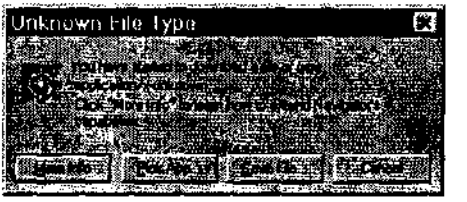



*Рис.10.6. Диалоговое окно для сохранения вложения*

Когда вложение передается в броузер, в нем появляется диалоговое окно, в котором пользователя спрашивают, что сделать с полученным файлом. Файл можно открыть или сохранить (рис. 10.6).

# Пример: создание сообщения электронной почты

Куда бы годился этот пример, если бы мы не показали вам, как составить сообщение? Пользы от него было бы немного. Поэтому в следующем разделе мы рассмотрим создание и отправку сообщения электронной почты. В листинге 10.3 показан скрипт `pop-compose.cgi`, который будет служить нам интерфейсом для создания и отправки сообщений.

Листинг 10.3. **`pop-compose.cgi`** — скрипт, отображающий форму для составления сообщения

```
01:  #!/usr/bin/perl -wT
02:  # pop-compose.cgi
03:  use strict;
04:  use CGI qw(:standard);
05:  use Untaint;
06:  my $cookie = cookie ('WebMail') || 0;
07:  my ($user, $pass, $server);
08:  if ($cookie) {
09:    print redirect("pop-login.html"); exit;
10:  } else {
11:    print header;
12:    ($user, $pass, $server) = split(" ",$cookie);
13:   }
14:  $server = untaint(qr(.*), $server);
15:  $user = untaint(qr(^\w{1,8}$), $user);
16:  $pass = untaint(qr(^\w{1,8}$), $pass);
17:  print start_html('Создание сообщения'),
18:    h1('Создание сообщения'),
```

```
19:     start_form({-method=>'POST',-action=>
          'pop-send.cgi'}),
20:     "Кому: ",textfield('to'),p,
21:     "Копия: ",textfield('cc'),p,
22:     "Тема: ",textfield('subject'),p,
23:     "Сообщение: ",p,textarea({-name=>'body',
          -rows=>9,-cols=>70,-wrap=>'soft'}),p,
24:     submit({-value=>'Отправить'}),
25:     end_form,
26:  hr;
```

Все, что делает этот скрипт, вам уже должно быть знакомо. Он проверяет cookie, как каждый скрипт в этой главе, и отображает форму, в которой пользователь будет вводить данные адресата, копии, темы и тела сообщения электронной почты. Эта форма отправляется в другой скрипт — `pop-send.cgi`, описание которого приводится ниже.

```
01:  #!/usr/bin/perl -wT
02:  # pop-send.cgi
03:  use strict;
04:  use CGI qw(:standard);
05:  use MIME::Parser;
06:  use Net::SMTP;
07:  use Untaint;
08:  use CGI::Carp qw(fatalsToBrowser);
09:  my $cookie = cookie ('WebMail') || 0;
10:  my ($user, $pass, $server);
11:  if (!$cookie) {
12:    print redirect("pop-login.html"); exit;
13:  } else {
14:    ($user, $pass, $server) = split(" ",$cookie);
15:  }
16:  $server = untaint(qr(.*), $server);
17:  $user = untaint(qr(^\w{1,8}$), $user);
18:  $pass = untaint(qr(^\w{1,8}$), $pass);
```

**Строки 1—18** содержат только один новый элемент. Все остальное вы уже видели в предыдущих примерах. В **строке 6** мы импортируем новый модуль— `Net::SMTP`. С помощью этого модуля мы будем отправлять электронную почту.

```
19:  ray  @to = spiit(/;\s+/,param('to'));
20:   my @cc = split(/;\s+/,param('cc'));
21:   my $subject = param('subject');
22:   my @body = split(/\n/,param('body'));
```

**В строках 19—22** производится получение ввода из формы. Данные полей to и cc разбиваются на элементы, разделенные последовательностью из точки с запятой и по крайней мере одного пробела, и записываются в массивы. В данном примере мы предполагаем, что в эти поля может быть введено по несколько адресов, разделенных точками с запятой и пробелами. Это правило, разумеется можно изменить — отдельные адреса можно разделять каким угодно символом или последовательностью символов. Поле `subject` (тема сообщения) разделять на элементы не надо, так как это — одна текстовая строка; она сохраняется в скаляре. И, наконец, в **строке 22** данные тела сообщения (body) разбиваются на отдельные строки по символам конца строки. Каждая строка тела сообщения записывается в отдельный элемент массива.

```
23:  my Stop = build MIME::Entity ('X-Mailer' => 'WebMail',
24:                                  -From => "$user\@$server",
25:                                 -To => \@to,
26:                                 -Cc => \@cc,
```

```
27:                                  -Subject => $subject,
28:                                   Data => \@body
29:   );
```

**В строках 23–29** методом *build()* формируется объект `MIME::Entity`.[6] Результат формирования в $top будет содержать всю необходимую информацию сообщения, готового к отправке! **Строка 23** определяет используемый почтовый клиент, которому мы даем название WebMail. **Строка 24** указывает, от кого передается это сообщение. Это реализовано здесь довольно простым способом: мы принимаем, что сообщение исходит от той же учетной записи и сервера, которые использовались для проверки электронной почты. Далее, **строка 25** передает в метод *build()* данные адресата сообщения — ссылку на массив `@to`, созданный в **строке 19. Строка 26** подобным образом устанавливает адресатов для получения копии сообщения. Тема сообщения указывается в **строке 27,** а **строка 28** передает в метод тело сообщения через ссылку на массив `@body`. Теперь наше сообщение должным образом отформатировано и готово к отправке.

```
30:   my $smtp = new Net::SMTP ('smtp.host.com');
```

**В строке 30** создается новый объект `Net::SMTP` И определяется сервер SMTP, который мы будем использовать. Если сервер SMTP располагается на той же машине, на которой работает данный скрипт, это значение можно изменить на `'localhost'`.

```
31:   $smtp->mail("$user\$server");
```

**Строка 31** сообщает объекту `Net::SMTP`, от кого исходит это сообщение.

```
32:   $smtp->to(@to, @cc);
```

**Строка 32** сообщает объекту, кому это сообщение должно быть направлено. Эти действия выглядят лишними, так как мы уже задали эту информацию в объекте `MIME::Entity`, но для объекта `Net::SMTP` это нужно снова, чтобы он мог установить связь с сервером SMTP.

```
33:   $smtp->data;
```

**Строка 33** вызывает метод *data(),* сообщая этим объекту `Net::SMTP`, что мы собираемся начать передачу сообщения.

```
34:   $smtp->datasend($top->stringify);
```

**В строке 34** наш объект `MIME::Entity` передается в объект SMTP как тело сообщения. Метод *stringify()* преобразует все данные объекта в строку.

```
35:   $smtp->dataend;
36:   $smtp->quit;
```

**В строке 35** вызывается метод *dataend()* для нашего объекта `Net::SMTP` (хотя фактически этот метод находится в `Net::Cmd`). Таким образом мы сообщаем объекту SMTP, что передача данных закончена. **Строка 36** заставляет объект `Net::SMTP` послать серверу SMTP команду QUIT И закрыть сокет. Можно сказать, что `Net::SMTP` сам устанавливает связь с сервером и выдает необходимые для передачи сообщения команды SMTP, избавляя вас от этой работы.

```
37:   print redirect('index.cgi');
38:   exit;
```

**Строка** 37 направляет броузер назад к списку сообщений, когда отправка сообщения будет закончена. Наконец, в **строке 38** мы завершаем скрипт командой *exit()*, чтобы гарантировать, что он уже не будет выполняться. См. снимок экрана на рис. 10.7.

---

[6] *Здесь также целесообразно задавать кодировку сообщения (возможно, выделив для ее указания дополнительный элемент в форме). Достаточно указать в объекте MIME::Entity еще одно поле, например "Charset:" => "koi8-r" — Прим. ред.*

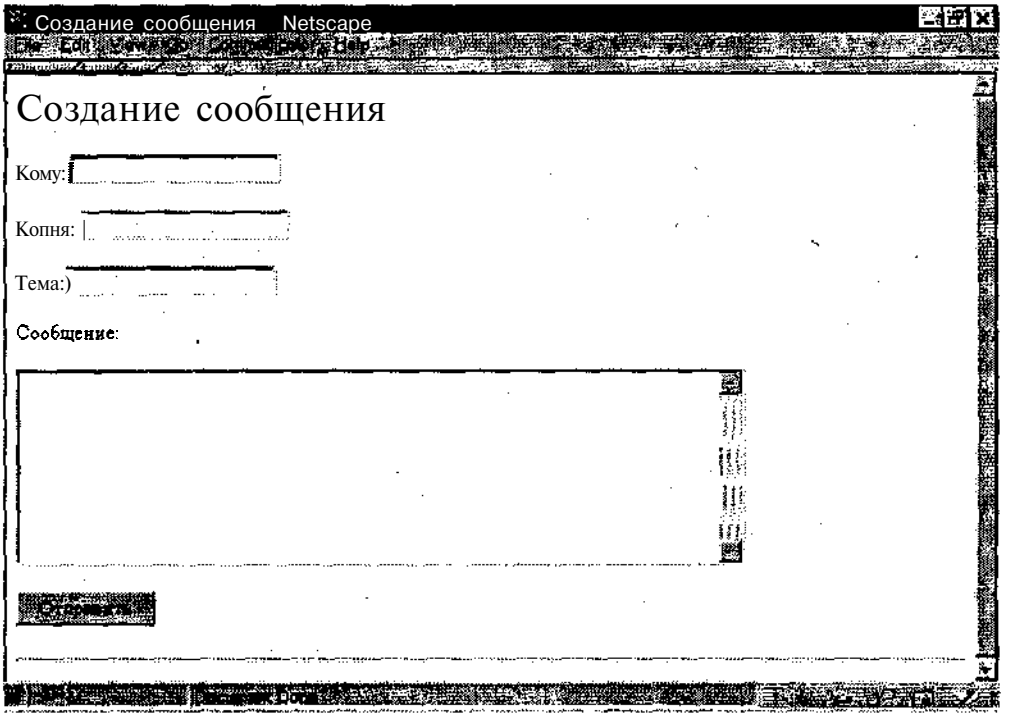



*Рис. 10.7. Форма для создания сообщения*

Все! Мы охватили в этой главе массу новой информации. Теперь наш пример почти закончен. Осталось сделать только одно дело — реализовать выход из системы. В любом Web-приложении, в котором пользователь должен регистрироваться, должна также быть и функция выхода. Она предусматривается главным образом по соображениям безопасности. Вы ведь не хотите, чтобы, пока вы ушли пообедать, кто-то другой мог подойти к вашему компьютеру и прочесть вашу почту или, хуже того, отправить сообщение от вашего имени!

В листинге 10.4 приводится небольшой скрипт `pop-logout.cgi`, который позволяет пользователю выйти из почтовой системы. Если пользователь вышел из системы, он не сможет проверять почту, пока снова не войдет.

### ЛИСТИНГ 10.4. Скрипт `pop-logout.cgi`

```
01:  #!/usr/bin/perl -wT
02:  # pop-logout.cgi
03:  use CGI qw(:standard) ;
04:  $cookie = cookie(-name => 'WebMail',
05:                   -value => 0
06:                  );
07:  print header(-cookie=>$cookie);
08:  print start_html(-title=>'Выход',
09:                   -BGCOLOR=>'white'
10:                  );
11:  print qq(Вы вышли из WebMail. Чтобы вернуться в
12:    систему, щелкните <A HREF="pop-login.html">здесь</A>.);
```

В этом скрипте нет ничего нового. Суть его в том, что мы устанавливаем новое значение cookie. Это значение — 0 — устанавливается в **строке 4.** Вследствие этого проверка cookie в других скриптах покажет, что данный пользователь не зарегистрирован в системе, следовательно, не может с ней работать. Так как это cookie является (и всегда было) временным, то, когда пользователь закрывает свой броузер (все процессы броузера), cookie исчезает. Это хорошая мера безопасности: если бы пользователь не вышел из системы должным образом и cookie было бы постоянным, кто-то другой мог бы просмотреть файл cookie и узнать имя пользователя и пароль. Если cookie временное, этого не случится!

# Упражнения

- Некоторые фрагменты кода, например, подпрограмма *error()* и подключение к серверу POP, встречаются в каждом скрипте. Попробуйте выделить эти функции и любые другие, которые вы посчитаете универсальными, в отдельный модуль, который смогут импортировать все скрипты.
- Скрипт для составления/отправки сообщений имеет один существенный недостаток: в нем нельзя отправлять вложения! Вспомните, чему вы научились из примеров главы 7, и добавьте эту функциональную возможность. Рекомендуем также прочитать документацию к MIME: `:Entity`.
- Еще один недостаток — отсутствие функции "Ответить" (Reply To). Однако все необходимое для того, что бы ее реализовать, можно найти в примерах данной главы. Вы знаете, как проанализировать заголовок, получить части тела сообщения, создать объект MIME и отправить его по почте. Для решения этой проблемы надо только объединить все вместе.

# Листинги

Листинг 10.5. Полный текст **index**.cgi

```
01:  #!/usr/bin/perl -wT
02:  # index.cgi
03:  $|=1;
04:  use strict;
05:  use Mail::POP3Client;
06:  use CGI qw(:standard :netscape *table);
07:  use CGI::Carp qw(fatalsToBrowser);
08:  use Untaint;
09: my $cookie = cookie('WebMail') || 0;
10: my $user = param('username') || 0;
11:  my $pass = param('password');
12:  my $server = param('server');
13:  if ((!$user || !$pass || !$server) && !$cookie) {
14:    print redirect('pop-login.html"); exit;
15:  }
16:  if (!$cookie) {
17:     $cookie = cookie(-name => 'WebMail',
18:                      -value => "$user $pass $server",
                          -path => '/'
19:     );
20:  print header(-cookie=>$cookie);
21:  } else {
```

```
22:    print header;
23:    ($user, $pass, $server) = split(" ",$cookie);
24: }
25:  $server = untaint(qr(.*), $server);
26:  $user = untaint(qr(^\w{1,8}$), $user);
27:   $pass = untaint(qr(^\w{1,8}$), $pass);
28:  system('/bin/rm","-rf","./$user-$server") unless
       $user !~ /^\w/;
29:  my $pop = new Mail::POP3Client(HOST => $server,
                                    AUTH_MODE => 'PASS');
30:  $pop->User($user);
31:  $pop->Pass($pass);
32:  $pop->Connect || error($pop->Message);
33:  my $count = $pop->Count;
34:  my %deleted;
35:  if (param('delete')) <
36:    ($pop->Delete($_) && $deleted{$_}++)
       foreach (param('mess'));
37: }
38:  print start_html ({-title=>
       'Пример почтового клиента на базе Web',
39:    -BGCOLOR=>'white'}),
40:  p({-align=>'center'},
41:    start_table({-border=>1,
42:                 -cellpadding=>4,
43:                -cellspacing=>4}) ,
44:   Tr,
45:    td({-align=>'CENTER',-colspan=>4},
         [strong("Примерпочты")]),
46:   Tr,
47:    th({-align=>'CENTER'}, ['Удалить', 'От',
                               'Тема','Дата']),
48:   Tr,
49:    start_form({-method=>'POST',-action=>'index.cgi'}),
50:    hidden({-name=>'delete',-value=>'1'}))
51: ;
52:  my $c = 1;
53:  for my Si (1..$count-1) {
54:    next if exists $deleted{$i};
55:    my @head;
56:    foreach($pop->Head($i)) {
57:      if(s/^From:\s+//i) {
           $head[0] = qq(<td align=center>
           <input type="checkbox" name="mess" value="$c">
           </td>\n<td align=center>
           <a href="pop-view.cgi?mess=$c">$_</a></td>\n);
         }
58:      if (s/^Subject:\s+//i) {
           $head[l] = qq(<td align=center>
           <a href="pop-view.cgi?mess=$c">$_</a></td>\n);
         }
59:       if (/"Date:\s+/i && $_ =~ s/^Date:\s+(.*?)
           (?:\s+[+-]\d+(?:\s+.*?)?)?$/$l/i) {
           $head[2] = qq(<td align=center>$_</td><tr>\n);
60:    } }
61:    $c++;
62:    print @head;
63: }
```

```
64:  $c--;
65:  print td({-colspan=>4,-align=>'LEFT'},
66:          submit(-value=>'Удалить') . " В вашем
               почтовом ящике находится <b>$c</b> сообщений"),
67:  end_form,
68:  end_table,
69:  end_html;
70:  $pop->Close;
71:  sub error {
72:    my $err = shift;
73:    print h3("Произошла ошибка при подключении к
                 серверу($err). Попробуйте подключиться
                 снова"),p,
74:    a({-href=>'index.cgi'},"Назад");
75:    exit;
76:  }
```

**ЛИСТИНГ 10.6. Полный текст pop-view.cgi**

```
01:  #!/usr/bin/perl -wT
02:  # pop-view.cgi
03:  $|=1;
04:  use strict;
05:  use Mail::POP3Client;
06:  use CGI qw(:standard);
07:  use CGI::Carp qw(fatalsToBrowser)
08:  use MIME::Parser;
09:  use URI::Escape;
10:  use Untaint;
11:  my $cookie = cookie('WebMail') || 0;
12:  my (Suser, $pass, Sserver);
13:  if (!$cookie) {
14:    print redirect('pop-login.html"); exit;
15:  } else {
16:    print header;
17:    (Suser, $pass, $server} = split(" ",$cookie);
18:  }
19:  my Smessage = param('mess');
20:  Sserver = untaint(qr(.*}, $server);
21:  $user = untaint(qr(^\w{1,8}$), Suser);
22:  $pass = untaint(qr(^\w{1,8}$), Spass);
23:  my $pop = new Mail::POP3Client(HOST => Sserver);
24:  $pop->User($user);
25:  $pop->Pass($pass);
26:  $pop->Connect || error($pop->Message);
27:  my Sparser = new MIME::Parser;
28:  if ('(-d "./$user-$server")) {
29:  system('/bin/mkdir", "-m","0777", "Suser-Sserver");
30:  }
31:  $parser->output_dir("./$user-$server");
32:  my $mail = $pop->Retrieve($message);
33:  my Sentity = $parser->parse_data($mail);
34:  my $head = Sentity->head;
35:  print qq(Кому: ) . text_to_html($head->get('To',0));
36:  print qq(Копия: ) . text_to_html($head->get('Cc',0))
       if $head->get('Cc',0);
37:  print qq(Тема: } . text_to_html ($head->get('Subject',0));
```

```
print qq(От: ) . text_to_html ($head->get('From',0) ) ;
print qq(Дата: ) . $head->get('Date',0) . qq(<br>\n);
my $parts = $entity->parts;
print p;
if (!$parts) {
  print text_to_html ($entity->body_as_string);
  $entity->purge;
) else {
  print text_to_html (
    $entity->parts(0)->bodyhandle->as_string);
 print h3("Вложения: " ) ;
  for (my $i=1; $i<$entity->parts;$i++) {
    (my $temp_file_name =
    $entity->parts($i)->bodyhandle->path)
    =~ s!^.*$user-$server/!!;
    my $un_file = un_escape($temp_file_name);
    if ($temp_file_name=~/\. (gif|jpg|png)$/i) {
      print qq(Файл: img src=
       "$user-$server/$temp_file_name">);
    } else {
      print qq(Файл: <a href="view_att.cgi?att=$uri_file">
       $temp_file_name</a><BR>);
    }
  }
}
print p(-{-align=>'center'},
a({-href=>"index.cgi?delete=1&mess=$message"},
  "Удалить сообщение"}, " i ",
a({-href=>index.cgi"}, "Вернуться к почте"), " | ",
a({-href=>"pop-logout.cgi"}, "Выход"));
$pop->Close;
sub text_to_html {
  my $raw = shift;
  $raw =~ s! ([<>])! ($1 eq '<')? "&lt;": "&gt\;"!eg;
  $raw =~ s!  ((ht|f)tps?://)([\w-]*)((\..[^\s]*)+)!
    <a href="$1$3$4" target="external">$l$3S4</a>!g;
  $raw — s!([\w\-.]+)\@([\w-]+)((\.[\w-]+)*)!
    <ahref=\"pop-compose.cgi?to=$1\@$2$3\">$1\@$2$3</a>!g;
  $raw =~ s/\n/<bi->/g;
  return $raw;
}
sub error {
  my $err = shift;
  print h3("Произошла ошибка при подключении к
          серверу($err). Попробуйте подключиться
          снова"),p,
  a({-href=>'index.cgi'},"Назад");
  exit;
}
```

**Листинг 10.7. Полный текст pop-send.cgi**

```
#!/usr/bin/perl -wT
# pop-send.cgi
use strict;
use CGI qw(:standard);
use MIME::Parser;
```

```
06: use Net::SMTP;
07: use Untaint;
08: use CGI::Carp qw(fatalsToBrowser);
09: my $cookie = cookie ('WebMail') || 0;
10: my ($user, $pass, $server);
11: if (!$cookie) {
12:   print redirect("pop-login.html"); exit;
13: } else {
14:   ($user, $pass, $server) = split(" ",$cookie);
15: }
16: $server = untaint(qr(.*), $server);
17: $user = untaint(qr(^\w{1,8}$), $user};
18: $pass = untaint(qr(^\w{1,8}$), $pass);
19: my @to = spiit(/;\s+/,param('to'));
20: my @cc = split(/;\s+/,param('cc'));
21: my $subject = param('subject');
22: my @body = split(/\n/,param('body'));
23: my $top = build MIME::Entity ('X-Mailer' => 'WebMail',
24:                                -From => "$user\@$server",
25:                                -To => \@to,
26:                                -CC => \@cc,
27:                                -Subject => $subject,
28:                                 Data => \@body
29: );
30: my $smtp = new Net::SMTP ('smtp.host.com');
31: $smtp->mail("$user\$server") ;
32: $smtp->to(@to, @cc);
33: $smtp->data;
34: $smtp->datasend($top->stringify);
35: $smtp->dataend;
36: $smtp->quit;
37: print redirect('index.cgi');
38: exit;
```


# Введение в DBI и базы данных в Web

## Введение

Вероятно, самое полезное для Web — это базы данных. Без них большинство Web-страниц будут статическими — а статические страницы очень быстро устаревают. В наше время все серьезные сайты имеют какие-то базы данных, на основании которых и строится содержимое, видимое посетителям сайта.

В этой главе мы расскажем об использовании интерфейса DBI/DBD. DBI означает DataBase Independent (независимый от базы данных). Это название выбрано потому, что этот модуль применяется для баз данных любого типа; он независим от конкретной базы данных. DBD означает DataBase Driver (драйвер базы данных) или DataBase Dependent (зависимый от базы данных) в зависимости от того, о чем вы спрашиваете (используются оба термина). DBD — это часть интерфейса для работы с конкретным типом базы данных, т.е. драйвер, который вы должны загрузить. Во всех примерах мы используем тип MySQL, так как это очень хорошая база данных, и кроме того, она бесплатна. Perl и интерфейс DBI/DBD могут работать с множеством различных типов баз данных. Поищите на начальной странице DBI на `http://www.symbolstone.org/technology/perl/DBI` или на `http://search.cpan.org` интерфейс для базы данных, которую Вы хотите использовать. Вполне вероятно, что он уже есть там.

Чтобы выполнить примеры этой главы, вы должны установить базу данных, модуль DBI и модуль-драйвер DBD для вашей базы данных. Вам также нужно хотя бы базовое знание SQL. Команды SQL в этой главе довольно просты, и если вы быстро схватываете, здесь для вас не будет проблем. Однако не лишним будет хороший справочник по SQL.

Мы начнем эту главу с ознакомления с интерфейсом DBI. Несколько первых примеров даже не будут связаны с Web, но, как только вы овладеете некоторыми основами, мы перейдем к созданию приложения CGI для поиска в базе данных изделий. Мы не будем рассматривать все методы модуля DBI. Существует слишком много возможностей и вариантов его применения, чтобы их можно было охватить в этой книге. Гораздо больше информации можно найти в книге Рэнди Дж. Яргера (Yarger) *"MySQL и mSQL"*.

В первую очередь нам необходимы таблицы базы данных. В листинге 11.1 приведена структура двух таблиц, которые используются в этой главе. Создайте эти таблицы в вашей базе данных и добавьте в каждую по несколько записей, чтобы иметь данные для работы.

**Листинг 11.1. Создание таблиц для примеров главы**

```
CREATE TABLE products (
  sku       VARCHAR(20) NOT NULL PRIMARY KEY,
  mfg_pn    VARCHAR(30),
  name      VARCHAR(30),
  descr     VARCHAR(30),
  stock     INT,
  unit      VARCHAR(10),
  image     VARCHAR(50),
  vend_num  INT,
  price     VARCHAR(15)
);
CREATE TABLE vendors (
  vend_num  INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  address   VARCHAR(50),
  address2  VARCHAR(50),
  city      VARCHAR(50),
  state     VARCHAR(2),
  zip       VARCHAR(10),
  phone     VARCHAR(25),
  email     VARCHAR(150),
  url       VARCHAR(150),
  vname     VARCHAR(100),
);
```

# Использование Perl DBI

Наш первый пример очень невелик и никак не взаимодействует с таблицами базы данных. Он только показывает, какие драйверы установлены на сервере. Это очень полезно, так как для работы с системой надо знать, что в ней доступно. Кроме того, в этом примере генерируется ошибка, если модуль DBI не загружен, поэтому вы сразу же можете узнать, надо ли загрузить модуль DBI на сервер.

```
01:  #!/usr/bin/perl -wT
02:  # Пример 11-1
```

Строка 1 обязательна почти в каждом примере. Она сообщает системе, где найти Perl, и включает проверку на загрязнение и предупреждения.

Строка 2 — просто комментарий с именем программы.

```
03:  use strict;
04:  use DBI;
```

Строка 3 указывает программе использовать строгий синтаксис.

Строка 4 загружает модуль DBI — тот модуль, которому и посвящена вся эта глава.

```
05: my @drivers = DBI->available_drivers();
```

**Строка 5** создает массив `@drivers` и записывает в него результат вызова метода *available_drivers()* модуля DBI. В результате в массиве окажутся названия всех драйверов баз данных, или DBD, известных системе.

```
06: print "На данный сервер загружены следующие драйверы:\n\n";
07: print join("\n", @drivers);
08: print "\n";
```

**В строке 6** начинается вывод результатов программы.

**Строка 7** имеет одну особенность: она выводит все содержимое массива `@drivers` через функцию *join ()*. Так как в вызове *join ()* в качестве соединительного элемента текста указано \n, каждый элемент массива будет выведен в Отдельной строке.

Строка 8 выводит дополнительную пустую строку, чтобы выделить результаты (см. листинг 11.2).

**оверка имеющихся драйверов**

```
01: #!/usr/bin/perl -wT
02: # Пример 11-1
03: use strict;
04: use DBI;
05: my @drivers = DBI->available_drivers() ;
06: print "На данный сервер загружены следующие драйверы:\n\n";
07: print join{"\n", @drivers);
08: print "\n";
```

Результаты, выведенные этой программой, должны выглядеть примерно как в листинге 11.3 (конкретные результаты будут зависеть от того, какие драйверы фактически загружены на сервер). Отсюда видно, что на сервере имеется DBI и загружены следующие драйверы: ADO, ExampleP, Proxy и mysql. При указании драйверов имеет значение регистр, поэтому запомните написание выведенных названий.

**Листинг 11.3. Результаты предыдущего скрипта**

```
На данный сервер загружены следующие драйверы:
ADO
ExampleP
Proxy
mysql
```

# Подключение к базе данных

Следующий метод **DBI,** который мы рассмотрим, играет значительную роль. Метод *connect ()* служит для создания подключения к базе данных, которую программа будет затем использовать. Этот метод довольно прост; он будет применяться во всех примерах, где происходит обращение к базе данных, так что скоро он станет для вас очень привычным. А теперь посмотрим формат этого метода — мы должны знать, как действует каждая часть наших примеров.

```
$DBH =
  DBI->connect(источник_данных:компьютер:порт,пользователь,
    пароль, \%атрибуты)
  or die "Нельзя подключиться! $DBI::errstr\n";
```

*Источник_данных* и его синтаксис очень важны; указывать *компьютер* и порт необязательно. Если эти параметры не заданы, подразумевается локальный компьютер (localhost). Параметры *компьютер* и *порт* позволяют создать подключение и использовать методы DBI на одном сервере для базы данных на другом сервере.

Для подключения к базе данных MySQL под названием book на локальном компьютере (localhost) метод *connect ()* должен выглядеть так.

```
$DBH = DBI->connect("DBI:mysql:book", "username",
  "password") or die "Ошибка: $DBI::errstr\n";
```

Следующую форму будет иметь этот метод при подключении к той же базе данных на компьютере *www.myserver.com* через порт 333.

```
$DBH = DBI->connect("DBI:mysql:book:www.myserver.com:333",
  "username", "password")
  or die "Ошибка: $DBI::errstr\n";
```

Нельзя просто так подключиться к любому серверу базы данных. Пользователь и машина, с которой он подключается, должны иметь соответствующие разрешения. Как установить разрешения базы данных, можно узнать в документации к ней.

Параметр *источник_данных* начинается с обозначения DBI, за которым следуют название драйвера (mysql) и имя базы данных, а затем — имя компьютера и порт (если мы подключаемся к базе данных на удаленной машине). Имя пользователя и пароль необходимы в зависимости от настройки прав доступа к этой базе данных.

## Отключение от базы данных

За каждым успешным подключением должно следовать отключение. (Это просто неудачная шутка; продолжим изложение.) Метод *disconnect ()* служит для отключения от базы данных. Программа обязательно должна вызывать метод *disconnect <),* завершая свою работу. Не забывайте вызывать его и в подпрограммах обработки ошибок, чтобы в момент завершения программы, даже в случае ошибки, подключение было обязательно закрыто.

Если вы используете сервер, который поддерживает транзакции, необходимость в вызове *disconnect ()* дополнительно возрастает из-за опасности ошибок базы данных и непредвиденных откатов. Кроме того, если своевременно вызвать метод *disconnect ()* не удается, некоторые базы данных продолжают активную работу вплоть до истечения времени ожидания процесса, которое может быть довольно большим. Если несколько процессов в системе не делают ничего, это означает только пустую трату ресурсов и снижение производительности сервера. Метод *disconnect ()* очень прост. См. следующий пример:

```
$result = $DBH->disconnect;
```

В случае ошибки disconnect *()* возвращает значение, отличное от нуля. Возвращать значение в переменную ($result) не обязательно, но полезно, если требуется убедиться в успешном закрытии подключения.

# Подготовка и выполнение запроса SQL

Сейчас мы опишем два метода DBI. Два метода сразу мы рассмотрим по той причине, что они всегда вместе, как "жареный арахис и мягкая карамель" в сникерсе. Сейчас вы поймете, что я имею в виду.

Метод *prepare()* преобразует команду SQL во внутреннюю, откомпилированную форму и сохраняет ее. В MySQL и mSQL это внутреннее компилирование не поддерживается, но команда SQL все равно *сохраняется,* и поэтому prepare <) полезен и

для этих баз данных. Затем метод execute() выполняет, иначе говоря, запускает предварительно подготовленную команду SQL. Параметры, переданные в execute(), заменяют метки-заполнители в команде значениями. Следующий пример поможет вам понять, как все это происходит.

```
01:  #!/usr/bin/perl -wT
02:  # Пример 11-2
03:  use strict;
04:  use DBI;
```

Строка 1 сообщает программе, где найти Perl, и включает проверку на загрязнение и предупреждения.

Строка 2 — комментарий с именем программы.

Строка 3 загружает модуль strict.

Строка 4 загружает модуль DBI, что дает возможность работать с базами данных.

```
05:  my $dbh = DBI->connect("DBI:mysql:book","book","addison")
06:     or die "Ошибка: $DBI::errstr\n";
```

Строки 5 и 6 — фактически одна строка Perl. В этой строке создается переменная $dbh — скаляр, в котором сохраняется возвращаемое значение метода *connect ()*, дескриптор базы данных. Мы подключаемся к базе данных типа mysql под названием books на локальном компьютере (так как компьютер и порт не указаны). Далее, имя пользователя — book, а пароль — addison.

```
07:  my $sql = "SELECT mfg_pn, name, price FROM products,
       WHERE vend_num = ?";
08:  my $sth = $dbh->prepare($sql);
```

В строке 7 создается новая переменная и в нее записывается команда SQL. Обратите внимание на вопросительный знак (?) в конце этой команды. Это так называемая *метка-заполнитель*. Она играет роль переменной, позволяя программе передавать данные в команду SQL при ее выполнении.

Строка 8 вызывает метод prepare *()*, который сохраняет команду SQL (в предкомпилированной форме — для баз данных, которые это поддерживают). Ссылка на команду сохраняется в $sth.

```
09:  $sth->execute("31");
10:  print DBI::dump_results($sth);
```

В строке 9 вызывается метод execute *()*. Заметьте, что при его вызове мы не указывали дескриптор базы данных ($dbh). Вместо этого мы использовали дескриптор команды ($sth). Обратите также внимание на параметр, переданный в метод *execute ()*. Это значение — 31 — заменит метку-заполнитель, о которой мы говорили выше. Команда SQL будет выполнена со значением 31. Количество параметров в метод *execute() должно* соответствовать количеству меток-заполнителей в подготовленной команде SQL, иначе возникает ошибка.

Строка 10 выводит результаты только что выполненного запроса. Обычно метод *DBI: :dump_results* не применяется для вывода результатов пользователю, так как данные отображаются в "сыром", неформатированном виде. Здесь мы ввели его, только чтобы можно было убедиться в выполнении команды SQL.

```
11:  print "\n\n";
```

Строка И просто выводит два символа конца строки, чтобы легче было разделить в тексте разные запросы SQL.

```
12:  $sth->execute("5");
13:  print DBI::dump_results($sth);
```

**Строки 12-13** точно повторяют **строки 9—10.** Различие — только в значении, которое передается в метод `execute()`.

```
14:  $dbh->disconnect;
```

И так как все хорошее рано или поздно кончается, в **строке 14** мы вызываем метод *disconnect()*, чтобы завершить связь с базой данных.

Результаты работы этой программы будут несколько похожи на листинг 11.4.

**Листинг 11.4. Вывод программы**

```
0 rows
0
'Raq3-128', 'RaQ 3', '2379.99'
'CPD-G500', 'Sony Monitor', '1139'
'CPD-G400', 'Sony Monitor', '679.99'
'E400', 'Sony Monitor', '635'
'CPD-G200', 'Sony Monitor', '439.99'
5 rows
5
```

В первый раз мы выполнили команду SQL с числом 31. Для этого значения в базе данных не было найдено соответствия, поэтому было возвращено 0 строк. Во вторую команду мы передали 5 и получили пять соответствий. То, что переданное значение равно количеству возвращенных строк, — простое совпадение. 5 строк данных, которые соответствуют запросу, были выведены методом *dump_results()*.

Итак, теперь мы можем подключаться к базе данных и отключаться от нее, а также получать данные из базы. А сейчас мы научимся выполнять более сложные запросы и выдавать выборку данных таким способом, что форматирование вывода в HTML станет намного проще.

# Выборка данных

Способность выбирать данные, которые нам нужны, разумеется, имеет большое значение. Также для разработчиков очень важно, чтобы это было не особенно трудно. К счастью, Perl и модуль DBI позволяют получать данные довольно легко.

## Метод `fetchall_arrayref()`

Первый метод выборки данных, который мы рассмотрим — *fetchall_arrayref()*. Этот метод возвращает все имеющиеся в дескрипторе команды данные как ссылку на массив массивов. Каждый элемент такого массива — массив, содержащий строку данных. Если запрос не возвратил никаких данных, метод возвращает неопределенное значение (`undef`). Далее, если раньше была вызвана какая-то из двух функций *fetchrow_* (к которым мы перейдем чуть позже), *fetchall_arrayref()* возвратит все *оставшиеся* записи, а не записи, которые уже были выбраны.

Во всех последующих примерах мы будем работать с таблицами базы данных, описанными в начале главы.

```
01:  #!/usr/bin/perl -wT
02:   # Пример 11-3
03:  use strict;
04:  use DBI;
```

**Строки 1–4** остаются практически неизменными во всех наших программах.

```
05:  my $dbh = DBI->connect("DBI:mysql:book","book","addison")
06:    or die "Ошибка: $DBI::errstr\n";
```

Строки 5–6 — один оператор, в котором создается подключение к базе данных, и результат метода `connect()` сохраняется в переменной `$dbh`. Ее значение — ссылка на базу данных или ее дескриптор.

```
07:  my $sql = "SELECT sku, name, descr, stock FROM products";
08:  my $sth = $dbh->prepare($sql);
09:   $sth->execute;
```

Строка 7 создает команду SQL и сохраняет ее в переменной `$sql`. Вполне допустимо также передавать строку команды в метод *prepare ()* непосредственно, но сохранение строк SQL в переменных облегчает дальнейшую работу с ними. Например, можно создать хэш или массив, содержащий команды SQL, и затем только передавать в *prepare ()* ссылку на нужную команду.

Строка 8 подготавливает команду SQL и сохраняет ее в `$sth`.

Строка 9 вызывает метод *execute ()* для непосредственного выполнения команды SQL в базе данных.

```
10:  my $data = $sth->fetchall_arrayref;
```

**Строка 10** — вызов метода *fetchall_arrayref()*. В результате в переменной `$data` сохраняются все записи, извлеченные из базы при выполнении команды SQL в **строке 9.**

```
11:  foreach (@$data) {
12:    print join(' * ', @$_), "\n";
13:  }
```

В **строках 11—13** выводятся все полученные данные. После выполнения **строки 10** переменная `$data` содержит ссылки на несколько массивов. Так как она сама по сути является массивом, мы должны обращаться к ней, как к массиву.

В **строке** 11 мы обходим в цикле каждый элемент массива— переменной `$data`. Поместить `@` перед именем переменной — то же самое, что **написать** `@ [$data]`. В результате значение, хранящееся в `$data`, представляется в **виде** массива.

В **строке 12** текущая строка данных обрабатывается функцией *join ()* и выводится. Обычно, если просто вывести массив, его элементы будут отображены друг за другом без перерыва. Функция *join()* позволяет разделить данные любыми заданными символами. В данном случае мы поместим между элементами массива звездочку, окруженную пробелами. Данные передаются в функцию как список, как и требуется для функции *join ()*. Цикл *foreach ()* сохраняет текущий элемент массива в `$_`. По тем же соображениям, что и в **строке 11,** мы помещаем перед именем этой специальной переменной знак @. Таким образом, `@$_` в **строке 12** можно также записать как `@ [$_]`.

Строка 13 просто закрывает блок *foreach ()*.

```
14:  print "\n\n";
```

**Строка 14** выводит две пустые строки, чтобы сделать результаты более читаемыми.

```
15:  print "Ниже - значение в 2-й записи, в 3-й колонке:\n";
16:  print $data->[1][2], "\n\n";
```

Строка 15 выводит некоторый поясняющий текст.

Строка 16 выводит значение в определенной строке и колонке результатов запроса SQL.

```
17:  $dbh->disconnect;
```

Строка 17 отключает программу от базы данных.

Вот и все, что нужно для применения метода *fetchall_arrayref ()*. Этот метод обычно используется, когда нужно получить весь объем данных, выбранных из базы.

Во многих случаях требуется выдавать за один раз не *все* данные, а одну запись (строку). Для этой цели предназначены два метода: *fetchrow_arrayref()* и *fetchrow_hashref()*. Так как мы начали говорить о массивах, естественным будет рассмотреть сначала метод *fetchrow_arrayref ()*.

## Метод **fetchrow_**arrayref ()

Метод fetchrow_arrayref () получает одну строку данных и сохраняет отдельные поля записи в массиве. При каждом новом вызове этого метода содержимое массива заменяется новыми данными. Это означает — нельзя сразу получить, например, третье поле четвертой записи; нужно сделать так, чтобы программа работала одновременно только с одной строкой. Это условие может показаться очень стеснительным, но на самом деле это не так страшно.

Следующий пример показывает, как можно обрабатывать данные по одной строке за раз. Кроме того, в этом примере мы переходим от интерфейса командной строки к настоящему CGI.

```
01: #!/usr/bin/perl -wT
02: # Пример 11-4
03: use strict;
04: use DBI;
05: use CGI qw(:standard);
```

**Строки 1–5** — такие же, что и в предыдущих примерах. Дополнительно мы загружаем модуль CGI, так как вывод будет производиться в Web-броузер.

```
06: my $data;
07: my $dbh = DBI->connect("DBI:mysql:book","book","addison")
08:   or die "Ошибка: $DBI::errstr\n";
```

**В строке 6** объявляется переменная $data. Это позволяет обращаться к ней далее в тексте скрипта, не получая предупреждений о нарушении строгого синтаксиса.

**В строках 7 и 8** мы подключаемся к базе данных и проверяем, не возникли ли при этом ошибки.

```
09: my $sql = "SELECT sku, name, descr, stock, price
      FROM products";
10: my $sth = $dbh->prepare($sql);
```

**Строка 9** — команда SQL, по которой мы получим данные из базы.

**Строка 10** готовит команду SQL к выполнению.

```
11: $sth->execute;
```

**В строке 11** команда SQL выполняется.

```
12: print header;
```

В **строке** 12 выводится заголовок **HTTP.** Использованная здесь функция CGI.pm позволяет сэкономить время и избежать ошибок при вводе кода.

```
13: print «HTML;
14:   <HTML>;
15:   <HEAD><TITLE>Вывод примера 11-4</TITLEx/HEAD>
16:   <BODY>
17:   <CENTER>
18:   <TABLE BORDER="1" CELLSPACING="0">
19:   <TR>
```

```
20:     <TD><B>Код</B></TD>
21:     <TD><B>Изделие</B></TD>
22:     <TD><B>Описание</B></TD>
23:     <TD><B>Кол-во на складе</B></TD>
24:     <TD><B>Цена</B></TD>
25:     </TR>
26:  HTML
```

**Строки 13—26** — включенный документ, в котором выводится начало кода HTML для страницы, которую генерирует программа. Также здесь начинается таблиц и выводятся заголовки колонок.

```
27:  while ($data = $sth->fetchrow_arrayref) {
28:    print qq(<TR>\n);
```

**Строка 27** начинает **цикл** *while ()*, который считывает строку из данных, соответствующих команде SQL. Цикл выполняется, пока не будут извлечены все данные, соответствующие запросу. Как только эти данные закончатся, цикл завершится, и программа перейдет к следующей части кода.

**Строка 28** выводит дескриптор HTML <TR>. На каждой итерации цикла *while ()* мы начинаем в таблице HTML новую строку для отображения строки извлеченных данных.

```
29:    foreach(@$data) {
30:      print qq(<TD>$_</TD>\n) ;
31:    }
```

**В строке 29** начинается цикл *foreach ()*, который выводит каждый элемент строки данных. Здесь переменная $data — ссылка на массив — путем помещения перед ней знака @ разыменовывается и представляется как массив. На каждой итерации цикла *foreach ()* **размещает** текущий элемент массива в специальной переменной $_.

**Строка 30** выводит код HTML, требуемый для отображения каждого элемента в таблице, и сам этот элемент.

**Строка 31** закрывает цикл *foreach ()*.

```
32:    print qq(</TR>\n);
33:  }
```

**Строка 32** выводит закрывающий дескриптор строки таблицы. В результате следующая строка данных будет выведена в новой строке таблицы.

**Строка 33** завершает цикл *while ()*.

```
34:  print qq(</TABLE></CENTER></BODY></HTML>);
35:  $dbh->disconnect;
```

**Строка 34** выводит код HTML для завершения страницы.

Наконец, **строка 35** производит отключение от базы данных. Никогда не забывайте закрывать открытые подключения, когда программа заканчивает работу, — это позволяет избежать лишнего расхода ресурсов.

## Метод **fetchrow_**hashref()

Сейчас мы рассмотрим еще более простой метод доступа к данным из базы. Этот раздел будет очень коротким, потому что доступ к данным по этому методу почти такой же, как и в предыдущем методе *(fetchrow_arrayref ()*). Предыдущую программу мы можем перенести сюда без изменений вплоть до строки 27. Нам понадобится разобраться заново только в цикле *while ()*, где отображаются данные, соответствующие запросу.

```
27:  while ($data = $sth->fetchrow_hashref) {
28:    print qq(<TR>\n);
29:    print qq(<TD>$data->{sku}</TD>\n);
30:    print qq(<TD>$data->{name}</TD>\n);
31:    print qq(<TD>$data->{descr}</TD>\n);
32:    print qq(<TD>$data->{stock}</TD>\n);
33:    print qq(<TD>$data->{price}</TD>\n);
34:    print qq(</TR>\n);
35  }
```

**В строке 27** начинается цикл *while(),* в котором выбираются данные, соответствующие запросу SQL. При вызове *fetchrow_hashref()* создается хэш, содержащий одну строку данных, причем имена полей становятся ключами хэша. Именно потому этот метод оказывается проще в использовании. С ним вы всегда будете точно знать, с каким полем работаете. В методе *fetchrow_array()* ко всем полям надо обращаться не по именам, а по индексам массива, и поэтому легко забыть, какой индекс какому полю соответствует.

**Строка 28** начинает новую строку данных.

Строки 29—33 выводят отдельные элементы данных в строке таблицы.

**Строка 34** завершает текущую строку.

Наконец, **строка 35** завершает цикл *while ().*

Остальная часть программы полностью повторяет предыдущий пример. Как можно убедиться, единственное различие между *fetchrow_array()* и *fetchrow_hashref()* состоит в том, что строка результатов возвращается в разных структурах данных, и поэтому значения из этой структуры извлекаются по-разному.

# Метод bind_columns ()

Одна из замечательных вещей, которые происходят, когда вы пишете книгу — то, что вам приходится постоянно сверяться с документацией, чтобы гарантировать, что вы даете правильную информацию. Польза от этого заключается в том, что, вместо того, чтобы не задумываясь писать код как обычно, вы вынуждены узнавать много нового. Моим любимым методом доступа к данным был *fetchrow_hashref()*, я постоянно применял его в своих программах. Меня вполне устраивал этот метод, и потому я не искал альтернатив для него.

Но теперь дело обстоит не так. Кроме *fetchrow_hashref ()*, я предпочитаю метод *bind_column():* он не только позволяет ясно видеть, с какими данными я работаю, но и экономит время при вводе.

В этом разделе мы подробно расскажем о важных особенностях этого метода, а полный листинг программы дадим в конце главы. По сути этот метод доступа не очень отличается от других. Отличие проявляется только в нескольких строках:

```
09:  my $sql = "SELECT sku, name, descr, stock, price
       FROM products";
10:  my $sth = $dbh->prepare($sql);
11:  $sth->execute;
12:  my ($sku,$prod,$descr,$stock,$price);
13:  $sth->bind_columns(undef, \($sku, $prod, $descr,
       $stock, $price));
```

**Строки 9—11** — такие же, как в предыдущих методах доступа.

**В строке 12** просто объявляются переменные, которые мы будем использовать для доступа к данным.

Все дело в **строке 13.** Здесь мы вызываем метод *bind_columns()* и передаем в него два параметра. Первый параметр служит для передачи настроек в базу данных и в MySQL и mSQL не используется (поэтому здесь стоит неопределенное значение undef). Второй параметр — это список скалярных ссылок на поля, которые мы выбираем. Ссылки *должны* передаваться в том порядке, в котором выбираются поля в команде SQL; количество скалярных ссылок должно соответствовать количеству данных, возвращаемых при запросе SQL.

А теперь — главное преимущество метода *bind_columns()*. Он "волшебным образом" связывает переменные, которые были в него переданы, со значениями текущей строки данных. Теперь не надо обращаться к элементу массива или ключу хэша, чтобы получить значение определенного поля. Достаточно просто взять имя переменной!

Остальная часть программы — такая же, что и в предыдущих примерах. Единственное отличие — цикл *while()* упрощается, уменьшается в объеме и уменьшает вероятность ошибки при наборе кода.

# Соединим все вместе

Итак, мы теперь умеем выполнять отдельные операции и даже создали несколько программ, которые обращаются к данным из базы. Но у нас нет практически ничего для взаимодействия с пользователем. Сейчас мы на базе всего изученного создадим небольшое Web-приложение, позволяющее пользователю зайти на Web-сайт и провести поиск в базе данных изделий. Чтобы оно выглядело более реалистично, мы подключим к нему обе созданные в начале главы таблицы: таблицу поставщиков и таблицу изделий.

В этом приложении пользователь сможет вводить строку для поиска и указывать поле, по которому приложение будет сортировать результаты. Затем приложение отобразит данные, которые искал пользователь. Такое приложение — отличная основа для формирования настоящей базы данных изделий, которую можно использовать для сетевого магазина.

```
01:  #!/usr/bin/perl -wT
02:  # Пример 11-7
03:  use strict;
04:  use DBI;
05:  use CGI qw(:standard);
06:  use CGI::Carp qw(fatalsToBrowser);
```

**Строки** 1–6 — в основном стандартный код, применяемый для программ CGI. Единственное отличие здесь — загрузка в **строке 6** модуля Carp, чтобы все ошибки отображались в броузере.

```
07:  my  ($Data, $prod_sth, $vend_sth);
OS:  my  $Search_String = param('search_for');
09:  my  $Sort_Field = param('sort_by');
```

В строке 7 создается несколько переменных для последующего использования в программе. Переменная $Data будет содержать ссылку на текущую строку данных, а два дескриптора команд ($prod_sth, $vend_sth) предназначены для будущих команд SQL.

В строках 8–9 в новые переменные записывается информация из формы HTML, которая вызывает эту программу.

```
10:  my $dbh = DBI->connect("DBI:mysql:book","book","addison")
11:     or die "Ошибка: $DBI::errstr\n";
```

Строки **10—11** — один оператор Perl, который подключает нас к базе данных и сохраняет дескриптор базы данных в переменной $dbh.

```
12:  print header;
13:  get_products();
14:  print_output();
```

**Строка 12** выводит заголовок HTTP, необходимый для всех программ CGI, которые направляют свои результаты в броузер.

**Строка** 13 вызывает подпрограмму *get_products ()*, которая служит для получения данных об изделиях из базы.

**Строка 14** вызывает подпрограмму *print_output ()*, которая выводит пользователям найденные данные.

```
15:  sub get_products{
16:    my $sql   = "SELECT * FROM products WHERE
17:      descr LIKE ? ORDER BY ?";
```

**В строке 15** начинается подпрограмма *get_products*. Эта подпрограмма извлекает из базы все изделия, имеющие соответствие с тем, что ищет пользователь..

**Строки 16-17** — команда SQL, которая непосредственно извлекает соответствующие записи. Эта команда содержит две **метки-заполнителя** (обозначенных вопросительными знаками): одну для искомого текста и вторую для указания поля, по которому должны быть отсортированы результаты.

```
18:    $prod_sth = $dbh->prepare($sql);
19:   $prod_sth->execute("%$Search_String%", $Sort_Field) ;
20:    }
```

**Строка** 18 подготавливает команду SQL, созданную в **строках 16 и 17.**

**Строка 19** выполняет команду SQL. В метод *execute ()* передаются два параметра, соответствующие двум **меткам-заполнителям** в команде **SQL**. Заметьте, что первый параметр, `$Search_String`, окружен символами %. В SQL знак % считается символом шаблона, и поэтому будут найдены все строки, в которых перед и после значения `$Search_String` может стоять какой-то текст. Если не сделать этого, при поиске будут найдены только точные совпадения.

Наконец, **строка 20** завершает подпрограмму.

```
21:  sub get_vendor {
22:    my $vendor = shift;
```

**Строка 21** начинает подпрограмму *get_vendor()*. Эта подпрограмма вызывается каждый раз, когда в цикле *while ()* извлекается строка данных, и позволяет получить название поставщика из таблицы поставщиков. Благодаря этому мы можем определить по номеру поставщика, который хранится в таблице изделий, его фактическое название.

**Строка 22** создает переменную $vendor и записывает в нее значение, переданное при вызове подпрограммы.

```
23:    my $sql = "SELECT vname FROM vendors WHERE
24:      vend_num = ?";
```

**Строки 23—24** — команда SQL, с помощью которой мы будем определять название поставщика. Эта команда снова включает **метку-заполнитель.** На сей раз через нее будет передан номер поставщика из текущей записи в таблице изделий.

```
25:  $vend_sth = $dbh->prepare($sql);
26:  $vend_sth->execute($vendor);
27:  |
```

**Строка** 25 подготавливает команду SQL из **строк 23 и 24.**

**Строка 26** выполняет команду SQL и передает в метод execute *()* номер поставщика.

Строка 27 завершает подпрограмму *get_vendor ()*.

Итак, теперь у нас есть почти все, что нужно. Мы можем извлечь из базы данные изделия, а из другой таблицы — название поставщика. Остается только отобразить результаты, чтобы пользователь смог увидеть то, что он искал.

```
28:  sub print_output {
29:    print «HTML;
30:      <HTML>
31:      <HEAD><TITLE>Вывод примера 11-7</TITLEx/HEAD>
32:      <BODY>
33:      <CENTER>
34:      <TABLE BORDER="1" CELLSPACING="0">
35:      <TR>
36:      <TD><B>Код</B></TD>
37:      <TD><B>Поставщик</B></TD>
38:      <TD><B>Название</B></TD>
39:      <TD><B>Описание</B></TD>
40:      <TD><B>Кол-во на складе</B></TD>
41:      <TD><B>Цена</B></TD>
42:      </TR>
43:  HTML   ;
```

**Строка 28** начинает подпрограмму *print_outp()*.

**В строке 29** начинается включенный документ, в котором формируется страница результатов.

**Строки 30—42** — код HTML, образующий начало страницы результатов.

**Строка 43** завершает включенный документ.

```
44:     while($Data = $prod_sth->fetchrow_hashref) {
45:       get_vendor($Data->{vend_num});
46:       my $vendor = $vend_sth->fetch->[0];
```

**Строка 44** начинает цикл *while()* для вывода строк данных, соответствующих запросу. Этот фрагмент кода очень напоминает **циклы** из предыдущих примеров. В данном примере мы используем для доступа к данным метод *fetchrow_hashref*

**Строка 45** вызывает подпрограмму *get_vendor()* и передает в нее номер поставщика текущего изделия ~ значение `$Data->[vend_num]`.

В **строке** 46 создается переменная $vendor и в нее записывается название поставщика текущего изделия.

При получении имени поставщика мы делаем несколько предположений относительно целостности базы данных, с которой мы работаем. Мы считаем, что каждому номеру поставщика соответствует только одно название. Так как номер поставщика — первичный ключ таблицы и автоинкрементная переменная, мы можем быть достаточно уверены, что каждый номер уникален. Дополнительную гарантию этого может дать проверка ошибок.

Сделав эти предположения, мы с помощью метода *fetch()* получаем данные из таблицы поставщиков. Этот метод практически идентичен методу *fetchrow_arrayref()* по сути, это другое его название, так что можно вызывать любой из них. Поскольку мы получаем данные в массиве и наши предположения говорят, что должно быть найдено только одно соответствие, оно будет находиться в нулевом элементе массива. Поэтому мы приходим к строке

```
my $vendor = $vend_sth->fetch->[0];
```

Здесь создается переменная $vendor и ей присваивается значение, возвращенное в 0-м элементе методом `fetch()`, вызванным для дескриптора команды $vend_sth. Вот и все.

```
47:       print «HTML;
48:         <TR>
49:         <TD>$Data->{sku}</TD>
```

```
50:         <TD>$vendor</TD>
51:         <TD>$Data->{name}</TD>
52:         <TD>$Data->{descr}</TD>
53:         <TD>$Data->{stock}</TD>
54:         <TD>$Data->{price}</TD>
55:         </TR>
56: HTML
57:     }
```

**Строки 47—56** — включенный документ, в котором каждый элемент данных отображается в таблице.

**Строка** 57 завершает цикл *while ()*. Этот цикл, занимающий строки **с** 44 по 57, для каждого найденного соответствия вызывает подпрограмму *get_vendor()* и выводит данные. Когда цикл завершается, все найденные данные уже отображены, и остается только закончить HTML страницы.

```
58:     print qq{</TABLE>};
59:     print qq(<A HREF="/book/db/example_11-7.html">
          Повторить поиск</A>);
60:     print qq{</CENTER></BODY></HTML>};
```

**Строка 58** завершает таблицу **HTML.**

**В строке 59** выводится ссылка на тот случай, если пользователь захочет вернуться к программе и повторить поиск.

**Строка 60** выводит остальные дескрипторы HTML, необходимые для завершения страницы.

```
61:     $dbh->disconnect;
62: }
```

**Строка 61** выполняет отключение от базы данных.

**Строка 62** завершает подпрограмму *print_results*

Взгляните на этот пример: только 62 строки кода, большей частью просто вывод HTML, и у нас получилось приложение CGI, которое просмотрит для нас базу данных, найдет нужную информацию в нескольких таблицах и выведет результаты на экран в удобочитаемой форме. Не так уж и плохо!

## Метод do ()

Прежде чем перейти к упражнениям, полезно рассмотреть еще один заслуживающий внимания метод. При желании вы можете применить этот метод, выполняя упражнения.

Метод do (} используется для команд SQL, которые *не возвращают* никаких данных. В него надо просто передать команду SQL, и она будет выполнена. Этот метод заменяет оба метода prepare *()* и execute *()*. Применять его или нет, зависит только от вашего выбора. prepare *()* и execute *()* работают прекрасно, но *do ()* позволяет сократить программу на один вызов. Ниже приведен пример использования *do ()*. Работать с ним очень легко.

```
$sth->do("INSERT INTO products VALUES ('a','b','c','d')");
```

## Заключение

Модуль **DBI** дает очень эффективный и простой способ внедрения баз данных в Web. Этот модуль надежен и обладает полным набором возможностей. Он обладает универсальными свойствами. Если требуется перейти к другому типу базы данных, это не составляет труда. Надо просто загрузить драйвер DBD для этой базы данных, установить его, изменить строку подключения — и можно продолжать работу.

Есть также несколько методов DBI, не описанных в этой главе. Информацию по всем доступным методам и примеры их использования можно найти в превосходной документации, поставляемой с модулем **DBI**, и на нескольких Web-сайтах.

# Упражнения

- Создайте Web-интерфейс для добавления в таблицы изделий и поставщиков новых записей.
- Примените в приложении метод do *()*.
- Создайте телефонный справочник на базе Web.

# Листинги

### Листинг 11.5. Пример 11-2

```
01: #!/usr/bin/perl -wT
02:  # Пример 11-2
03:  use strict;
04:  use DBI;
05:  my $dbh = DBI->connect("DBI:mysql:book","book","addison")
06:    or die "Ошибка: $DBI::errstr\n";
07:  my $sql = "SELECT mfg_pn, name, price FROM products
       WHERE vend_num = ?";
08:  my $sth = $dbh->prepare($sql);
09:  $sth->execute("31");
10:  print DBI::dump_results($sth);
11:  print "\n\n";
12:  $sth->execute("5");
13: print DBI::dump_results($sth);
14:  $dbh->disconnect;
```

### Листинг 11.6. Программа 11-3

```
01: #!/usr/bin/perl -wT
02:  # Пример 11-3
03:  use strict;
04:  use DBI;
05:  my $dbh = DBI->connect("DBI:mysql:book","book","addison")
06:    or die "Ошибка: $DBI::errstr\n";
07:  my $sql = "SELECT sku, name, descr, stock FROM products";
08:  my $sth = $dbh->prepare($sql);
09:  $sth->execute;
10:  my $data = $sth->fetchall_arrayref;
11:  foreach (@$data) {
12:   print join(' * ', @$_), "\n";
13:  }
14:  print "\n\n";
15:  print "Ниже - значение в 2-й записи, в 3-й колонке:\n";
16:  print $data->[1][2], "\n\n";
17:  $dbh->disconnect;
```

### Листинг 11.7. Программам 1-4

```
01:  #!/usr/bin/perl -wT
02:  # Пример 11-4
03:  use strict;
04:  use DBI;
05:  use CGI qw(:standard);
06:  my $data;
07:  my $dbh = DBI->connect("DBI:mysql:book","book","addison")
08:    or die "Ошибка: $DBI::errstr\n";
09:  my $sql = "SELECT sku, name, descr, stock, price
       FROM products";
10:  my $sth = $dbh->prepare($sql);
11:  $sth->execute;
12:  print header;
13:  print «HTML;
14:    <HTML>;
15:    <HEAD><TITLE>Вывод примера 11-4</TITLE></HEAD>
16:    <BODY>
17:    <CENTER>
18:    <TABLE BORDER="1" CELLSPACING="0">
19:    <TR>
20:    <TD><B>Код</B></TD>
21:    <TD><B>Изделие</B></TD>
22:    <TD><B>Описание</B></TD>
23:    <TD><B>Кол-во на складе</B></TD>
24:    <TD><B>Цена</B></TD>
25:    </TR>
26:  HTML
27:  while ($data = $sth->fetchrow_arrayref) {
28:    print qq(<TR>\n);
29:    foreach(@$data) <
30:      print qq(<TD>$_</TD>\n);
31:    }
32:    print qq(</TR>\n);
33:  }
34:  print qq(</TABLE></CENTER></BODY></HTML>);
35:  $dbh->disconnect;
```

### Листинг 11.8. Программа 11-5

```
01:  #!/usr/bin/perl -wT
02:  # Пример 11-5
03:  use strict;
04:  use DBI;
05:  use CGI qw(:standard);
06: ray $data;
07:  my $dbh = DBI->connect("DBI:mysql:book","book","addison")
08:    or die "Ошибка: $DBI::errstr\n";
09:  my $sql = "SELECT sku, name, descr, stock, price
       FROM products";
10:  my $sth = $dbh->prepare($sql);
11:  $sth->execute;
12:  print header;
13:  print «HTML;
14:    <HTML>;
15:    <HEAD><TITLE>Вывод примера 11-5</TITLE></HEAD>
```

```
16:     <BODY>
17:     <CENTER>
18:     <TABLE BORDER="1" CELLSPACING="0">
19:     <TR>
20:     <TD><B>Код</B></TD>
21:     <TD><B>Изделие</B></TD>
22:     <TD><B>Описание</B></TD>
23:     <TD><B>Кол-во на складе</B></TD>
24:     <TD><B>Цена</B></TD>
25:     </TR>
26:   HTML
27:   while ($data = $sth->fetchrow_hashref) {
28:     print qq(<TR>\n);
29:     print qq(<TD>$data->{sku}</TD>\n);
30:     print qq(<TD>$data->{name}</TD>\n);
31:     print qq(<TD>$data->{descr}</TD>\n);
32:     print qq(<TD>$data->{stock}</TD>\n);
33:     print qq(<TD>$data->{price}</TD>\n);
34:     print qq(</TR>\n);
35   }
36:   print qq(</TABLE></CENTER></BODY></HTML>);
37:   $dbh->disconnect;
```

**Листинг 11.9. Программа 11-6**

```
01:  #!/usr/bin/perl -wT
02:   # Пример 11-6
03:   use strict;
04:   use DBI;
05:   use CGI qw(:standard);
06:   my $data;
07:   my Sdbh = DBI->connect("DBI:mysql:book","book","addison")
08:     or die "Ошибка: $DBI::errstr\n";
09:   my $sql = "SELECT sku, name, descr, stock, price
        FROM products";
10:  ray $sth = $dbh->prepare($sql);
11:   $sth->execute;
12:   my ($sku,$prod,$descr,$stock,$price);
13:   $sth->bind_columns(undef, \(Ssku, $prod, $descr,
        $stock, $price));
14:   print header;
15:   print «HTML;
16:     <HTML>;
17:     <HEAD><TITLE>Вывод примера 11-6</TITLEx/HEAD>
18:     <BODY>
19:     <CENTER>
20:     <TABLE BORDER="1" CELLSPACING="0">
21:     <TR>
22:     <TD><B>Код</B></TD>
23:     <TD><B>Изделие</B></TD>
24:     <TD><B>Описание</B></TD>
25:     <TD><B>Кол-во на складе</B></TD>
26:     <TD><B>Цена</B></TD>
27:     </TR>
28:   HTML
29:   while ($data = $sth->fetch) {
30:     print qq(<TR>\n);
```

```
31:    print qq(<TD>$sku</TD>\n);
32:    print qq(<TD>$prod</TD>\n);
33:    print qq(<TD>$descr</TD>\n);
34:    print qq(<TD>$stock</TD>\n);
36:    print qq(<TD>$price</TD>\n};
36:    print qq(</TR>\n);
37  }
38:  print qq(</TABLE></CENTER></BODY></HTML>);
39:  $dbh->disconnect;
```

## Листинг 11.10. Программа 11-7

```
01: #!/usr/bin/perl -wT
02:  # Пример 11-7
03:  use strict;
04:  use DBI;
05:  use CGI qw(:standard);
06:  use CGI::Carp qw(fatalsToBrowser);
07: my  ($Data, $prod_sth, $vend_sth);
08:  my  $Search_String = param('search_for');
09:  my  $Sort_Field = param('sort_by');
10:  my $dbh = DBI->connect("DBI:mysql:book","book","addison")
11:    or die "Ошибка: $DBI::errstr\n";
12:  print header;
13:  get_products();
14: print_output() ;
15:  sub get_products{
16:    my $sql   = "SELECT * FROM products WHERE
17:      descr LIKE ? ORDER BY ?";
18:    $prod_sth = $dbh->prepare($sql);
19:    $prod_sth->execute("%$Search_String%", $Sort_Field);
20:    }
21:  sub get_vendor {
22:  ray  Svendor = shift;
23:    my $sql = "SELECT vname FROM vendors WHERE
24:      vend_num = ?";
25:  Svend_sth = $dbh->prepare($sql);
26: $vend_sth->execute($vendor);
27:  )
28:  sub print_output {
29:    print <<HTML;
30:      <HTML>
31:       <HEAD><TITLE>Вывод примера 11-7</TITLE></HEAD>
32:      <BODY>
33:      <CENTER>
34:      <TABLE BORDER="1" CELLSPACING="0">
35:      <TR>
36:      <TD><B>Код</B></TD>
37:      <TD><B>Поставщик</B></TD>
38:      <TD><B>Название</B></TD>
39:      <TD><B>Описание</B></TD>
40:      <TD><B>Кол-во на складе</B></TD>
41:      <TD><B>Цена</B></TD>
42:      </TR>
43:  HTML
44:    while($Data = $prod_sth->fetchrow_hashref) {
45:      get_vendor($Data->{vend_num});
```

```
46:      my $vendor = $vend_sth->fetch->[0];
47:      print <<HTML;
48:        <TR>
49:        <TD>$Data->{sku}</TD>
50:        <TD>$vendor</TD>
51:        <TD>$Data->{name}</TD>
52:        <TD>$Data->{descr}</TD>
53:        <TD>$Data->{stock}</TD>
54:        <TD>$Data->{price}</TD>
55:        </TR>
56:  HTML
57:      )
58:     print qq(</TABLE>);
59:     print qq(<A HREF="/book/db/example_11-7.html">
          Повторить поиск</A>);
60:     print qq(</CENTER></BODY></HTML>);
61:     $dbh->disconnect;
62:  }
```


# Связанные переменные

## Введение

При написании программ время от времени оказывается, что ту или иную работу надо проделывать несколько раз, и желательно избавиться от такого повторения или уменьшить объем самой работы. Во многих таких ситуациях нас могут выручить связанные переменные.

Но действительно ли они сами могут выполнять все задачи, которые мы на них возлагаем? На самом деле связанные переменные — это просто переменные, с которыми для выполнения определенных задач связаны определенные методы. Это достигается ценой некоторой предварительной работы. Но после этой подготовки вы получаете мощный инструмент, который можно применять снова и снова.

Например, мы хотим, чтобы в строках таблицы чередовались разные цвета, но не желаем переписывать все значения цветов каждый раз, когда потребуется изменить цветовую схему. Или мы хотим от набора из двух цветов перейти к трем цветам — тогда придется пересмотреть код HTML и исправить *все* дескрипторы цвета для строк таблицы. Неужели нельзя сделать это проще? В подобных случаях связанная переменная прекрасно себя показывает.

Можно связывать скаляры, массивы, хэши и дескрипторы файлов. Хотя концепция в основном едина для всех типов, методы, которые надо создать, специфичны для каждого из них. В этой главе мы рассмотрим связывание скаляра и хэша. Механизм работы связанных переменных состоит в том, что, когда с этой переменной выполняются различные действия, вместо операций, которые были бы применены для обычной переменной, вызываются методы, созданные самим программистом. Это довольно трудно объяснить, так что попробуем разобрать все на простом примере. Допустим, мы создаем переменную и присваиваем ей значение 9.

```
$myvar = 9;
```

Можно сказать, что над этой переменной производится операция сохранения (STORE) значения 9. Затем, если мы хотим вывести эту переменную, мы пишем следующий код.

```
print "Значение: $myvar\n";
```

Можно ожидать, что результат будет выглядеть так:

```
Значение: 9
```

Но вместо этого мы видим

```
Значение: 99
```

Что произошло? Если переменная связана, мы можем изменять ее поведение. При выводе переменной мы произвели над ней операцию выборки значения (FETCH).

Получается, что для связанных скаляров можно задать и переопределить два метода: FETCH и STORE. В данном случае метод FETCH может выглядеть примерно так:

```
sub FETCH {
  my $self = shift;
  my $data = shift;
  return ($data * 11);
}
```

Это очень простой пример, но, надеемся, он поможет вам понять концепцию. А сейчас мы немного углубимся в скучные подробности. Нам придется разобрать кое-какой код, даже огромное количество кода! Другой пример связанного скаляра — переменная, которая увеличивает свое значение каждый раз, когда к ней обращаются. Из такой переменной получился бы прекрасный счетчик!

Скаляры — самый простой тип переменных, который можно связывать. Для них определены методы TIESCALAR, FETCH, STORE и DESTROY.

С массивами связываются методы TIEARRAY, FETCH, STORE, DESTROY, FETCHSIZE и STORESIZE. Можно также определить методы POP, PUSH, SHIFT, UNSHIFT, SPLICE, CLEAR, EXTEND, DELETE и EXISTS. С хэшами связываются методы TIEHASH, FETCH, STORE, DELETE, CLEAR, EXISTS, FIRSTKEY, NEXTKEY и DESTROY. Связанные методы дескрипторов файлов — TIEHANDLE, PRINT, PRINTF, WRITE, READLINE, GETC, READ и DESTROY. Можно также определить методы FILENO, SEEK и TELL. *Все* имена связанных методов/функций должны быть записаны в верхнем регистре и должны соответствовать приведенным выше.

К счастью, чтобы связать переменную, нет необходимости определять *все* эти методы самостоятельно. Для каждого типа переменных существуют модули Perl, которые определяют базовые возможности каждого требуемого метода. Это позволяет определять заново только те методы, которые вы желаете. Такие модули называются Tie::Scalar, Tie::Hash, Tie::Array и Tie::Handle. Более подробную информацию обо всем, что связано со связыванием, можно получить по команде perldoc perltie.

# Подготовительные работы

Так как в этой главе нам предстоит сделать довольно много, чтобы выполнить примеры, мы должны будем провести некоторую подготовку. Надеемся, что вы еще не удалили базу данных со своего сервера. В этих примерах мы снова будем использовать MySQL. Сейчас создадим несколько новых таблиц. Для работы с примерами нам будут нужны таблица "тележки для покупок" (cart), таблица изделий (products) и таблица сеанса (session). Ниже приведена информация, необходимая для создания этих таблиц в MySQL.

```
# Структура таблицы cart
CREATE TABLE cart (
```

```
  sku varchar(25),
  qty int(11),
  modified varchar(25),
  session varchar(25)
);
# Структура таблицы products
CREATE TABLE products (
  sku varchar(20) DEFAULT '' NOT NULL,   # Код изделия
  mfg_pn varchar(30),                    # Код, данный
                                         # производителем
  name varchar(50),                      # Название изделия
  descr varchar(255),                    # Описание изделия
  stock int(11),                         # Количество на складе
  unit varchar(10),                      # Единица измерения
  image varchar(50),                     # Название изображения
                                         #  (еще не используется)
  vend_num int(11),                      # Номер поставщика
                                         # для ссылки
  price varchar(15),                     # Цена (без знака валюты
  PRIMARY KEY (sku)   # Принимаем поле sku как первичный ключ
);
# Структура таблицы session
CREATE TABLE session (
  UID int(11) DEFAULT '0' NOT NULL auto_increment,
  expires varchar(25),
  PRIMARY KEY (UID)
);
# Структура таблицы vendors
CREATE TABLE vendors (
  vend_num int(11) DEFAULT '0' NOT NULL auto_increment,
                          # Номер поставщика
  address varchar(50),    # Строка адреса 1
  address2 varchar(50),   # Строка адреса 2
  city varchar(50),       # Город
  state char(2),          # Штат
  zip varchar(10),        # Почтовый индекс
  phone varchar(25),      # Номер телефона
  email varchar(150),     # Адрес электронной почты
  url varchar(150),       # URL, если есть
  vname varchar(100),     # Название поставщика
  PRIMARY KEY (vend_nura) # Принимаем поле vend_num как
                          # первичный ключ
);
```

Создав эти таблицы, введите какие-нибудь данные в таблицы изделий и продавцов, чтобы у нас была информация для поиска и обработки. Рекомендуется создать по крайней мере десять записей, чтобы вы смогли увидеть, как все это работает вместе.

# Начало

Мы будем опираться на то, что вы уже узнали в этой книге, особенно в главе, где речь шла о доступе к базе данных. Предупреждаем — наши примеры будут очень сложными, но вы должны хотя бы попробовать вникнуть в них. Слишком легко сразу, посмотрев на код, махнуть рукой и сказать: "А, это слишком сложно!" Действительно, хотя часть этого кода весьма не проста, мы хотим, чтобы вы точно поняли, как работает каждый его фрагмент.

В этой главе мы создадим три файла: cart.cgi— нашу главную программу, product_search.cgi — скрипт для выборки данных из базы и показа их пользователю и ShopCart.pm, содержащий методы для наших связанных переменных.

В первую очередь мы сформируем интерфейс, позволяющий пользователю искать изделия. Эта часть работы охватит простую версию программы cart.cgi, всю программу product_search.cgi и начало модуля ShopCart.pm.

# Погружение

Сейчас мы с вами "погрузимся" в первую программу. Это будет первый вариант cart.cgi. Версия cart.cgi в конце главы будет выглядеть несколько по-другому, но код, который мы сейчас напишем, войдет в нее практически без изменений.

```
01:  #!/usr/bin/perl -w
02:  use strict;
03:  use CGI qw(:standard);
04:  use lib qw(.);
05:  use ShopCart;
06:  $|=1;
```

**Строки 1—3** — стандартные строки, которые встречаются почти во всех наших программах.

В **строке** 4 в переменную @INC добавляется текущий каталог, представленный точкой (.). В результате Perl сможет находить модули в текущем каталоге. Обычно модули могут быть установлены только в одном из стандартных расположений Perl.

**Строка 5** импортирует модуль ShopCart.pra. Этот модуль, который мы вскоре создадим, будет сохранен в том же каталоге, что и данная программа. Поэтому нам и была нужна **строка 4.**

**Строка 6** задает для Perl автоматический сброс буфера. Если не сделать этого, Perl будет ожидать, пока вывод не закончится, перед тем как направить данные в броузер. При автосбросе буфера выводимые данные передаются в броузер сразу же.

```
07:  my ($color);
08:  tie $color, 'Colors', qw(ffffff e0e0e0);
```

**В строке 7** создается новая переменная для дальнейшего применения в программе. Прагма strict требует, чтобы все переменные были предварительно объявлены.

В **строке** 8 переменная $color связывается с классом Colors, содержащим значения цветов, которые будут чередоваться на нашей странице. Можно задать и большее количество цветов, не только два. С такой же легкостью мы можем поместить в этот класс и десять цветов, но тогда у нас получится чрезмерно пестрая таблица. Теперь, если нам потребуется изменить цвета или добавить новые, надо будет исправить только эту строку кода.

```
09:  Display_Search_Page();
10:  exit;
```

В **строке** 9 вызывается подпрограмма *Display_Search_Page()*, которая просто отображает форму HTML для поиска в базе данных изделий.

**Строка** 10 завершает программу. Эта строка не особенно необходима, но обычно там, где программа должна логически закончиться, помешают команду *exit()*, чтобы гарантировать, что любой код после этой строки будет выполняться только через вызов функции. В результате программист может легко распознать и отделить код самой программы от подпрограмм.

```
11:  sub Display_Search_Page {
12:    print header;
13:    print «HTML;
14:      <HTML><HEAD><TITLE>Поиск изделия</TITLE></HEAD>
15:      <BODY BGCOLOR="#FFFFFF">
16:      <CENTER>
17:      <FORM ACTION="/cgi-bin/tie/product_search.cgi"
          METHOD="POST">
18:      <H2>Тележка для товаров на базе связанного хэша</H2>
19:      <TABLE BORDER="1" CELLSPACING="0">
20:      <TRxTD>
21:      <INPUT TYPE="text" NAME="search_for">
22:      </TDXTD>
23:      <INPUT TYPE="submit" VALUE="Начать поиск">
24:      </TD><TR>
25:      </TABLE></FORM>
26:      <P>
27:      <A HREF="/cgi-bin/tie/cart.cgi?action=view"
28:      METHOD="POST">Просмотр тележки</A>
29:      </CENTER></BODY></HTML>
30:  HTML
31:  }  # Конец Display_Search_Page
```

**Строки 11–30 —** подпрограмма *Display_Search_Page()*. Эта подпрограмма не представляет собой ничего особенного; она просто содержит код HTML для формы поиска и выводит его привычным способом — через включенный документ.

Заметьте, что ссылка в **строке 27** фактически только возвращает нас снова на страницу поиска. В дальнейшем мы модифицируем программу `cart.cgi`, чтобы она действовала по-разному в зависимости от того, что передается в переменной `action`.

Но и в этом варианте программа совсем не плоха. Все в ней, кроме **строки 8,** должно быть уже хорошо знакомо вам. На рис. **12.1** показан **примерный** результат работы этой программы, которого можно будет достичь, когда мы напишем модуль `ShopCart.pm`. Учтите, что, пока мы не создадим этот модуль, `cart.cgi` не будет работать должным образом.

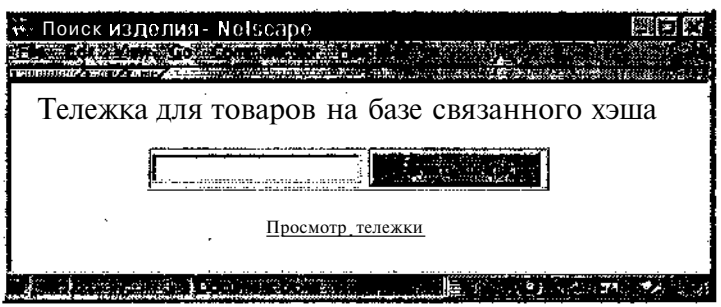



*Рис. 12.1. Поиск в тележке для покупок*

Все хорошо, но в форме HTML упоминается файл, который **еще** не существует, а в `cart.cgi` импортируется модуль, который еще должен быть создан. Сейчас мы перейдем к делу и создадим модуль, который позволит нам связать скаляр. После этого программу `cart.cgi` можно будет запустить и получить страницу, показанную на рис. 12.1, не сталкиваясь с ошибками.

## Начало модуля

Теперь мы можем приступить к формированию самого модуля. Хотя первая часть этого модуля предназначена для работы с цветами, мы выбрали для него название ShopCart.pm, которое отражает его главное назначение.

```
01:  package Colors;
02:  use Tie::Scalar;
03:  sub TIESCALAR (
04:    my ($class, @values) = @_;
05:    bless \@values, $class;
06:    return \@values;
07:  }
```

**Строка 1** сообщает Perl, что мы создаем пакет Colors. Это означает, что все объекты в этом блоке принадлежат к пространству имен Colors. Таким образом Perl может поддерживать организацию объектов программы. Если мы обращаемся к переменной $class, мы обращаемся к значению, которое находится в пространстве имен Colors. Поэтому, если в другом месте программы будет другая переменная $class, для Perl эти две переменные будут совершенно различными.

**В строке 2** загружается модуль Tie:: Scalar. Так как мы не хотим самостоятельно определять *все* методы для скаляра, который мы связываем, эта строка предоставляет стандартные методы, так что мы можем переопределять только те методы, которые хотим.

**Строка 3** начинает подпрограмму TIESCALAR. Вспомните, что это название стандартное и должно быть записано в верхнем регистре.

**В строке 4** создаются две переменные: Sclass — ссылка на сам класс и @values — список значений, переданных при вызове метода *tie()*.

**В строке 5** вызывается функция *bless ()*. Эта функция превращает класс в объект с принадлежностью к определенному классу. В данном случае это класс Colors.

**В строке 6** значение ссылки на объект возвращается из подпрограммы в массиве @values. Теперь у нас есть ссылка на массив, принадлежащий к классу Colors.

**Строка 7** завершает подпрограмму TIESCALAR.

```
08:  sub FETCH {
09:  my $self = shift;
10:  push(@$self, shift(@$self));
11:  return $self->[-1];
12:  }
13:  1;
```

**Строка 8** начинает подпрограмму FETCH. Нам требуется определить только эту подпрограмму, так как изо всех операций с данными нам нужна будет только эта.

**Строка** 9 создает ссылку на объект и называет ее именем Sself. Как мы помним, это ссылка на массив, для которого задана принадлежность к классу.

В **строке** 10 с помощью функций push *()* и *shift ()* значение из конца массива перемещается в его начало.

В **строке** И из подпрограммы возвращается последний элемент массива. Если указать для массива отрицательный индекс, Perl обращается к его элементам, начиная с конца, в обратном порядке. Этот прием позволяет сразу получать последний элемент из массива любой величины, не узнавая предварительно, сколько элементов он содержит.

**Строка** 12 завершает подпрограмму FETCH.

**В строке 13** возвращается значение истины. Это значение должны возвращать все модули, которые импортируются в программу.

Я хочу поблагодарить за эти замечательные материалы по использованию связывания скаляра Тома Кристиансена (Christiansen) и Натана Торкингтона (Torkington), которые в своей книге "*The Perl Cookbook*" приводят похожий пример. Этот способ применения связанных скаляров очень прост и полезен. Мы бы никогда не догадались создать связанный скаляр для цветов таблицы! Их пример содержит несколько больше дополнительных функций — мы взяли только те, которые нужны нам для нашей задачи.

Нашу начальную версию модуля ShopCart.pm можно считать готовой. Какие возможности есть у нас на этот момент? Уже можно отобразить страницу для поиска, как на рис. 12.1. Но ничего *действительно* полезного мы сделать еще не можем. Поэтому сейчас напишем скрипт, который извлекает данные из базы и возвращает их в программу.

## Поиск изделия

Так как задача поиска в базе данных и всей нашей программы — покупка товаров, пользователь должен иметь возможность искать товары, которые он хочет купить. Мы создадим отдельную программу, которая будет проводить поиск и выдавать его результаты. Все остальные функции будут выполняться в программе `cart.cgi`. Впрочем, этот выбор программист может сделать сам. В нашем случае мы посчитали, что задача поиска достаточно обособлена и достаточно сложна для того, чтобы выделить ее в отдельную программу.

Программа поиска изделия принимает значение, которое пользователь хочет найти, и ищет соответствие этому значению в полях описания нашей базы данных MySQL. После этого программа отображает таблицу записей, которые соответствуют критериям поиска, и дает пользователю возможность добавить какое-либо из найденных изделий в "тележку для покупок".

```
01:  #!/usr/bin/perl -wT
02:  use strict;
03:  use DBI;
04:  use CGI qw(:standard);
05:  use lib qw(.);
06:  use ShopCart;
```

Строки 1–6 не требуют пояснений.

Строка 5 добавляет в массив `@INC` текущий каталог, чтобы Perl мог найти модуль ShopCart в каталоге, в котором находится сама программа.

Строка 6 загружает в программу модуль ShopCart.

```
07:  my ($Data, $prod_sth, $vend_sth, $color);
08:  my $Search_String = param('search_for');
```

В строке 7 объявляется несколько переменных для последующего использования в программе.

В строке 8 создается переменная и ей присваивается значение, переданное из текстового поля `search_for` на странице HTML, которая вызвала эту программу.

```
09:  my $dbh = DBI->connect("DBI:mysql:book","book","addison"
10:    or die "Ошибка: $DBI::errstr\n";
```

Строки 9—10 — один оператор, занимающий две строки. Здесь мы создаем переменную $dbh и вызываем метод *DBI->connect()* для подключения к базе данных. Эта операция уже должна быть знакома вам, если вы прочли предыдущую главу.

```
11:  tie $color, 'Colors', qw(ffffff e0e0e0);
```

В строке 11 мы связываем скалярную переменную. Для этого вызываем функцию `tie()` и указываем в ней переменную, с которой связываем новые методы, а затем имя связываемого класса (с учетом регистра) и список данных, которые передаются в класс.

В данном случае мы передаем два **шестнадцатеричных** значения цвета, которые будут использоваться для строк нашей таблицы. Вспомните, что функция *qw()* (quote word), сопровождаемая круглыми скобками, заключает в кавычки *каждый* элемент в скобках.

```
12:  print header;
13:  get_products();
14:  print_output();
```

В **строке 12** функцией *header()* модуля CGI выводится заголовок HTTP.

В **строке 13** вызывается подпрограмма *get_products*, которая выдает все изделия, соответствующие тому, что искал пользователь.

В **строке** 14 вызывается подпрограмма *print_output*, которая отображает результаты для пользователя.

```
15:  sub get_products {
16:    $prod_sth = $dbh->prepare( qq{ SELECT * FROM products
17:                                   WHERE descr LIKE ?
18:                                   ORDER BY name });
```

**Строка 15** начинает подпрограмму *get_products()*.

**Строки 16—18 —** один оператор, охватывающий несколько строк. В строке 16 в переменную $prod_sth записывается ссылка на команду SQL, которую мы передаем в метод *$dbh->prepare()*. Дополнительно обрабатываем эту команду функцией *qq()*, позволяющей заключить в кавычки строку, которая сама содержит кавычки, без необходимости их сначала удалить.

```
19:    $prod_sth->execute("%$search_String%");
20:  }
```

**Строка 19** выполняет команду SQL. Данные, которые мы передаем в вызове метода *execute()*, заменяют метку-заполнитель (?) в команде SQL из **строки 17.** Знаки процента с каждой стороны строки — это символы шаблона SQL. Они позволяют находить не только точное соответствие, но и каждое поле, содержащее искомый текст.

**Строка 20** завершает подпрограмму *get_products()*.

```
21:  sub get_vendor {
22:    my $vendor = shift;
```

**Строка 21** начинает подпрограмму *get_vendor()*.

**Строка 22** создает переменную $vendor и присваивает ей значение, переданное в функцию. В данном случае в функцию передается номер поставщика, поэтому мы знаем, информацию о каком поставщике надо искать.

```
23:    $vend_sth = $dbh->prepare( qw{ SELECT vname FROM VENDORS
24:                                   WHERE vend_num = ? }),-
```

**Строки 23—24** напоминают **строки 16—18,** только выполняются для таблицы поставщиков, а не таблицы изделий. Здесь мы также формируем команду SQL для поиска в базе данных.

```
25:    $vend_sth->execute($vendor);
26:  }
```

**В строке 25** выполняется команда **SQL** из **строк 23-24.**

**Строка 26** закрывает подпрограмму *get_vendor()*.

```
27:  sub print_output {
28:    print <<HTML;
29:      <HTML>
30:      <HEAD><TITLE>Результаты поиска изделия</TITLE></HEAD>
```

```
31:       <BODY><CENTER>
32:       <TABLE BORDER="1" CELLSPACING="0">
33:       <TR BGCOLOR="#c0c0c0">
34:       <TD><B>Купить</B></TD>
35:       <TD><B>Поставщик</B></TD>
36:       <TD><B>Название</B></TD>
37:       <TD><B>Описание</B></TD>
38:       <TD><B>Кол-во на складе</B></TD>
39:       <TD><B>Цена</B></TD>
40:       </TR>
41:   HTML
```

**В строке 27** начинается подпрограмма *print_output ()*.

**В строке 28** начинается включенный документ, в котором мы выводим код **HTML** для начала страницы результатов.

**Строки 29—40** — сам этот код **HTML.**

**Строка 41** закрывает включенный документ.

```
42:     while (SData = $prod_sth->fetchrow_hashref) {
43:       get_vendor($Data->{vend_num});
44:       my $vendor = $vend_sth->fetch->[0];
45:       my $cart_link;
```

**Строка 42** начинает цикл *while ()* по всем данным, возвращенным из запроса к таблице изделий. Мы получаем данные методом *fetchrow_hashref()*, по одной строке за раз. В каждой итерации цикла в $Data записывается ссылка на хэш, содержащий данные.

**В строке 43** вызывается подпрограмма *get_vendor()* и в нее передается номер поставщика.

**В строке 44** создается переменная `$vendor` и ей присваивается значение 0-го элемента из полученного методом *fetch ()* массива. Так как в команде SQL выбирается только одно поле, а номер поставщика уникален (ведь мы сделали его первичным ключом), то в массиве будет получено только одно значение, а именно 0-й элемент.

В **строке 45** мы объявляем переменную `$cart_link`. Эта переменная в дальнейшем будет использована при создании ссылки HTML, позволяющей пользователю добавить изделие в свою тележку.

```
46.:      if ($Data->{stock} > 0) {
47:         $cart_link = qq(<A HREF="cart.cgi?) ;
48:         $cart_link .= qq(action=add&sku=$Data->{sku});
49:         $cart_link .= qq(">Добавить в тележку</A>);
50:       }
```

**В строке 46** начинается структура if. . .else. В этой структуре формируется ссылка, которая добавляет текущее изделие в тележку для покупок. Сначала мы проверяем, имеются ли такие изделия на складе (содержимое поля stock больше 0). Если да — начинаем первый блок if. . .else и создаем ссылку.

**В строках 47—49** формируется код **HTML** для ссылки. Оператор .= предоставляет простой способ конкатенации строк. При этом значение в правой части дописывается в конец существующей строки. **В строке 48** мы вводим **переменную** action, присваиваем ей значение add, а также присваиваем параметру sku значение $Data- > {sku}. Все эти данные передаются в программу cart.cgi в **форме операции HTTP** GET. **Переменные** action и sku **еще** не использовались в программе cart.cgi. Скоро **дойдет очередь** и до **них.**

**Строка 50** завершает первую часть **if...else.**

```
51:       else {
52:         $cart_link = qq(Добавить в тележку);
53:       }
```

**Строки 51—53—** вторая часть структуры `if...else`. Она выполняется, если на складе нет ни одного изделия данного типа. В этом случае мы сохраняем тот же текст, но не делаем его ссылкой.

**Строка 53** завершает структуру `if.. .else`.

```
54:       my $price = sprintf("%.2f", $Data->{price});
```

**В строке 54** в новую переменную `$price` записывается значение поля `price`, округленное функцией *sprintf()* до двух знаков в дробной части.

```
55:       print <<HTML;
56:         <TR BGCOLOR="$color">
57:         <TD>$cart_link</TD>
58:         <TD>$vendor</TD>
59:         <TD>$Data->{name}</TD>
60:         <TD>$Data->{descr}</TD>
61:         <TD>$Data->{stock}</TD>
62:         <TD>$price</TD>
63:         </TR>
64:   HTML
65:       >
```

**Строка 55** начинает включенный документ, в котором выводится текущая строка данных.

**В строке 56** устанавливается цвет фона для строки таблицы. Здесь мы видим переменную `$color` — наш связанный скаляр. Во время связывания мы передали в нее два цвета — эти цвета будут чередоваться в строках таблицы. Если вы хотите, чтобы чередовалось десять цветов, просто передайте при связывании десять значений! Что может быть проще?

**В строках 57—63** выводятся отдельные ячейки таблицы. В некоторые ячейки записываются значения обычных переменных, а в других используется ссылка на хэш `$Data`; все зависит от того, где хранятся данные.

**Строка 64** закрывает включенный документ.

**В строке 65** завершается цикл *while ()*, начатый в **строке 42. В** этом цикле мы выводили строку данных для каждого найденного соответствия.

```
66:       print qq(</TABLE>);
67:       print qq(<P><A HREF="cart.cgi">Вернуться в тележку</A>);
68:       print qq(</CENTER></BODY></HTML>);
```

**Строки 66—68** завершают код HTML, требуемый для страницы результатов поиска.

```
69:       $vend_sth->finish;
70:       $prod_sth->finish;
71:       $dbh->disconnect;
72:   }
```

**В строках 69—71** очищаются дескрипторы команд базы данных, созданные ранее.

**В строке 71** производится отключение от базы данных.

**Строка 72** завершает подпрограмму *print_output*.

Теперь, когда у нас есть код, дающий какое-то отображение на экране, попробуем его в действии, прежде чем перейти к главному "блюду" этой главы.

Если вызвать `cart.cgi` без параметров, должна получиться приблизительно такая страница поиска, как показано на рис. 12.1.

Введите в поле какой-нибудь текст и щелкните на кнопке Начать поиск. Вы получите примерно такой результат, как на рис. 12.2, в зависимости от того, что содержится в базе данных.

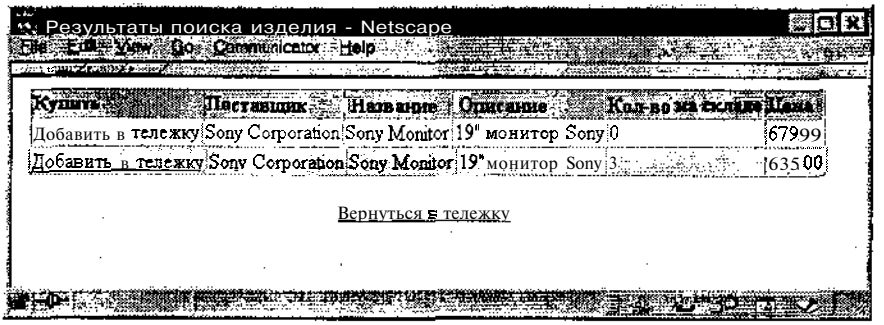



*Рис. J2.2. Результаты поиска*

Заметьте, что в колонке Купить в строке второго монитора находится подчеркнутая гиперссылка, а в строке первого — просто текст. Причина здесь в том, что для первого изделия в базе данных указано нулевое количество на складе, поэтому мы не создаем для него ссылку. Заметьте также, что строки таблицы имеют разные цвета: чередуются белый и серый. Более темный цвет заголовка таблицы был задан в коде вручную; он не определяется связанным скаляром.

Вы считаете, что уже можно приступать к самому проекту виртуальной тележки для покупок? Сначала мы должны определить, какие функции будут нужны для этого проекта. Ниже приводится примерный список таких необходимых функций.

## Функции

- Управление сеансом, чтобы содержимое "тележки для покупок" не терялось, когда пользователь делает перерыв в работе с системой.
- Способность менять количество каждого товара в тележке.
- Способность удалять товары из тележки.
- Использование связанного хэша.

Эту программу нельзя считать полнофункциональным продуктом промышленного качества, так как в ней нельзя делать настоящих покупок — ведь мы не интегрировали ее с программами составления счетов и операций с кредитными карточками. Все же она может послужить мостом для перехода к более серьезным проектам, так что когда придет время, вы уже будете знать, что делать.

Размышляя, с чего начать рассмотрение — с программы `cart.cgi` или с модуля `ShopCart.pm`, мы решили сначала разобрать программу `cart.cgi`, так как в модуле `ShopCart.pm` нам придется писать много функций, о назначении и применении которых хорошо бы что-то знать заранее. Если мы начнем с описания `cart.cgi`, то сможем увидеть, *что* должно выполняться в программе, а когда перейдем к `ShopCart.pm`, мы узнаем, *как* это будет выполняться.

Сейчас мы перейдем к программе `cart.cgi`, которая выполняет основную работу в этом проекте. Как мы помним, часть кода `cart.cgi` использует объекты, которых в модуле ShopCart.pm еще нет. Поэтому некоторые элементы нашего описания могут показаться несколько странными. Но когда мы допишем остальную часть `ShopCart.pm`, мы охватим и эти отсутствующие пока функции.

# Главная программа

```
01: #!/usr/bin/perl -w
02: use strict;
03: use CGI qw(:standard);
04: use lib qw(.);
05: use ShopCart;
06: $|=1;
```

Строки 1–6 — обычные для многих наших программ.

В строке 4 мы добавляем в массив @INC текущий каталог, благодаря чему Perl сможет найти модуль ShopCart, который мы загружаем в строке 5.

Строка 6 задает автосброс буфера, в результате чего данные немедленно передаются в броузер, а не хранятся в буфере, пока программа не завершится.

```
07:  my $action = param('action');
08:  my $Session_ID = Check_Session();
09: my (%cart, $color);
```

В строке 7 создается новая переменная и в нее методом *param ()* модуля CGI считываются данные, переданные в URL под именем action. Эти данные следуют в URL за знаком ?, например: `http://www.mysite.com/cart.cgi?action=view`.

В строке 8 создается переменная `$Session_ID` и в ней сохраняется результат вызова `Check_Session()`.

В строке 9 просто создаются две переменные, которые мы будем использовать позже.

```
10:  tie %cart, 'ShopCart';
11:  tie $color, 'Colors', qw(fffffffOeOeO);
```

В строке 10 хэш `%cart` связывается с классом ShopCart.

В строке 11 скаляр $color связывается с классом Colors и в класс передается список данных (значения цветов).

```
12:  if ($action =~ /view/) { View_Cart(); }
13:   elsif ($action =~ /update/) { Update_Cart(); }
14:   elsif($action =~ /add/) ( Add_Item(); }
15: else { Display_Search_Page(); }
16:  exit;
```

Строка 12 начинает структуру `if...elsif... else`. В этой структуре мы определяем, какое действие пользователь хотел выполнить, и вызываем соответствующую функцию. В строке 12 мы проверяем, хотел ли пользователь просмотреть тележку (действие view).

В строках 13 и 14 проверяются соответственно действия обновления и добавления элемента (update и add) и вызываются соответствующие подпрограммы.

В строке 15, если не выбрано ни одно из действий, пользователь направляется в подпрограмму *Display_Search_Page()*.

В строке 16 программа останавливается функцией *exit()*. Применение этой функции здесь необязательно, но рекомендуется для лучшей читаемости кода: можно легко заметить, где заканчивается основная программа и начинаются подпрограммы. Это также выручает в ситуации, когда после изменений в программе какой-то код случайно остался за пределами подпрограмм. Если бы этот код не был отделен командой `exit ()`, программа попыталась бы его выполнить, и вам бы пришлось гадать, почему выполняется какой-то код, когда программа уже должна закончиться.

```
17:  sub Add_Item {
18:    my Ssku = param('sku');
19:    Scart{$sku} = '1';
20:    Print_Added($sku);
21:  } # Конец подпрограммы Add_Item
```

**Строка** 17 начинает подпрограмму *Add_Item().*

В **строке** 18 создается переменная **$sku** и в нее функцией *param()* считывается значение sku с вызывающей **Web-страницы.**

В **строке** 19 в кэш %cart под ключом Ssku записывается значение 1. Как мы знаем, кэш %cart — связанный, поэтому с ним могут происходить **вещи,** недоступные для "нормальных" хэшей. В данном случае, если в %cart уже есть элемент под **этим** ключом, в него не записывается единица, а прибавляется 1 к прежнему значению.

В **строке** 20 вызывается подпрограмма *Print_Added()* и в нее передается Ssku. Эта подпрограмма создает простую страницу HTML, которая заверяет пользователя, что в тележку успешно добавлен еще один элемент.

**Строка** 21 завершает подпрограмму *Add_Item(),*

```
22:  sub Print_Added {
23:    my Ssku = shift;
```

**Строка** 22 начинает подпрограмму *Print_Added().* Эта подпрограмма создает страницу HTML, которая сообщает пользователю об успешном добавлении элемента (изделия) в тележку. В нашем примере эта страница не особенно красива — вы можете сами сделать ее такой изящной, как хотите.

В **строке** 23 создается переменная Ssku и в нее считывается значение, переданное в подпрограмму.

```
24:    print «HTML;
25:      <HTML><HEAD><TITLE>Добавлен элемент $sku</TITLE></HEAD>
26:      <BODY BGCOLOR="#ffffff">
27:      <CENTER>
28:      <H3>Добавлен элемент,$sku</H3>
29:      i <A HREF="/cgi-bin/book/tie/cart.cgi">
         Повторить поиск</A> ]
30:      [ <A HREF="/cgi-bin/book/tie/cart.cgi?action=view">
         Просмотр тележки</A> ]
31:      </CENTER>
32:      </BODY>
33:      </HTML>
34:  HTML
35:  } # Конец подпрограммы Print_Added
```

**Строки 24—34 —** включенный документ, в котором создается страница HTML.

Строка 35 завершает подпрограмму *Print_Added().*

```
36:  sub Check_Session {
37:    my Ssession = cookie('session');
```

**Строка 36** начинает подпрограмму *Check_Session().* Эта подпрограмма проверяет, имеет ли пользователь cookie сеанса. Если нет, такое cookie создается для него. Этот способ обладает тем недостатком, что, если пользователь отключит cookie в своем броузере, он ничего не сможет сохранить в тележке. В реальном **электронном** магазине следует поместить на главной странице предупреждение о необходимости включить cookie, если пользователь хочет сделать какой-то заказ.

**В строке 37** создается переменная Ssession и в нее с помощью подпрограммы CGI .pm *cookie ()* записывается значение cookie под названием session.

```
38:    if ($session) {
39:       print header();
40:       return $session;
41:    }
```

В **строке 38** проверяется, содержит ли переменная $session данные. Если это так, то пользователь уже имеет cookie, и мы выполняем первый блок кода.

В **строке 39** выводится стандартный заголовок **HTTP.** Пользователь уже имеет cookie, поэтому нам нет нужды создавать его.

В **строке 40** значение Ssession возвращается из подпрограммы.

**Строка 41** завершает Первую часть структуры if.

```
42:    else {
43:       $session = timed . $$;
44:       my $cookie = cookie( -name => 'session',
45:                            -value => $session,
46:                            -expires => '3h' );
```

**Строка 42** выполняется, если cookie не существует.

**В строке 43** в переменную записывается конкатенация текущего времени (количества секунд с начала эпохи) и идентификатора процесса (PID) текущей программы. В результате должно получиться уникальное число.

**В строках 44—46** создается cookie и в него записываются данные. Здесь мы задаем имя cookie — session, значение cookie — содержимое переменной $session, т.е. уникальное число, и срок действия — три часа. Вы сами можете решить, какой срок действия задать. Например, можно установить его в одну минуту, чтобы посмотреть, что произойдет с тележкой, когда срок действия cookie истечет.

```
47:       print header( -cookie => $cookie );
48:       return Ssession;
49:    }
50:  } # Конец подпрограммы Check_Session
```

**Строка 47** выводит заголовок HTTP и устанавливает в броузере пользователя cookie.

**Строка** 48 возвращает значение Ssession.

**Строка 49** завершает блок if. . .else.

**Строка 50** завершает блок *Check_Session().*

```
51:  sub Update_Cart {
52:     ray  $sku = param('sku');
53:  ray  $qty = param ('qty') ;
```

**Строка** 51 начинает подпрограмму *Update_Cart().* Эта подпрограмма вызывается, когда пользователь хочет набрать в свою тележку определенное количество изделий какого-то типа. На странице "Просмотр тележки" пользователь может изменить количество изделий и затем обновить тележку. Действие этой функции станет понятнее, когда мы закончим код и сможем запустить программу.

В строках 52 и 53 создаются две переменные и в них считываются данные, переданные из формы **HTML.** Это код изделия и новое количество.

```
54:    $cart{$sku} = $qty;
```

**В строке 54** в хэш %cart под **ключом** Ssku записывается новое количество — $qty.

```
55:    View_Cart();
56:    exit;
57:  } # Конец подпрограммы Update_Cart
```

В **строке** 55 вызывается подпрограмма *View_Cart ()*. Когда пользователь изменит количество товара, это изменение фиксируется в тележке и пользователь направляется в подпрограмму просмотра тележки.

**Строка** 56 завершает работу программы функцией *exi()* • Еще раз скажем, что это не обязательно, но впоследствии может сэкономить время при отладке.

**Строка 57** завершает подпрограмму *Upda te_Cart ()*.

```
58:  sub Session {
59:    return $Session_ID;
60:  } # Конец подпрограммы Session
```

**Строки 58—60** образуют подпрограмму, которая выглядит довольно тривиальной, но, используя связанный хэш, мы должны иметь какой-то способ получения `$Session_ID` из главной программы. Переменная `$Session_ID` имеет область действия в пределах главной программы, а мы должны обращаться к этому значению из пространства имен ShopCart. Данная подпрограмма решает именно эту задачу: передает значение `$Session_ID` из пространства имен главной программы.

```
61: ' sub View_Cart {
62:    print «HTML;
63:      <HTML>
64:      <HEAD><TITLE>Страница проверки тележки</TITLE></HEAD>
65:      <BODY><CENTER>
66:      <TABLE BORDER="1" CELLSPACING="0">
67:      <TR BGCOLOR="#c0c0c0"><TD ALIGN="CENTER">
68:      <H1>Страница проверки тележки</H1>
69:      </TDx/TR>
70:      <TR><TD>
71:      <B>Доставить по адресу:</B></BR>
72:      Amelia A. Camel<BR>
73:      321 Desert Dr.<BR>
74:      Sahara, CA 90220<BR>
75:      </TD></TR>
76:      <TRxTD>
77:      <TABLE BORDER="1" CELLSPACING="0">
78:      <TR BGCOLOR="#c0c0c0">
79:      <TD><B>Код</B></TD>
80:      <TD><B>Изделие</B></TD>
81:      <TD><B>Кол-во</B></TD>
82:      <TD><B>Цена</B></TD>
83:      <TD><B>Всего</B></TD>
84:      <TD><B>Обновить</B></TD>
85:      </TR>
86:  HTML
```

**Строка 61** начинает подпрограмму *view_Cart()*. Эта подпрограмма отображает пользователю содержимое тележки для покупок.

**Строки 62—86** — включенный документ, в котором формируется начало страницы и начинается таблица для содержимого тележки. Поле "Адрес отправки" в этом примере жестко задано, но в реальном проекте эту информацию можно было бы получать из базы данных пользователей.

```
87:  ray  $grand_total = 0;
```

**В строке 87** создается переменная `$grand_total` и ей присваивается начальное значение 0. В этой переменной будет храниться общая стоимость всех элементов (товаров) в тележке.

```
88:     while ( my($sku, $qty) = each %cart) {
89:       my ($name, $price) = Get_Product_Info($sku);
90:       my $total = sprintf("%.2f", ($price * $qty));
91:       $price = sprintf("%.2f", $price);
92:       $grand_total += $total;
```

Строка 88 начинает цикл *while()* по всем элементам хэша %cart. Пары ключ-значение сохраняются в переменных $sku и **$qty.**

Строка 89 вызывает подпрограмму *Get_Product_Info()* передает в нее значение $sku текущего элемента. Эта подпрограмма возвращает название и цену изделия, которые записываются в переменные $name и $price.

В строке 90 полная стоимость текущего элемента вычисляется как произведение $price * $qty, округляется функцией sprintf() до двух знаков после запятой и присваивается переменной $total.

В строке 91 функция *sprintf()* округляет значение $price до двух знаков после запятой.

В строке 92 $total прибавляется к текущему значению $grand_total.

```
93:       print <<HTML;
94:         <TR BGCOLOR="#$color">
95:         <FORM METHOD="POST">
96:        <INPUT TYPE="hidden" NAME=" action" VALUE="update">
97:        <INPUT TYPE="hidden" NAME="sku" VALUE="$sku">
98:         <TD ALIGN="left">
99:         $sku
100:        </TD>
101:        <TD ALIGN="left">$name</TD>
102:        <TD ALIGN="center">
103:       <INPUT TYPE=TEXT SIZE="2" NAME="qty" VALUE="$qty">
104:        </TD>
105:        <TD ALICN="right">\$$price</TD>
10 6:       <TD ALIGN="right">\$$total</TD>
107:        <TD ALIGN="center"><INPUT TYPE="SUBMIT"
                                VALUE="Обновить"></TD>
108:        </FORM>
109:        </TR>
110:  HTML
111:    }
```

**Строки 93-110** — включенный документ, в котором выводится одна строка, содержащая данные о текущем элементе в тележке. Заметьте, что в строке 94 мы используем наш связанный скаляр $color. В строках 97, 99, **101,** 103, 105 и 106 в ячейки таблицы динамически (через переменные) выводится некоторая информация.

Строка **111** завершает цикл *while ()*. Цикл продолжается с новой итерации, пока не обойдет все элементы в тележке для покупок, затем выполнение продолжится со строки **112.**

```
112:      $grand_total = sprintf("%.2f", $grand_total);
```

В строке **112** функцией *sprintf()* значение $grand_total округляется до двух знаков после запятой. Это происходит уже после окончания цикла *while ()*. В этот момент в переменной $grand_total будет накоплена общая стоимость всех товаров в тележке.

```
113:      print «HTML;
114:        </TABLE>
115:        <TR><TD ALIGN="RIGHT">
116:        <B>Bcero:</B> \$$grand_total
117:        </TD></TR>
```

```
118:        </TD></TR></TABLE>
119:        <P><A HREF="cart.cgi">Вернуться в тележку</A>
120:        <P>Чтобы удалить элемент, установите для него
121:          количество 0 и щелкните на кнопке "Обновить"
              около него:
122:        </CENTER></BODY></HTML>
123: HTML
124: } # Конец подпрограммы View_Cart
```

**Строки 113—123** — включенный документ, в котором выводится остальной код страницы **HTML. В строке 116** на странице отображается общая стоимость товаров.

**Строка 124** завершает подпрограмму *View_Cart()*.

```
125: sub Cet_Product_Info {
126:   my $sku = shift;
127:   my $dbh = DBI->connect("DBI:mysql:book","book","addison")
128:     or die "Ошибка: $DBI::errstr\n";
```

**Строка 125** начинает подпрограмму *Get_Product_Info*. Эта подпрограмма принимает переданный в нее код элемента (sku) и ищет этот элемент в базе данных изделий. Подпрограмма возвращает название элемента и его цену.

**В строке 126** значение sku, переданное в подпрограмму, помещается в переменную **$sku.**

**В строках 127—128** мы устанавливаем подключение к базе данных и сохраняем дескриптор базы данных в $dbh. Также мы проверяем, действительно ли получено подключение. Если подключиться не удалось, генерируется сообщение об ошибке.

```
129:   my $SQL = "select * from products where sku = ?";
130:   my $sth - $dbh->prepare($SQL);
```

**Строка 129** создает команду SQL, выбирающую из базы всю информацию по изделию, и сохраняет строку команды в $SQL.

**В строке 130** вызывается метод `prepare()` для команды в $SQL и результат сохраняется в $sth.

```
131:   $sth->execute($sku);
```

**В строке 131** вызывается функция `execute()`. Значение **$sku,** передаваемое в нее, заменяет метку-заполнитель (?) в команде SQL из **строки 129.**

```
132:   my $p = $sth->fetchrow_hashref;
133:   my $price = $p->{price};
134:   my $пате = $p->{пате};
```

**В строке 132** создается переменная $p, которую мы будем использовать как указатель. Мы присваиваем ей результат вызова *fetchrow_hashref()*. Это значение — указатель на данные, возвращенные методом *fetchrow_hashref()*. Так как поле sku, в котором мы искали, уникально, мы должны получить только одно значение. Поэтому *fetchrow_hashref()* можно не помещать в цикл.

В **строках 133—134** извлекаются значения цены и названия изделия и сохраняются в переменных $price и $name.

```
135:   $sth->finish();
13 6:  $dbh->disconnect();
```

**Строка 135** методом *finish ()* освобождает данные в дескрипторе команды.

В **строке 136** методом *disconnect ()* производится отключение от базы данных.

```
137:    return ($name, $price);
138: } # Конец подпрограммы Get_Product_Info
```

В строке 137 из подпрограммы возвращаются $name и $price.

Строка 138 завершает подпрограмму *Get_Product_Info*.

```
139: sub Display_Search_Page {
140:   piint «HTML;
141:     <HTML><HEAD><TITLE>Поиск изделия</TITLE></HEAD>
142:     <BODY BGCOLOR="#FFFFFF">
143:     <CENTER>
144:     <FORM ACTION="/cgi-bin/book/tie/product_search.cgi"
         METHOD="POST">
145:     <H2>Тележка для покупок на базе связанного хэша</H2>
146:     <TABLE BORDER="1" CELLSPACING="0">
147:     <TRxTD>
148:     <INPUT TYPE="text" NAME="search_for">
149:     </TDxTD>
150:     <INPUT TYPE="submit" VALUE="Начать поиск">
151:     </TD></TR>
152:     </TABLE></FORM>
153:     <P>
154:     <A HREF="/cgi-bin/book/tie/cart.cgi?action=view"
155:     METHOD="POST">Просмотр тележки</A>
156:     </CENTER></BODY></HTML>
157: HTML
158: } # Конец подпрограммы Display_Search_Page
```

Строки 139—158 — просто включенный документ, в котором отображается страница поиска. Эта страница отображается по умолчанию, если при вызове cart.cgi в URL не передается никаких параметров. Теперь программа cart.cgi готова, и нам осталось сделать только одну вещь, чтобы наша тележка для товаров стала работоспособной. Нам пришлось проделать довольно много работы, но если речь идет о таких сложных вещах, как виртуальная тележка для покупок, простым скриптом здесь не обойдешься. В итоге мы получили менее 350 строк кода, более 80 из которых посвящены выводу HTML, а не собственно операциям Perl. (Попробуйте добиться чего-то подобного на Java!).

# Доработка модуля ShopCart

В начале этой главы мы создали небольшой модуль под названием ShopCart.pm. В этот модуль были включены подпрограммы, необходимые для примера связанного скаляра. Теперь мы переделаем этот модуль так, чтобы он позволял работать с хэшем, связанным с базой данных.

```
01:  use Tie::Hash;
02:  use DBI;
```

В строках 1—2 мы загружаем некоторые необходимые модули. Модуль Tie: :Hash мы используем, чтобы не переписывать заново все модули для связанного хэша. DBI используется для доступа к базе данных. Заметьте, что в модуле нет начальной строки #! /usr/bin/perl. Модуль — не отдельная выполняемая программа, а должен вызываться другой программой Perl.

```
03:  my $dbh = DBI->connect("DBI:mysql:book","book","addison")
04:    or die "Ошибка: $DBI::errstr\n";
05: ray @KEYS;
```

**В строках 3-4** происходит подключение к базе данных и дескриптор базы данных сохраняется в $dbh.

**В строке** 5 объявляется переменная @KEYS.

```
06:  package ShopCart;
07:  @ISA = qw(Tie::StdHash);
```

**В строке 6** вводится новый пакет под названием ShopCart. В результате для всех методов после этой строки создается новое пространство имен.

**Строка 7** сообщает Perl, что мы хотим наследовать методы из модуля Tie::StdHash.

```
08:  sub STORE {
09:    my ($self, $key, $val) = @_;
```

**В строке 8** создается метод STORE для нашего связанного хэша.

**В строке 9** значения из массива @_ сохраняются в трех переменных. Здесь $self — ссылка на сам пакет, $key — переданный ключ хэша и $val — переданное значение.

```
10:    my $raodified = time();
11:    my $session = main::Session();
12:    my $existed = $self->EXISTS($key); # Ключ уже существует?
13:    my $action = main::param('action');
14:    my $new_qty;
```

**В строке 10** создается переменная $modified и в ней сохраняется текущее время. Мы внесем это значение в базу данных, чтобы облегчить очистку от данных, которые уже не должны храниться в базе.

**В строке 11** создается переменная $session.

Заметьте, что функция, из которой мы получаем значение, вызывается в форме *main::Session()*. Причина в том, что мы теперь находимся в пространстве имен ShopCart, и, если вызвать подпрограмму только по имени *Session (),* Perl станет искать подпрограмму *Session ()* в пространстве имен ShopCart, где ее нет. Программы работают в главном пространстве имен (main) по умолчанию, поэтому определять пакет main в программах на Perl нет необходимости. Указание пространства имен и :: перед именем подпрограммы — способ обратиться к ней из другого пространства имен.

**В строке 12** вызывается метод *$self->EXISTS($key)* и результат записывается в новую переменную $existed. Таким образом мы вызываем подпрограмму EXISTS из текущего пакета. Эта подпрограмма будет создана создадим чуть позже.

**Строка 13** создает переменную $action и присваивает ей значение, возвращенное функцией *param ('action '),* которая находится в главном пространстве имен.

**В строке 14** просто объявляется переменная, которую мы будем использовать позже.

```
15:    unless ($val) {
16:      $self->DELETE($key);
17:      return;
18:  }
```

**В строке 15** с помощью функции *unless ()* проверяется, не содержит ли $val какие-то данные. Эта функция — то же самое, что и условие "если не" (if not). Иными словами, здесь мы проверяем, содержит ли $val значение 0.

**В строке 16** вызывается метод *$self->DELETE<$key)*, который удаляет запись, соответствующую $key.

**Строка 17** возвращает нас в ту точку, откуда была вызвана подпрограмма.

**Строка 18** завершает блок подпрограммы.

```
19:    if ($existed) {
20:      if (Saction eq "update") {
21:         $new_qty = $val;
22:      } else {
23:        $new_qty = $existed + $val;
24:      }
```

**Строка 19** начинает блок if. . .else, в котором проверяется, существовал ли `$key` раньше. Если это так, выполняется первая часть этого блока.

**Строка 20** начинает еще один блок if. . .else. Здесь проверяется, не имеет ли `$action` значение "update". Если это так, в **строке 21** мы присваиваем элементу хэша новое значение, которое ввел пользователь.

**Строка 22** завершает первую часть блока if. . .else.

**Строка 23** начинает часть else. Эта часть выполняется, если `$action` имеет другое значение. Это происходит, если пользователь щелкнул на ссылке для добавления изделия в тележку.

**В строке 24 в** переменную *$new_qty* записывается сумма `$val` к текущему значению `$existed`. Когда пользователь добавляет изделие в свою тележку, щелкая на ссылке на странице результатов поиска, передается значение 1. Поэтому количество данных изделий просто увеличивается на 1.

**Строка 25** завершает второй блок if...else.

```
26:      my $sth = $dbh->prepare( qq{ UPDATE cart
27:                                   SET qty = '$new_qty'
28:                                   WHERE session = ?
29:                                   AND sku = ? });
```

**В строках 26—29** создается и подготавливается команда SQL для обновления таблицы тележки.

Помните, как мы записывали команду SQL в отдельную переменную и затем передавали ее в метод *$dhh->prepar()*? В Perl и эту операцию можно сделать несколькими способами (ЭМСНС). Функция qq заключает нашу строку в кавычки, избавляя нас от необходимости искать в ней и удалять специальные символы.

```
30:       $sth->execute($session, $key);
31:    }
```

**В строке 30** выполняется команда SQL, в которую передаются переменные `$session` и `$key`. Переменная `$session` сообщает, с каким пользователем в базе данных мы должны работать, а `$key` указывает изделие в тележке этого пользователя, для которого надо внести изменения.

Строка 31 завершает первую часть внешнего блока if...else.

```
32:    else {
33:      my $sth = $dbh->prepare( qq{ INSERT INTO cart
34:                                   VALUES (?,?,?,?) });
```

**Строка** 32 начинает часть else блока if. . .else.

**Строки 33—34** создают и подготавливают команду SQL для вставки нового элемента в таблицу тележки. Четыре **метки-заполнителя** представляют значения, которые будут переданы при вызове метода execute *()* для этой команды.

```
35:       $sth->execute($key, $val, $modified, $session);
36:    } # Конец блока if...else
37: } # Конец метода STORE
```

**Строка** 35 выполняет команду SQL, передавая в нее ключ, значение, время изменения и идентификатор сеанса.

**Строка 36** завершает внешний блок `if. . . else`.

**Строка** 37 завершает подпрограмму *STORE ()*.

```
38:  sub EXISTS {
39:    my($self, $key) = @_;
40:     my $session = main::Session();
```

**Строка 38** начинает подпрограмму *EXISTS ()*. Эта подпрограмма должна возвращать 0, если значение не существует, или другую величину, если оно существует.

**Строка 39** создает переменные $self и $key и присваивает им значения, переданные в подпрограмму.

**В строке 40** создается переменная $session и в нее записывается результат, возвращенный подпрограммой *main::Session()*.

```
41:     my $sth = $dbh->prepare( qq{ SELECT qty FROM cart
42:                                  WHERE session = ?
43:                                  AND sku = ? });
```

**В строках 41—43** создается и подготавливается команда SQL, которая выбирает из таблицы значение qty для данной записи.

```
44:     $sth->execute($session, $key);
45:     my $temp = $sth->fetch;
```

**В строке 44** выполняется команда SQL. Метки-заполнители заменяются переменными $session **и** $key.

**В строке 45** из базы извлекаются данные и ссылка на возвращенный массив сохраняется в $temp.

```
46:     return $temp->[0] ? $temp->[0] : 0;
47:  } # Конец метода EXISTS
```

**В строке 46** Возвращаемое значение определяется с помощью трехместного оператора. Perl вычисляет значение слева от знака ?. Если оно истинно (не равно 0), возвращается первое значение в правой части (между знаками ? и :). Если это ложное значение (равно нулю), возвращается второе значение.

**Строка 47** завершает подпрограмму *EXISTS ()*.

```
48:  sub DELETE {
49:    my ($self, $key) = @_;
50:     my Ssession = main::Session();
```

**Строка 48** начинает подпрограмму *DELETE ()*.

**В строке 49** создаются переменные $self и $key.

**В строке 50** создается переменная $session и в нее записывается значение, возвращенное подпрограммой *main : : Session ()*.

```
51:     my $sth = $dbh->prepare( qq{ DELETE FROM cart
52:                                  WHERE session = ?
53:                                  AND sku = ? }) ;
```

**В строках 51—53** создается и подготавливается команда SQL для удаления элемента из тележки.

```
54:     $sth->execute($session, $key);
55:  } # Конец метода DELETE
```

**В строке 54** выполняется команда SQL, которая удаляет элемент из тележки для покупок.

**Строка 55** завершает подпрограмму *DELETE <)*.

```
56:     sub FETCH {
57:     my ($self, $key) = @_;
58:     my $session = main::Session();
```

**Строка 56** начинает подпрограмму *FETCH()*.

**Строка 57** создает переменные $self и $key.

**В строке 58** создается переменная $session и в нее записывается значение, возвращенное подпрограммой *main: : Session ()*.

```
5.9:    my $sth = $dbh->prepare( qq{ SELECT qty FROM cart
60:                                  WHERE session = ?
61:                                  AND sku = ? });
```

**В строках 59—61** создается и подготавливается команда SQL для выборки элемента из таблицы тележки.

```
62:     $sth->execute($session, $key);
63:     my $temp = $sth->fetch;
```

**В строке 62** выполняется команда SQL, которая выбирает элемент в таблице тележки.

**В строке 63** в новую переменную $temp записывается указатель на данные, возвращенные методом *$sth->fetch ()*.

```
64:     return $temp->[0];
65:   } # Конец метода FETCH
```

**Строка 64** возвращает значение, хранящееся в $temp-> [ 0 ]. Так как мы выбирали в команде SQL только одно поле qty, в массиве, созданном методом *fetch()*, будет только один элемент. Это значение, если оно возвращается, — количество изделий данного типа в тележке для покупок. Если возвращается нуль, это означает, что под данным ключом в тележке ничего нет.

**Строка 65** завершает подпрограмму FETCH.

```
66:   sub FIRSTKEY {
67:     my $self = shift;
68:     my $session = main::Session();
```

**Строка 66** начинает подпрограмму *FIRSTKEY()*. Эта подпрограмма служит для получения первого ключа в хэше.

В **строке** 67 создается переменная, указывающая на текущий пакет.

**В строке 68** создается и заполняется данными переменная сеанса.

```
69:     my $sth = $dbh->prepare( qq{ SELECT qty, sku
70:                                  FROM cart
71:                                  WHERE session = ? });
```

**В строках 69—71** создается и подготавливается команда SQL, которая выбирает в тележке поля qty и sku для всех элементов, соответствующих текущему сеансу.

```
72:     $sth->execute($session);
```

**В строке 72** выполняется команда SQL, в которую передается текущее значение $session.

```
73:     $self->{DATA} = $sth->fetchall_arrayref;
74:     $self->{INDEX} = 0;
75:     my $val = $self->{DATA}->[0][0];
76:     my $key = $self->{DATA}->[0][1];
```

В **строке 73** создается элемент хэша с ключом DATA, в который записывается ссылка на данные, возвращенные методом *fetchall_arrayref()*.

**В строке 74** создается элемент хэша с ключом INDEX И В него записывается начальное значение 0. В дальнейшем мы будем отслеживать с его помощью индексы массивов.

В **строке** 75 создается переменная $val и в ней сохраняется значение `$self->{DATA}->[0][0]`.

Здесь дело немного усложняется Метод *fetchall_arrayref()* возвращает ссылку на массив, содержащий массивы, которые, в свою очередь, содержат данные, возвращенные по команде SQL. Иными словами, `$self->{DATA}->[0][0]` — это обращение к первому элементу первого массива в массиве, на который указывает `$self->(DATA}`. А `$self->{DATA}->` [0] [1] соответствует второму элементу первого массива в массиве, на который указывает `$self->{DATA}`.

**Строка 76** создает переменную $key и записывает в нее значение $self->{DATA)->[0][1].

```
77:     return if !@{$self->{DATA}};
        # Нет данных - возврат из метода.
```

**В строке 77** подпрограмма завершается, если по запросу SQL не было получено никаких данных. Нет смысла продолжать подпрограмму, если ей не с чем работать.

```
78:   foreach (@{$self->{ DATA)}) {
79:   ray   $key = $_->[1];
80:      my $val = $_->[0];
81:    - $self->{LIST}{$key} = $val;
82:   )
```

**Строка 78** начинает цикл *foreach ()* по всем записям, хранящимся в массиве, на который указывает $self-> [DATA].

**В строке 79** в новую переменную $key записывается значение второго элемента текущей записи.

**В строке 80** в новую переменную `$val` записывается значение первого элемента текущей записи.

**В строке 81** в хэш `$self->{LIST}{$key}` помещаются текущие ключ и значение. Пусть запись `$self->{LIST}{$key}` вас не смущает. Здесь `$self->{LIST}` подобно имени обычной переменной, но в данном случае эта переменная генерируется динамически.

**Строка 82** завершает цикл *foreach ()*.

```
83:   @KEYS = keys ( %{$self->{LIST}} );
```

**В строке 83** в массив `@KEYS` заносятся все ключи только что созданного хэша.

```
84:     $self->{FIRSTKEY} = $key;
85:     return ($val, $key) ;
86:  } # Конец метода FIRSTKEY
```

**В строке 84** `$self->{FIRSTKEY}` присваивается значение $key.

В строке 85 возвращаются ключ и значение.

**Строка 86** завершает подпрограмму *FIRSTKEY()*

```
87:  sub NEXTKEY {
88:    my $self = shift;
89:    my $lastkey = shift;
```

**Строка** 87 начинает подпрограмму *NEXTKEY()*. Эта подпрограмма извлекает в хэше следующий ключ.

В **строке** 88 создается переменная $self.

**В строке 89 создается** переменная $lastkey и в нее записывается следующее значение, переданное в подпрограмму.

```
90:    $key = $KEYS[$self->{INDEX}++];
```

**Строка 90** извлекает следующее значение из массива @KEYS и сохраняет его в переменной $key. Оператор ++ после $self->{INDEX} увеличивает эту переменную на единицу, так что в следующий раз мы получаем уже следующий элемент массива. Этот оператор хорош тем, что он не только увеличивает значение на единицу, но и возвращает текущее значение *перед* тем, как оно было увеличено.

```
91:    next if ($key eq $self->{FIRSTKEY});
92:    return($key);
93:  } # Конец метода NEXTKEY
```

В строке **91** программа переходит к следующему элементу массива, если текущий ключ равен $self->{FIRSTKEY}, чтобы мы не возвращали первый ключ дважды.

**Строка 92** возвращает $key для текущего значения.

**Строка 93** завершает подпрограмму *NEXTKEY()*.

```
94:  sub DESTROY {
95:    $dbh->disconnect();
96:  } # Конец метода DESTROY
```

**Строки 94—96—** подпрограмма *DESTROY()*. Эта подпрограмма просто закрывает подключение к базе данных.

```
97:  package Colors;
```

В **строке 97** начинается новый пакет. Все, что находится после этой строки, будет относиться к пакету Colors, пока не встретится новый пакет. Как мы помним, пакет в Perl — по существу просто пространство имен.

```
98:   use Tie::Scalar;
```

**В строке 98** импортируется модуль Tie::Scalar. Он содержит основные методы для скаляров. Это позволяет нам определять не все методы, а только те, которые мы хотим.

```
99:   sub TIESCALAR {
100:    my ($class, ©values) = @_;
101:    bless \@values, $class;
102:    return \@values;
103:  }
```

**Строка 99** начинает подпрограмму *TIESCALAR()*. Помните, что это имя стандартно и должно состоять из символов верхнего регистра.

**В строке 100** создаются две переменные: $class — ссылка на сам класс и @values — список значений, переданных при вызове метода *tie()*.

**В строке 101** вызывается функция *bless()*. Эта функция превращает класс в объект с принадлежностью к определенному классу. В данном случае это класс Colors.

**В строке 102** значение ссылки на объект возвращается из подпрограммы в массиве @values. Теперь у нас есть ссылка на массив, принадлежащий к классу Colors.

**Строка 103** завершает подпрограмму TIESCALAR.

```
104:  sub FETCH {
105:    my $self = shift;
106: S  push(@$self, shift(@$self));
107:    return $aelf->[-1];
108:  }
109:  1;
```

**Строка 104** начинает подпрограмму FETCH. Нам требуется определить только эту подпрограмму, так как изо всех операций с данными нам нужна будет только эта. Из скаляра мы будем только извлекать данные; операция сохранения не будет использоваться.

**Строка 105** создает ссылку на объект. Как мы помним, это ссылка на массив, для которого задана принадлежность к классу.

**В строке 106** с помощью функций *push ()* и *shift ()* значение из конца массива перемешается в его начало.

**В строке 107** из подпрограммы возвращается последний элемент массива. Если указать для массива отрицательный индекс, Perl обращается к его элементам, начиная с конца, в обратном порядке. Этот прием позволяет сразу получать последний элемент из массива любой величины, не узнавая предварительно, сколько элементов он содержит.

**Строка 108 завершает** подпрограмму *FETCH ()*.

**В строке 109** возвращается значение истины. Это значение должны возвращать все модули, которые импортируются в программу.

# Запуск программы

Наконец-то! Мы создали модуль. Мы получили программы управления тележкой для товаров и поиска изделий. Теперь охватим взглядом все приложение. Главная программа, которую надо запустить, — это программа `cart.cgi`. При первом запуске `cart.cgi` будет получен экран, показанный на рис. 12.1. Введите какой-нибудь текст для поиска. Мы задали слово *монитор* и получили страницу, показанную на рис. 12.3. Ваши результаты могут быть другими в зависимости от того, что вы ввели в базу данных.

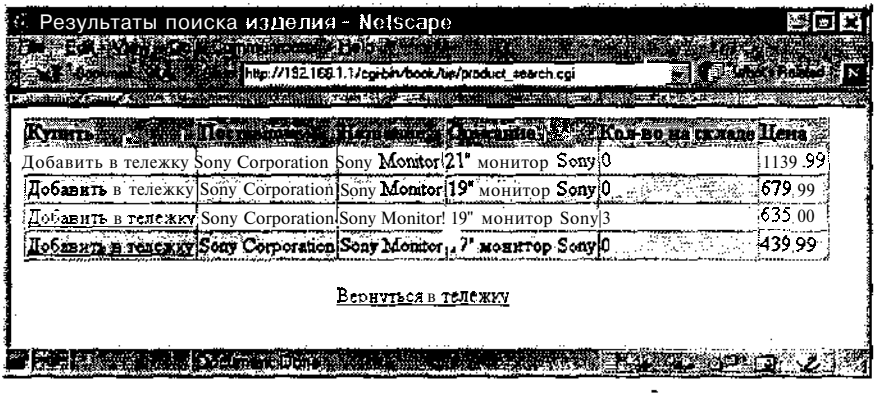



*Рис. 12.3. Результаты поиска*

Эта программа вызовет другую программу— `product_search.cgi`, которая выполнит поиск в базе данных и выдаст результаты в красивой таблице. Строки таблицы имеют чередующиеся цвета, так как для цвета строки таблицы мы применили связанный скаляр `$color`. Теперь пользователь может "положить" в свою тележку для покупок какой-нибудь элемент, щелкнув на одной из ссылок в колонке Купить. URL этих ссылок будет выглядеть примерно так.

```
http://www.mysite.com/cgi-bin/cart.cgi?action=add&sku=52973
```

Как можно заметить, здесь вызывается программа `cart.cgi` и в нее передаются некоторые данные способом GET. Эти данные — действие (`action`) add и код изделия (`sku`) 52 973. В результате программа `cart.cgi` добавит в тележку элемент 52973.

После шелчка на этой ссылке появляется экран, подтверждающий добавление изделия в тележку (см. рис. 12.4). Теперь вы можете щелкнуть на ссылке Повторить поиск или Просмотр тележки. Попробуйте добавить в тележку еще несколько элементов, а затем щелкните на ссылке Просмотр тележки.

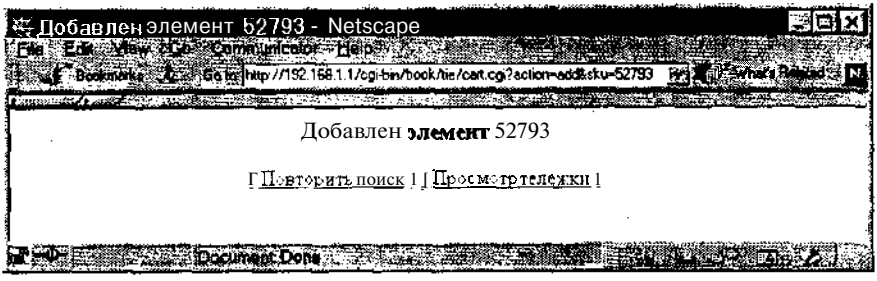



*Рис. 12.4. Страница подтверждения*

Появится страница, приблизительный вид которой показан на рис. 12.5.

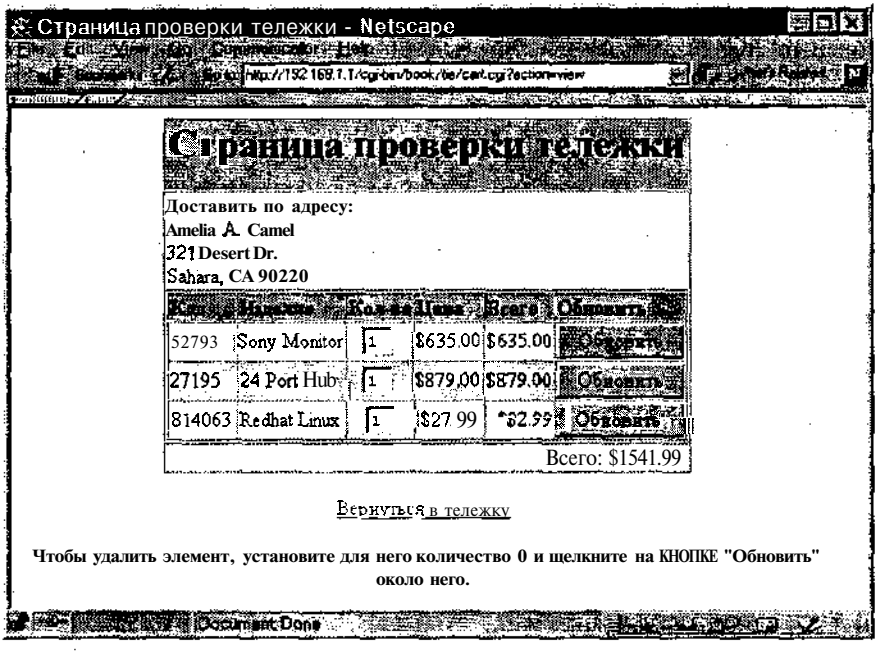



*Рис. 12.5. Страница проверки тележки*

Теперь URL будет выглядеть так:

```
http://www.mysite.com/cgi-bin/cart.cgi?action=view
```

Мы снова вызвали программу `cart.cgi`, но уже для другого действия (view). В результате будет выполнена другая часть программы `cart.cgi`. Результат — страница оформления заказа. Имя заказчика в этой программе жестко зафиксировано, но на реальном сайте оно также может быть динамическим.

Заметьте, что программа "запомнила", что было в вашей тележке. Это стало возможным благодаря cookie сеанса, которое мы установили в броузере. Сеанс уникален, т.е. данный номер сеанса есть только у вас.

Измените количество какого-нибудь товара и нажмите кнопку Обновить. Будет выведена новая страница, в которой соответственно изменится также общая стоимость покупки. Кроме этой суммы и измененного количества товара, все на этой странице останется, как на рис. 12.5. В ходе обновления мы направили новое значение в связанный хэш. Методы связанного хэша проверяют, имелся ли этот элемент в тележке раньше, и, если это так, обновляют запись базы данных в соответствии с новым количеством.

Теперь задайте для одного из элементов количество 0 и нажмите кнопку Обновить. Снова появится новая страница, но теперь элемент с нулевым количеством исчезнет! На этот раз подпрограммы вызвали метод DELETE.

Когда вы добавляете в тележку товары или изменяете их количество, программа `cart.cgi` обращается к созданным нами методам и при их помощи управляет связанной переменной. Можно даже закрыть броузер, но, когда вы снова вернетесь в программу, содержимое тележки окажется неизменным. Оно будет сохраняться, пока в системе существует cookie. Что касается функциональных возможностей, то мы достигли целей, которые поставили в начале работы над этим проектом.

### Функции

- Управление сеансом, чтобы содержимое "тележки для покупок" не терялось, когда пользователь делает перерыв в работе с системой.
- Способность менять количество каждого товара в тележке.
- Способность удалять товары из тележки.
- Использование связанного хэша.

Не так уж и плохо для одной главы!

# Заключение

При создании реального сайта можно будет обратить внимание на некоторые моменты.

- Можно создать скрипт для удаления старых элементов из базы данных и настроить его как событие планировщика (cron event).
- Дать пользователям возможность вводить свои имена и адреса для доставки.
- Динамически отображать имя пользователя на странице просмотра.
- Привести страницы программы в соответствие со стилем и дизайном вашего сайта.

# Листинги

### Листинг 12.1. cart.cgi

```
01: #!/usr/bin/perl -w
02: # cart.cgi
03: use strict;
04: use CGI qw{ :standard) ;
05: use lib qw(.);
06: use ShopCart;
07: $|=1;
08: my $action = param('action');
09: my $Session_ID = Check_Session();
10: ray (%cart, $color);
11: tie %cart, 'ShopCart';
12: tie $color, 'Colors', qw(ffffff e0e0e0);
```

```
13:  if ($action =~ /view/) { View_Cart (); }
14:    elsif ($action =~ /update/) { Update_Cart(); }
15:    elsif($action =~ /add/) { Add_Item(); }
16:  else { Display_Search_Page () ; }
17:  exit;
18:  sub Add_Item {
19:     my $sku = param('sku');
20:     $cart{$sku} = '1';
21:     Print_Added($sku);
22:  } # Конец подпрограммы Add_Item
23:  sub Print_Added {
24:    my $sku = shift;
25:    print «HTML;
26:       <HTML><HEAD><TITLE>Добавлен элемент $sku</TITLE></HEAD>
27:       <BODY BGCOLOR="#ffffff">
28:       <CENTER>
29:       <H3>Добавлен элемент $sku</H3>
30:       [ <A HREF="/cgi-bin/book/tie/cart.cgi">
          Повторить поиск</A> ]
31:       [ <A HREF="/cgi-bin/book/tie/cart.cgi?action=view">
          Просмотр тележки</A> ]
32:       </CENTER>
33:    .  </BODY>
34:       </HTML>
35:  HTML
36:  } # Конец подпрограммы Print_Added
37:  sub Check_Session {
38:    my $session = cookie('session');
39:    if ($session) {
40:      print header();
41:      return $session;
42:    }
43:    else {
44:      $session = time() . $$;
45:      my $cookie = cookie( -name => 'session',
46:                           -value => $session,
47:                           -expires => '3h' );
48:      print header( -cookie => $cookie );
49:      return $session;
50:    }
51:  } # Конец подпрограммы Check_Session
52:  sub Update_Cart {
53:    my $sku = param('sku');
54:    my $qty = param('qty');
55:    $cart{$sku} = $qty;
56:    View_Cart();
57:    exit;
58:  } # Конец подпрограммы Update_Cart
59:  sub Session {
60:     return $Session_ID;
61:  } # Конец подпрограммы Session
62:  sub View_Cart {
63:    print <<HTML;
64:       <HTML>
65:       <HEAD><TITLE>Страница проверки тележки</TITLE></HEAD>
66:       <BODY><CENTER>
67:       <TABLE BORDER="1" CELLSPACING="0">
68:       <TR BGCOLOR="#c0c0c0"><TD ALIGN="CENTER">
```

```
69:       <H1>Страница проверки тележки</H1>
70:       </TD></TR>
71:       <TR><TD>
72:       <B>Доставить по адресу:</B></BR>
73:     - Amelia A. Camel<BR>
74:       321 Desert Dr.<BR>
75:       Sahara, CA 90220<BR>
76:       </TD></TR>
77:       <TR><TD>
78:       <TABLE BORDER="1" CELLSPACING="0">
79:       <TR BGCOLOR="#c0c0c0">
80:       <TD><B>Код</B></TD>
81:       <TD><B>Изделие</B></TD>
82:       <TD><B>Кол-во</B></TD>
83:       <TD><B>Цена</B></TD>
84:       <TD><B>Всего</B></TD>
85:       <TD><B>Обновить</B></TD>
86:       </TR>
87:   HTML
88:     my $grand_total = 0;
89:     while ( my($sku, $qty) = each %cart) {
90:       my ($name, $price) = Get_Product_Info($sku) ;
91:       my $total = sprintf("%.2f", ($price * $qty));
92:       $price = sprintf("%.2f", $price);
93:       $grand_total += $total;
94:       print «HTML;
95:         <TR BGCOLOR="#$color">
96:         <FORM METHOD="POST">
97:        <INPUT TYPE="hidden" NAME=" action" VALUE="update">
98:        <INPUT TYPE="hidden" NAME="sku" VALUE="$sku">
99:         <TD ALIGN="left">
100:        $sku
101:        </TD>
102:        <TD ALIGN="left">$name</TD>
103:        <TD ALIGN="center">
104:        <INPUT TYPE=TEXT SIZE="2" NAME="qty" VALUE="$qty">
105:        </TD>
106:        <TD ALIGN="right">\$$price</TD>
107:        <TD ALIGN="right">\$$total</TD>
108:        <TD ALIGN="center"><INPUT TYPE="SUBMIT"
                                VALUE="Обновить"></TD>
109:        </FORM>
110:        </TR>
111:  HTML
112:    }
113:      $grand_total = sprintf("%.2f", $grand_total);
114:      print «HTML;
115:        </TABLE>
116:        <TR><TD ALIGN="RIGHT">
117:        <B>Всего:</B> \$$grand_total
118:        .</TDx/TR>
119:         </TDx/TR></TABLE>
120:       <P><A HREF="cart.cgi">Вернуться в тележку</A>
121:       <P>Чтобы удалить элемент, установите для него
122:         количество 0 и щелкните на кнопке "Обновить"
             около него.
123:       </CENTER></BODY></HTML>
124: HTML
```

```
125: } # Конец подпрограммы View_Cart
126: sub Cet_Product_Info {
127:   my Ssku - shift;
128:   my $dbh = DBI->connect("DBI:mysql:book","book","addison")
129:     or die "Ошибка: $DBI::errstr\n";
130:   my $SQL = "select * from products where sku = ?";
131:   my $sth - $dbh->prepare($SQL);
132:   $sth->execute($sku);
133:   my $p = $sth->fetchrow_hashref;
134:   my $price = $p->{price};
135:  ray $name = $p->{name};
136:   $sth->finish();
137:   $dbh->disconnect();
138:   return ($name, $price);
139: } # Конец подпрограммы Get_Product_Info
140: sub Display_Search_Page {
141:   piint <<HTML;
142:     <HTML><HEAD><TITLE>Поиск изделия</TITLE></HEAD>
143:     <BODY BGCOLOR="#FFFFFF">
144:     <CENTER>
145:     <FORM ACTION="/cgi-bin/book/tie/product_search.cgi"
        METHOD="POST">
146:     <H2>Тележка для покупок на базе связанного хэша</H2>
147:     <TABLE BORDER="1" CELLSPACING="0">
148:     <TR><TD>
149:     <INPUT TYPE="text" NAME="search_for">
150:     </TDxTD>
151:     <INPUT TYPE="submit" VALUE="Начать поиск">
152:     </TD></TR>
153:     </TABLE></FORM>
154:     <P>
155:     <A HREF="/cgi-bin/book/tie/cart.cgi?action=view"
156:     METHOD="POST">Просмотр тележки</A>
157:     </CENTER></BODY></HTML>
158: HTML
159: } # Конец подпрограммы Display_Search_Page
```

## ЛИСТИНГ 12.2. product_search.cgi

```
01: #!/usr/bin/perl -wT
02: # product_search.cgi
03: use strict;
04: use DBI;
05: use CGI qw(:standard);
06: use lib qw(.);
07: use ShopCart;
08: my ($Data, $prod_sth, $vend_sth, Scolor);
09: my $Search_String = param('search_for');
10: my $dbh = DBI->connect("DBI:mysql:book","book","addison"
11:   or die "Ошибка: $DBI::errstr\n";
12: tie Scolor, 'Colors', qw(ffffff eOeOeO);
13: print header;
14: get_products();
15: print_output();
16: sub get_products {
17:   $prod_sth = $dbh->prepare( qq{ SELECT * FROM products
18:                                  WHERE descr LIKE ?
```

```
                                        ORDER BY name });
    $prod_sth->execute("%$Search_String%");
}
sub get_vendor {
  my $vendor = shift;
  $vend_sth = $dbh->prepare( qw{ SELECT vname FROM VENDORS
                                 WHERE vend_num = ? });
  $vend_sth->execute($vendor);
}
sub print_output {
  print <<HTML;
    <HTML>
    <HEAD><TITLE>Результаты поиска изделия</TITLE></HEAD>
    <BODY><CENTER>
    <TABLE BORDER="1" CELLSPACING="0">
    <TR BGCOLOR="#c0c0c0">
    <TD><B>Купить</B></TD>
    <TD><B>Поставщик</B></TD>
    <TD><B>Название</B></TD>
    <TD><B>Описание</B></TD>
    <TD><B>Кол-во на складе</B></TD>
    <TD><B>Цена</B></TD>
    </TR>
HTML
  while ($Data = $prod_sth->fetchrow_hashref) {
    get_vendor($Data->{vend_num});
    my $vendor = $vend_sth->fetch->[0];
    my $cart_link;
    if ($Data->{stock} > 0) {
      $cart_link = qq(<A HREF="cart.cgi?);
      $cart_link .= qq(action=add&sku=$Data->{sku});
      $cart_link .= qq(">Добавить в тележку</A>);
    }
    else {
      $cart_link = qq(Добавить в тележку);
    }
    my $price = sprintf("%.2f", $Data->{price});
    print <<HTML;
      <TR BGCOLOR="$color">
      <TD>$cart_link</TD>
      <TD>$vendor</TD>
      <TD>$Data->{name}</TD>
      <TD>$Data->{descr}</TD>
      <TD>$Data->{stock}</TD>
      <TD>5price</TD>
      </TR>
HTML
  }
  print qq(</TABLE>);
  print qq(<PXA HREF="cart.cgi">Вернуться в тележку</A>);
  print qq(</CENTER></BODY></HTML>);
  $vend_sth->finish;
  $prod_sth->finish;
  $dbh->disconnect;
}
```

**Листинг 12.3. ShopCart.pm**

```
01: use Tie::Hash;
02: use DBI;
03: my $dbh = DBI->connect("DBI:mysql:book","book","addison")
04:   or die "Ошибка: $DBI::errstr\n";
05: my @KEYS;
06: package ShopCart;
07: @ISA = qw(Tie::StdHash);
08: sub STORE (
09:   my (Sself, $key, $val) = @_;
10:   my $modified = time();
11:   my $session = main::Session();
12:   my Sexisted - $self->EXISTS($key); # Ключ уже существует?
13:   my Saction = main::param('action');
14:   my $new_qty;
15:   unless ($val) {
16:     $self->DELETE($key);
17:     return;
18:   }
19:   if ($existed) {
20:     if (Saction eq "update") {
21:       $new_qty = $val;
22:     } else {
23:       $new_qty = $existed + $val;
24:     }
26:     my $sth = $dbh->prepare( qq{ UPDATE cart
27:                                  SET qty = '$new_qty'
28;                                  WHERE session = ?
29:                                 AND sku = ? });
30:     $sth->execute($session, Skey);
31:   }
32:   else {
33:     my $sth = $dbh->prepare( qq{ INSERT INTO cart
34:                                  VALUES (?,?,?,?) }};
35:     $sth->execute($key, $val, $modified, $session);
36:   } # Конец блока if...else
37: } # Конец метода STORE
38: sub EXISTS {
39:   my($self, $key) = @_;
40:   my Ssession = main::Session();
41:   my Ssth = $dbh->prepare( qq{ SELECT qty FROM cart
42:                                WHERE session = ?
43:                               AND sku = ? }};
44:   $sth->execute($session, Skey);
45:   my $temp = $sth->fetch;
46:   return $temp->[0] ? $temp->[0] : 0;
47: } # Конец метода EXISTS
48: sub DELETE {
49:   my (Sself, $key) - @_;
50:   my Ssession = main::Session();
51:   my Ssth = $dbh->prepare( qq{ DELETE FROM cart
52:                                WHERE session = ?
53:                               AND sku - ? });
54:   $sth->execute($session, $key);
55: } # Конец метода DELETE
56:   sub FETCH {
57:   my (Sself, Skey) = @—;
```

```
58:     my $session = main::Session();
59:    my $sth = $dbh->prepare( qq{  SELECT qty FROM cart
60:                                    WHERE session = ?
61:                                  AND sku = ? });
62:    $sth->execute($session, $key);
63:    my $temp = $sth->fetch;
64:    return $temp->[0];
65:  } # Конец метода FETCH
66:  sub FIRSTKEY {
67:    my $self = shift;
68:    my Ssession = main::Session();
69:    my $sth = $dbh->prepare( qq{  SELECT qty, sku
70:                                    FROM cart
71:                                  WHERE session = ? });
72:    $sth->execute($session);
73:    $self->{DATA} = $sth->fetchall_arrayref;
74:    $self->{INDEX} = 0;
75:    my $val = $self->{DATA}->[0][0];
76:    my Skey - $self->{DATA}->[0][1];
77:    return if !@{$self->{DATA}};
       # Нет данных - возврат из метода.
78:    foreach(@{$self->{DATA}}) {
79:      my $key = $_->[1];
80:      my $val = $_->[0];
81:      $self->{LIST}{$key} = $val;
82:    }
83:    @KEYS = keys ( %{$self->{LIST}} );
84:    $self->{FIRSTKEY} = $key;
85:    return ($val, $key) ;
86:  } # Конец метода FIRSTKEY
87;  sub NEXTKEY {
88:    my $self = shift;
89:    my $lastkey = shift;
90:    $key = $KEYS[$self->{INDEX}++];
91:    next if ($key eq $self->{FIRSTKEY});
92:    return($key) ;
93:  } # Конец метода NEXTKEY
94:  sub DESTROY {
95:    $dbh->disconnect() ;
96:  } # Конец метода DESTROY
97:  package Colors;
98:  use Tie::Scalar; .
99:  sub TIESCALAR {
100:    my {$class, ©values) = @_;
101:    bless \@values, $class;
102:    return \@values;
103:  }
104:  sub FETCH {
105:    my $self - shift;
106:    push(@$self, shift(@$self));
107:    return $self->[-1];
108:  }
109:  1;
```


# Внедрение Perl в HTML с помощью Mason

## Введение

Mason — это инструмент для внедрения кода Perl в документы HTML. Он позволяет создавать сайты на основе компонентов. Mason работает на Web-сервере Apache с установленным `mod_perl`. Он дает возможность строить мощные, динамичные Web-сайты, затрачивая минимум усилий.

В Mason используется "компонентная" архитектура, которая состоит в том, что разработчик создает небольшие фрагменты кода Perl, внедренные в HTML, а затем формирует Web-страницу из этих компонентов. Компоненты в Mason могут вызывать другие компоненты и т.д. Например, можно создать компонент для заголовка, для верхнего колонтитула, нижнего, пару компонентов для навигации и один компонент для тела страницы. Затем мы собираем из этих компонентов Web-страницу. В Mason имеется так называемый компонент-автообработчик (autohandler), который применяет определенный стиль ко всем страницам в каталоге. Также существует возможность создать свое специальное сообщение об ошибке вместо раздражающего стандартного "Error 404".

Mason обладает множеством привлекательных возможностей. Недавно Брент применил Mason при разработке нового проекта и был совершенно восхищен им. Этот новый проект должен был иметь одну новую особенность — "совместный брэндинг". Это означает, что дизайн сайта может изменяться в зависимости от желания его владельца, тогда как функциональные возможности сохраняются. Mason дает очень простое решение для совместного брэндинга.

Код HTML, который Брент получил от компании-разработчика, требовал значительного редактирования. Разработчик применял редактор HTML, который "не признает" пробелы, и поэтому код получился совершенно нечитаемым. Потребовался це-

лый день, чтобы привести HTML в "читабельный" вид. Но затем для обработки всех страниц был применен подход Mason. Менее чем через час весь сайт, состоящий примерно из 20 страниц, был уже готов. Как только общая схема была закончена, введение в нее других страниц стало очень простым делом.

Несмотря на то, что Mason предоставляет весьма простые средства, это чрезвычайно мощный инструмент, так как он позволяет внедрять Perl прямо в HTML. Но хватит рекламы! Mason хорошо делает именно ту работу, для которой он предназначен. Для примера мы возьмем переделку Web-сайта. Мы выбрали для переделки сайт `http://www.perlguy.net` по вполне определенным причинам: он требовал обновления, производил не очень сильное впечатление и нуждался в большем количестве функций.

# Инсталляция

Установка `mod_perl` и Mason — не особенно сложные задачи, но вы обязательно должны прочитать документацию по установке. Здесь мы не описываем инсталляцию `mod_perl` или Mason, так как в документации об этом рассказано очень подробно. Кроме того, документация дает наиболее свежую информацию, какую только можно найти. Для работы Mason нужен как минимум Web-сервер Apache, на котором работает `mod_perl` и установлен модуль Per] HTML: :Mason.

# Стратегия

Когда Брент приступил к созданию нового сайта, первое, что он сделал — нарисовал черновую схему, или эскиз сайта. Схема типичной страницы включала верхний и нижний колонтитулы, тело страницы и, для главной страницы, несколько блоков новостей. Имея такую схему, можно подумать о стратегии, которой она будет реализована. Так как Бренту не нравятся фреймы, естественным выбором для размещения элементов на страницах становится таблица. Теперь, когда у нас есть схема и выбран способ размещения, целесообразно рассмотреть основы синтаксиса Mason.

# Синтаксис Mason

Мы не будем рассказывать обо всем синтаксисе Mason. Одно это может занять целую книгу. Однако мы опишем достаточно, чтобы вы смогли начать самостоятельную работу. Единственную трудность может создать методология, которую использует Mason. Если вы долгое время занимались разработкой Web-сайтов и привыкли к обычной методике CGI, Mason заставит вас посмотреть на вещи немного по-иному.

Создание страниц с помощью Mason не требует привлечения множества операторов `print` и включенных документов. Вместо этого перед выполняемым кодом просто ставится один из дескрипторов Mason, а все остальные данные обрабатываются как стандартный HTML. Не нужно помещать скрипты в каталог `cgi-bin`, так как Mason сам обрабатывает весь код, подлежащий обработке.

## Компонент

Первое понятие, которое в дальнейшем будет часто использоваться, — это *компонент*. Компонент — это просто файл, содержащий HTML, код Perl или и то и другое вместе. Так как Mason работает со множеством небольших частей, компонент — наилучший термин, которым можно обозначить эти части.

Компоненты дают возможность воспользоваться всеми выгодами многократного использования кода. Компонентами могут быть самостоятельные фрагменты кода,

выполняющие определенные задачи, или даже группы из нескольких простых дескрипторов HTML. Компоненты не должны быть большими; несколько малых компонентов, выполняющих определенные задачи, дают больший выигрыш, чем один большой компонент, который пытается делать все сам.

Если над сайтом работает несколько человек, компоненты значительно облегчают распределение работы между ними. Каждому программисту можно назначить отдельный компонент, над которым он будет работать, что позволяет избежать многих проблем, обычно возникающих при групповой разработке.

## <%perl>.. .</%perl>

Дескриптор <%perl> позволяет включить в компонент большой блок кода Perl. Все, что находится между открывающим и закрывающим дескрипторами, рассматривается как обычный код Perl. Как можно заметить, этот дескриптор, как и большинство дескрипторов Mason, очень похож на HTML.

## %

Строки, которые начинаются со знака %, также рассматриваются как код Perl. Знак % должен быть в начале строки, иначе строка будет считаться обычной строкой HTML. Этот дескриптор целесообразно применять, когда требуется вставить в документ пару строк кода Perl, не формируя для них блок <%perl>.

## <%...%>

Синтаксис <%...%> дает возможность помещать код Perl непосредственно в строку HTML. Когда анализатор Mason замечает дескриптор <% %>, он выполняет то, что находится внутри него. Если это скалярная переменная, содержащая строку, это строковое значение появляется вместо переменной в конечном HTML. Если это вызов функции, например, localtime, отображается значение, возвращаемое функцией. Этот дескриптор при создании страниц в Mason очень часто забывают ставить. Если просто поместить переменную в строку, не заключив ее в дескриптор <% %>, например:

```
<center>$title<center>
```

то мы получим в броузере текст "$title", выровненный по центру. Нужно написать так:

```
<center><% $title %><center>
```

Тогда отобразится то, что мы ожидали: выровненное по центру значение переменной $title.

Пробелы между % и именем переменной необязательны, но этот прием делает код более легким для чтения. Запись <% $variable %> распознается легче, чем <%$variable%>. Пробелы не выводятся в броузере, поэтому их можно применять без опасения.

## <%init>...</%init>

Блок <%init> выполняется сразу же после вызова компонента. Это позволяет инициализировать переменные и выполнять код до того, как будет отображен какой-либо HTML в компоненте. Блок <%init> действует так же, как и блок <%perl>, но последний выполняется в тот момент и в той точке, где он встречается в компоненте, тогда как <%init> выполняется в первую очередь, даже если он находится в конце компонента.

Обычно принято помещать блоки <%init> в конце компонентов. Таким образом, когда в HTML надо внести изменения, тот, кто их делает, может не вникать в код Perl, который может быть ему незнаком.

### <%cleanup>...</%cleanup>

Эта пара дескрипторов создает блок кода, который выполняется непосредственно перед завершением компонента. Необходимость в этом возникает нечасто, так как Perl сам достаточно хорошо выполняет операции очистки. Но в случае, если требуется что-то сделать в самом конце компонента, подойдет именно этот блок.

### <%once>...</%once>

Код в этом блоке выполняется один раз при загрузке компонента. Все переменные, созданные в нем, будут доступны на протяжении существования компонента. Mason кэширует используемые компоненты. Блок <%once> позволяет выполнять код, который в нем содержится, только один раз, в родительском процессе, после чего результат выполнения будет доступен во всех дочерних процессах.

### <%shared>...</%shared>

Блок <%shared> содержит код (например, переменные), который должен быть включен во все компоненты, а также подкомпоненты.

### <%def name>...</%def>

Этот дескриптор позволяет определять подкомпонент внутри компонента. Подкомпонент подобен именованной подпрограмме в Perl, a name — имя, по которому он вызывается. Внутри блока <%def name> можно применять дескрипторы Mason, за исключением <%def>, <%method>, <%once> или <%shared>.

Область действия блока <%def name> отличается от области действия компонента, который его содержит. Его объекты можно назвать частными: они могут быть вызваны только из компонента, в котором находится этот блок. Иными словами, нельзя создать подкомпонент в одном компоненте и вызвать его из другого.

### <%method name>...</%method>

Методы определяются так же, как и блоки <%def name>. От подкомпонентов методы отличаются тем, что их можно вызывать из других компонентов. Методы имеют больше сходства с подпрограммами в Perl.

### <%attr>...</%attr>

Блок <%attr> позволяет создавать пары ключ-значение, которые могут считываться другими компонентами.

### <%filter>(регулярное_выражение)</%filter>

Этот блок позволяет задавать фильтр для вывода компонента, в котором он находится.

Это может показаться бесполезным: зачем выводить данные, а затем еще и фильтровать их? В последующем примере мы покажем фильтр, который проверяет, какие страницы имеются в наличии, и на этом основании автоматически обновляет состав меню.

### <%doc>...</%doc>

Этот блок позволяет включать в комментарий большие блоки текста. Mason игнорирует комментарий; в конечном документе HTML комментарий также не отображается.

### <%text>...</%text>

Этот дескриптор исключает блоки текста из сферы действия Mason. Это дает возможность создавать блоки, содержащие дескрипторы Mason, которые не будут обрабатываться в нем, а просто отобразятся.

### <%args> ... </%args>

Раздел <%args> служит для создания списка параметров для передачи в компонент, а также для установки значений по умолчанию.

Если в блоке <%args> находится переменная без значения по умолчанию и в компонент не передается ничего, генерируется ошибка. Если здесь указано значение по умолчанию, то это значение используется, когда в компонент ничего не передается..

Блоки <%args> очень полезны для форм HTML. Например, если в форме есть текстовое поле под названием first_name, в блоке <%args> можно поместить код $first_name, если это поле обязательно для заполнения, или $first_name=>'foobar', и тогда foobar станет значением по умолчанию.

Это почти все дескрипторы, с которыми вы должны быть знакомы. Mason постоянно развивается, так что не удивляйтесь, если заметите новые дескрипторы. В то же время Mason остается надежной и полезной технологией создания Web-сайтов.

# Специальные компоненты Mason

Mason содержит некоторые "специальные" компоненты, которые автоматически выполняют свою задачу, когда происходят определенные события.

### autohandler

Компонент autohandler (автообработчик) выполняется перед выполнением любого компонента. Этот компонент работает в масштабе всего каталога, т.е. обслуживает все файлы в каталоге *и* подкаталоги. Автообработчики очень полезны, если требуется поместить колонтитулы на все страницы сайта. Подобный автообработчик мы используем в нашем примере.

### dhandler

Компонент dhandler — заданный по умолчанию обработчик (default handler), который вызывается, если не найдено никаких компонентов, соответствующих запросу в URI. Обработчики по умолчанию позволяют создавать специальные сообщения об ошибках или обрабатывать ситуации, когда запрашиваемый URL нельзя найти. Чтобы создать обработчик по умолчанию, достаточно присвоить компоненту имя dhandler.

### $m

Компонент $m представляет объект запроса в Mason. В нем имеется API, предоставляющий доступ ко всем функциям Mason, не охватываемым дескрипторами. Компонент $m доступен для всех компонентов.

# Каскадное выполнение

Когда вы запрашиваете документ HTML с сервера Mason, этот документ собирается каскадным способом. Здесь мы имеем в виду не каскадные таблицы стилей, а каскады компонентов, из которых создается документ. Когда требуется компонент верхнего уровня, например, index.html, система в первую очередь выясняет, существует ли такой файл. Если он имеется, система проверяет, есть ли в текущем каталоге файл автообработчика. Если автообработчик найден, выполняется его код до вызова метода $m->call_next. ДО этого момента фактически запрашиваемый файл (index.html) еще не выполняется. Затем система обходит все строки запрашиваемого компонента и вызывает все компоненты, которые в нем вызываются. Эти компоненты в свою очередь могут вызывать другие компоненты и т.д.

Когда обработка компонента верхнего уровня (запрашиваемого компонента) будет завершена, сервер возвращается к компоненту автообработчика и выполняет весь оставшийся код. Именно это облегчает добавление колонтитулов. Так как автообработчик выполняет свой код до метода call_next, затем запрашиваемый компонент и, наконец, остаток своего кода, компонент оказывается "окруженным" кодом автообработчика. Допустим, что пользователь запрашивает файл foo.html. Процесс должен идти примерно так:

Пользователь:

запрос foo.html

Сервер:

автообработчик (все до $m->call_next)

foo.html

автообработчик (все после $m->call_next)

Это формирование "сэндвича" из компонентов происходит очень быстро. Многие компоненты кэшируются, что еще более ускоряет выполнение. Если компоненты в свою очередь вызывают другие компоненты, картина усложняется, и можно увидеть, как система проходит по каскадам различных компонентов и формирует запрашиваемый документ.

# Продолжаемдвижение

Мы вкратце рассмотрели синтаксис Mason, а затем перейдем к созданию сайта с использованием Mason. Хотя некоторые концепции мы еще не изложили, о них будет рассказано по ходу написания кода, чтобы вы могли увидеть все в действии.

Начнем с нового файла index.html (см. листинг 13.1). Главный файл, который вызывается пользователем, называется *компонентом верхнего уровня*. Этот компонент соединяет вместе все компоненты, образующие запрашиваемую страницу. Обычно компонент верхнего уровня — это тот файл HTML, на который указывает URL.

ЛИСТИНГ 13.1. index.html

```
<& menu_start &>
  <& rss2html, site=>"perl-news" &>
  <& rss2html, site=>"perl.com" &>
<& menu_end s>
<& body_start &>
  <& my_news s>
  <& pictures &>
  <& links &>
<& body_end &>
```

Больше в файле index.html ничего нет. Тем не менее, у нас получилась довольно хорошая страница, учитывая, что в файле index.html было только девять строк! Как это стало возможно? Конечно, с помощью Mason!

Вспомните, что Mason позволяет собирать страницу HTML из компонентов. В файле index.html вызывается восемь компонентов (один используется дважды). Давайте просмотрим компоненты и узнаем, что они делают. Некоторые из них очень просты, но зато другие довольно сложны.

Процесс начинается с того, что сервер пытается открыть запрашиваемую страницу. Поскольку index.html действительно существует, мы можем продолжить выполнение. Так как Mason установлен, он первым будет обрабатывать все запросы и при необходимости передавать их в соответствующие обработчики. Затем Mason проверяет существование файла, который считается автообработчиком. Если файл автообработчика сущест-

вует, *прежде* файла index.html выполняется код в автообработчике. На этом сайте автообработчик применяется, поэтому мы сначала рассмотрим его, так как этот файл выполняется в первую очередь. Скрипт автообработчика также очень невелик. Можно заметить, что некоторые компоненты используются на сайте по нескольку раз. Возможность многократного использования — ключевая характеристика Mason.

```
01:  <HTML><HEAD><TITLE><& SELF:title &></TITLE></HEAD>
```

В строке 1 создается начальный код HTML для страниц. Директива `<& SELF:title &>` нам еще незнакома, хотя она тоже относится к Mason. Как и при обычном вызове подпрограммы в Perl (`&имя_подпрограммы`), знак *&* в Mason означает вызов функции или внешнего компонента. Здесь мы вызываем метод *title* текущего компонента. Автообработчик содержит метод под названием *title* (в строках **11–13**), но если ваш компонент верхнего уровня также имеет метод title, он имеет преимущество над автообработчиком. Этот механизм очень полезен, так как в автообработчике можно создать стандартный заголовок, а в компоненты верхнего уровня поместить специфические заголовки для определенных страниц. Ключевое слово SELF означает, что мы вызываем метод title из данного компонента (автообработчика), а не из компонента верхнего уровня.

```
02:  <BODY BGCOLOR="<% $color %>">
```

В строке 2 устанавливается цвет фона страницы HTML. Переменная $color была инициализирована в блоке `<%init>` в строках 14—16. То, что находится внутри дескриптора `<% %>`, будет обработано как Perl, и выведен результат. Таким образом, если здесь поместить переменную, будет выведено ее значение. Например, если переменной было присвоено выражение `10-3`, будет выведено 7.

```
03:  <CENTER>
04:  <img src="/images/new_header.gif" width="475" height="74">
05:  </CENTER>
06:  <HR WIDTH=85%>
07:  <center>
08:  <table width="100%" border="0" cellspacing="0"
       cellpadding="0">
```

Строки 3–8 — просто код HTML, который передается в броузер безо всяких изменений.

```
09:  <% $m->call_next %>
```

Строка 9 особого рода. Раньше нам не встречалось что-то подобное. Вспомните, что $т — объект запроса Mason, доступный для всех компонентов. Этот объект предоставляет API для функций Mason, недоступных через синтаксические дескрипторы. Метод *$m->call_next()* указывает, что надо вызвать следующий компонент в цепочке. Фактически это означает вызов запрашиваемого файла HTML, в данном случае — index.html. Как только обработка файла HTML будет закончена, мы вернемся к строке, следующей за *$m->call_next()*. Это позволяет создавать "обертку" для всех страниц простым использованием автообработчика.

```
10:  <& footer &>
```

В строке 10 вызывается компонент нижнего колонтитула. Этот компонент содержит HTML, завершающий страницу. В нашем примере внизу страницы помещается список ссылок.

```
11:  <%method title>
12:    Perlguy's Pages
13:  </%method>
```

Строки 11—13 образуют метод title. Методы подобны подпрограммам в Perl. Данный метод просто возвращает стандартный заголовок для всех страниц.

```
14:  <%init>
15:    my $color = "#f1edd3";
16:  </%init>
```

**Строки 14—16** — блок `<%init>`. Здесь инициализируются все переменные, которые могут нам понадобиться в компоненте автообработчика или в любом другом компоненте. Здесь можно было бы поместить переменную `$title` со стандартным заголовком, но тогда у нас не оказалось бы примера для методов!

Итак, компонент автообработчика завершен. Он содержит всего 16 строк, но эти строки создают "обложку" для всех страниц сайта. Разработчик может просто генерировать страницу HTML, не заботясь о коде для колонтитулов — страницы автоматически получат нужные верхний и нижний колонтитулы.

## index.html

Теперь мы находимся в строке 9 компонента автообработчика и готовы приступить к отображению файла index.html.

**Строка 1** index.html вызывает компонент menu_start. Код этого компонента приведен в листинге 13.2.

### Листинг 13.2. Компонент menu_start

```
<tr>
<td width="15%" valign="top">
```

Вот и все, только две строки! Как мы говорили выше, компоненты не должны быть большими. Код этого компонента не требует пояснений.

В **строках** 2—3 index.html вызывается компонент `rss2html` и в него передается ключ site и связанное с ним значение. Обо всех упоминаемых компонентах мы расскажем сразу после того, как закончим index.html.

**В строке 4** вызывается компонент `menu_end`. Этот компонент также очень невелик; он содержит только дескриптор <td>.

Но зачем создавать целый компонент для одного-единственного дескриптора? Ведь даже чтобы вызвать его, приходится писать больше кода, чем он содержит! Ответ **прост**: *гибкость*. Если вы решите внести изменения в схему вашего сайта, намного проще будет изменить один компонент, чем производить те же изменения в нескольких файлах **HTML**.

**В строке** 5 вызывается компонент `body_start`. Этот компонент также очень мал **и** содержит лишь две строки HTML.

```
<td width="85%" valign="top">
<center>
```

Этот компонент создает таблицу, в которой будет размещаться тело документа.

**Строки 6—8** содержат вызовы компонентов, которые генерируют три основных блока страницы. Все эти строки не требуют пояснений; их очень легко изменить или удалить.

Строка 9 вызывает компонент body_end. И этот компонент не отличается сложностью: в нем всего три строки HTML.

```
</center>
</td>
</tr>
```

Вот и все содержание файла index.html.

Затем управление снова переходит к компоненту автообработчика, который помещает внизу страницы меню и список ссылок. О нем даже не стоит рассказывать. Mason сам выполняет все, что нужно, так что программисту не надо даже помнить о том, что надо поместить на страницу.

# rss2html

Теперь мы рассмотрим компонент rss2html. Этот замечательный компонент принимает файл RSS/RDF и преобразует его в красивый формат. Файлы RSS/RDF — это файлы XML в стандартном формате для распространения новостей и информации. Код компонента `rss2html` взят из модуля `XML::RSS`, разработанного Джонатаном Айзенцопфом (Eisenzopf), с небольшими изменениями.

В исходной версии `rss2html` использовался модуль `LWP::Simple` и файл RSS/RDF обрабатывался при каждом щелчке на странице. Это очень медленный способ, так как для каждого требуемого раздела надо было вышавать отдельный запрос и получать документ XML снова.

Вместо этого мы создали небольшой скрипт, который извлекает документы RSS/RDF и сохраняет их в базе данных MySQL. Скрипт выполняется как событие планировщика (cron event), поэтому для получения содержимого больше не требуется никаких действий. Переход к версии с базой данных значительно уменьшает время загрузки страницы и снижает нагрузку на сайты, которые предоставляют файлы RSS/RDF.

Рассмотреть компонент rss2html будет очень полезно, поскольку он демонстрирует, как, объединив код Perl и HTML при помощи Mason, можно добиться замечательных результатов.

```
01: <%perl>
02: $sth->execute($site);
03:  $content = $sth->fetch->[0];
04:  # Анализ содержимого RSS
05:  $rss->parse($content);
06: </%perl>
```

**Строка 1** начинает блок `<%perl>`. Все, что находится в этом блоке, вплоть до закрывающего дескриптора, будет обработано как код Perl.

**Строка 2** выполняет команду SQL в базе данных MySQL. Это может показаться странным, ведь код подключения к базе данных и сама команда SQL находятся в конце файла. Но в Mason принято помещать в конце компонента как можно большую часть кода Perl. В результате файл становится более понятным для Web-разработчика, Он не должен разбирать неизвестный ему код и может сосредоточиться на самом HTML.

**В строке 3** для базы данных вызывается метод *fetch()*, который возвращает первый элемент найденного массива данных. Мы извлекаем только один элемент, так что здесь вполне допустимо жестко задать 0-й индекс массива.

**Строка 4** — просто комментарий.

В **строке** 5 вызывается метод `parse()` модуля XML: :RSS. ЭТОТ метод анализирует содержимое файла RSS/RDF и загружает данные в структуру, с которой Perl легче работать.

**Строка 6** закрывает блок `</%perl>`.

```
07: <table bgcolor="#996600" border="0" width="200">
08: <tr>
09: <td>
10: <table cellspacing="1" cellpadding="4" bgcolor="#FFFFFF"
11: border=0 width="100%">
12: <tr>
13: <td valign="middle" align="center" bgcolor="#cc9900">
14: <font color="#000000" face="Arial,Helvetica">
15: <b>
```

**Строки 7—15** — просто HTML без внедренного кода.

```
16:  <a href="<% $rss->{'channel'}->{'link'} %>">
       <% $rss->('channel'}->{'title'} %></a>
```

Строка 16 занимает две строки листинга. В ней формируется ссылка на канал. Как можно заметить на Web-странице, заголовок каждого блока RSS/RDF является ссылкой. Эта ссылка создается именно здесь. Также в этой строке можно увидеть два блока <% %>. Они образуют внедренные фрагменты Perl; на их Жесте будет отображен результат выполнения этого кода.

```
17:  </b>
18:  </font>
19:  </td>
20:  </tr>
21:  <tr>
22:  <td align="center">
```

**Строки 17—22** — снова HTML без внедренного кода.

```
23:  %  # Вывод изображения для канала
24:  %  if ($rss->{'image'}->{'link'}) {
```

**Строка 23** — комментарий. Обратите внимание, что строка начинается со знака %. Поэтому все в строке воспринимается как Perl, даже комментарий. На комментариях можно не экономить, так как все, что рассматривается как код Perl, не появится в конечном документе HTML.

**В строке 24** проверяется, существует ли ссылка на изображение. Если это так, начинается блок кода.

```
25:        <a href="<% $rss->{'image'}->{'link'} %>"><img
26:          src="<% $rss->{'image'}->{'url'} %>"
27:          alt="<% $rss->{'image'}->{'title'} %>" border="0"
```

**В строке 25** формируется гиперссылка для изображения и начинается дескриптор IMG.

**В строке 26** устанавливается атрибут SRC дескриптора IMG.

**В строке 27** устанавливается атрибут ALT дескриптора IMG.

```
28:  %     if ($rss->{'image'}->{'width'}) {
29:          width="<% $rss->{'image'}->{'width'} %>"
30:  %    }
```

**В строке 28** проверяется, была ли вместе с изображением передана его ширина. Если ширина была указана, в **строке 29** создается атрибут WIDTH, а **строка 30** закрывает блок if.

Обратите внимание, что строки 28 и 30 начинаются со знака %, а строка 29 — нет. Несмотря на это, она пропускается, если условие в строке 28 оказалось ложным, так как она находится внутри блока if.

Строка 29 рассматривается как код HTML, в который внедрен код Perl.

```
31:  %    if ($rss->{'image'}->{'height'}) {
32:          height="<% $rss->{'image'}->{'height'} %>"
33:  %    }
```

Строки 31—33 аналогичны строкам 28—30, только здесь устанавливается атрибут ВЫСОТЫ (HEIGHT).

```
34:    ></a>
35:  % }
```

**Строка** 34 завершает ссылку **HTML.**

**Строка** 35 завершает блок if, начатый в строке 24.

```
36:  </td>
37:  </tr>
38:  <tr>
39:  <td><font face="arial,helvetica" size="2">
40: <p>
```

**Строки 36—40** — просто HTML.

```
41:  %  # Вывод элементов канала
42:  %    foreach my $item (@{$rss->{'items'}}) t
43:  %      next unless defined($item->{'title'}) &&
             defined($item->{'link'});
44:          <li><a href="<% $item->{'link'} %>">
               <% $item->{'title'} %></a><BR>
45:  %   }
```

**Строка 41** — комментарий.

**В строке 42** начинается цикл *foreach ()* по всем элементам массива @f$rss->{ 'items'}}.

**В строке 43** мы переходим к следующей итерации цикла, если заголовок и ссылка элемента не определены..

**В строке 44** создается ссылка на сообщение.

**Строка 45** завершает цикл *foreach ()*, начатый в строке 42.

```
46:  %  # Есть ли здесь поле ввода текста?
47:  %  if ($rss->{'textinput'}->{'title'}) {
```

**Строка 46** — комментарий.

В **строке 47** проверяется, есть ли в документе RSS поле ввода текста. Если да, мы должны создать на странице форму с соответствующим полем.

```
48:   <form method="get"
        action="<% $rss->{'textinput'}->{'link'} %>">
49:  <% $rss->{'textinput'}->{'description'} %><br />
50:  <input type="text"
       name="<% $rss->{'textinput'}->{'name'} %>"><br />
51:  <input type="submit"
       value="<% $rss->{'textinput'}->{'title'} %>">
52:  </form>
```

**В строке 48** создается форма HTML и для атрибута action устанавливается значение, хранящееся в $rss->{ 'textinput'}->{ 'link'}.

В **строке** 49 определяется и выводится описание для поля формы, а затем выводится дескриптор разрыва строки.

**В строке 50** в форму помещается само поле ввода текста.

В строке 51 создается кнопка отправки формы и выводится ее название.

Строка **52** завершает форму HTML.

**Строки 48—52** выглядят довольно сложными, так как кажется, что здесь выполняется очень много действий. Но давайте взглянем на эти строки, убрав все, что относится к Mason.

```
48:  <form method="get" action="имя_ссылки">
49:  Заголовок поля<br />
50:  <input type="text" name="имя_поля"><br />
51:  <input type="submit" value="текст_кнопки">
52:  </form>
```

Здесь мы только заменили коды Mason их возможными значениями, и текст стал гораздо проще и понятнее. Он выполняет такие же действия, что и приведенный выше вариант, и все описания можно применить и к нему.

```
53:  %  }
54:   %  # Если есть элемент копирайта
55:  %  if ($rss->{'channel'}->{'copyright'}) {
56:        <pxsub><% $rss->{'channel'}->{'copyright'} %>
             </sub></p>
57:  %  }
```

**Строка** 53 завершает блок if, начатый в строке 47. Благодаря ему поле формы выводится, только если оно существует.

**Строка 54** — комментарий.

**Строка** 55 начинает блок if, в котором проверяется, связана ли с данным файлом RCC какая-то информация копирайта. Если это так, начинается блок кода.

**В строке 56** выводится информация копирайта.

**Строка 57** завершает блок if.

```
58:  </fontx/td>
59:  </tr>
60:  </table>
61:  </td>
62:  </tr>
63:  </table>
```

**Строки 58-63 — просто HTML, который направляется в броузер.**

```
64:  <%args>
65:  Ssite
66:  </%args>
```

**Строки 64—66** — блок **<%args>**, содержащий переменные, которые будут переданы в компонент при его вызове. Так как в **строке 65** не задано значение по умолчанию, произойдет ошибка, если в компонент не будет передано значение `$site`.

В файле `index.html` **есть строки вида** `<& rss2html, site=>"perl.com" &>`, в которых вызывается компонент `rss2html` и в него передается имя сайта как параметр. Имя передаваемого параметра должно соответствовать имени, указанному в блоке **<%args>**.

```
67:  <%once>
68:  my $dbh;
69:  $dbh  = DBI->connect("DBI:mysql:book","book","addison")
70:     | 1 die "Нельзя подключиться: $DBI::errstr\n"
       unless $dbh;
71:  my $sth  = $dbh->prepare(
       qq{SELECT data FROM rss WHERE site = ?});
72:  </%once>
```

**Строки 67—72** — блок **<%once>**. Код в этом блоке выполняется только один раз и сохраняется в течение существования компонента.

**В строках 69—70** создается дескриптор базы данных. Этот дескриптор не исчезнет, когда выполнение модуля будет завершено; он будет храниться в памяти, пока будет существовать компонент. Такой механизм ускоряет работу, так как не надо каждый раз создавать новое подключение и дескриптор базы данных.

**Строка** 71 создает и подготавливает команду SQL. Чтобы не повторять и эти операции каждый раз, мы также помещаем их в блок <%once>.

Вспомните, что компоненты функционируют иначе, чем обычные программы CGI. Когда вызывается обычная программа, запускается интерпретатор Perl, программа интерпретируется и выполняется, затем закрывается, и при следующем вызове программы весь процесс повторяется сначала. В приложении Mason интерпретатор Perl работает постоянно и компоненты кэшируются, что еще более ускоряет выполнение. Поэтому, когда пользователь завершает скрипт, этот скрипт может еше оставаться в памяти и ждать следующего использования.

```
73:  <%init>
74:  my $content;
75:  my $£ile;
76:  my $rss;
```

В строке 73 начинается блок <%init>. Этот блок служит для объявления переменных и выполнения кода непосредственно после вызова компонента.

В строках 74—76 объявляется несколько переменных, которые будут использоваться компонентом.

Компоненты Mason требуют строгого синтаксиса, поэтому любая используемая переменная предварительно должна быть объявлена с ключевым словом my.

```
77:  $rss = new XML::RSS;
78:  </%init>
```

В строке 77 создается новый экземпляр модуля XML: :RSS и его дескриптор сохраняется в переменной $rss. Хотя мы используем только один экземпляр XML: :RSS, необходимо создавать новый экземпляр при каждом вызове компонента. Если не сделать этого, дескриптор будет использован повторно и нам придется пройти заново весь код, который уже был сгенерирован ранее.

Строка 78 закрывает блок <%init> и завершает компонент. Каскадный способ формирования страниц в Mason не позволяет легко перечислить компоненты в том порядке, в котором они вызываются. Поэтому похоже, что мы несколько упростили реальную картину. Мы — да, но не Mason, он делает все в точности как положено!

# my_news

Компонент my_news считывает текстовый файл, содержащий новости и информацию, форматирует его и отображает. На главной странице он появляется вверху в виде блока с заголовком "My News".

```
01:  <& wrap_top, width=>'100% ' &>
```

В строке 1 вызывается компонент wrap_top и в него передается желаемая ширина "обрамляющей таблицы".

```
02:  <table cellspacing="1" cellpadding="4" border="0"
       bgcolor="#ffffff"
03:  width="100%">
04:  <tr bgcolor="#cc9900">
05:  <td valign="top" align="left">
06:  <font face="Arial,Helvetica" size="5">
07:  <b>My News</b>
08:  </font>
09:  </td>
10:  </tr>
11:  <tr>    • •
12:  <td><font face="arial,helvetica" size="3">
13:  <p>
```

Строки 2—13 — просто HTML, генерирующий красивое обрамление для сообщений.

```
14:  % my $news = $m->file('news_text');
```

Строка 14 демонстрирует код, который нам еще не встречался. Как мы знаем, `$m` — это объект запроса Mason, и он содержит API для доступа к некоторым функциям Mason, не представленным дескрипторами. Метод `$m->file(имя_файла)` считывает данные из файла и возвращает их как строку. Не требуется открывать файл или закрывать его; достаточно убедиться, что файл существует, иначе произойдет ошибка.

Итак, эта строка читает содержимое файла и сохраняет его в переменной $news. Имя `news_text` — просто текстовый файл, содержащий HTML для текстовой области блока "My News".

```
15:  <% $news %>
```

В строке 15 выводятся данные, прочитанные в файле `news_text`.

```
16:  </p>
17:  </td>
18:  </font>
19:  </tr>
20:  </table>
```

Строки **16–20** завершают таблицу HTML для внутренней части блока "My News".

```
21:  <& wrap_bottom &>
```

В **строке** 21 вызывается компонент `wrap_bottom`. Этот компонент завершает внешнюю таблицу, обрамляющую весь блок "My News". Этот код также относится к компоненту `my_news`. Компоненты, предназначенные для создания раздела ссылок и раздела изображений, в основном имеют такую же структуру, что и данный файл, только HTML немного отличается.

# footer

Компонент `footer` — последний компонент, который мы рассмотрим подробно. Он демонстрирует хороший пример использования дескриптора `<%filter>`.

```
01:  </tr>
02:  <tr>
03:  <td colspan="2" align="center">
04:   <brxbr>
05:  <center>
06:  <hr width="85%">
07:   <brxbr>
08:   <font face=arial size=2>
09:  [
10:     <a href="/index.html">Home</a> |
11:     <a href="/articles.html">Articles</a> |
12:     <a href="/sql/index.html">DBI/SQL Tutorial</a> |
13:     <a href="/services.html">My Services</a> |
14:     <a href="/campcamel/index.html">Camp Camel</a> |
15:     <a href="/bbq/index.html">BBQ Anyone?</a> |
16:     <a href="/tattoo/Index.html">Camel Tattoo</a>
17:  ]
18:  </font>
19:  <br><br>
20:  </center>
21:  </td>
22:  </tr>
23:  <tr>
24:  <td colspan="2" align="center">
```

**Строки 1-24** — просто HTML, образующий раздел ссылок, который помещается внизу каждой страницы сайта.

```
25: <& Perl_ring &>
```

**В строке 25** вызывается компонент `perl_ring`. Этот компонент добавляет в самый низ каждой страницы ссылки на сайты Perl Ring.

```
26: </td>
27: </tr>
28: </table>
29: </center>
30: </body>
31: </html>
```

**Строки 26—31** завершают код HTML для страницы.

```
32: <%filter>
33: my $uri = $r->uri;
34: s{<a href="$uri/?">(.*?)</a>} {<b>$1</b>}i;
35: </%filter>
```

**Строки 32—35** образуют блок <%filter>, который изменяет результат, выводимый этим компонентом.

В **строке 33** можно заметить метод *$r->uri()*Объект `$r` очень похож на объект `$m`, о котором рассказывали выше, но предоставляет API к `mod_perl`, а не к Mason. В этой строке определяется URI вызываемой страницы.

**Строка 34** — регулярное выражение, которое выполняет поиск и замену. Попробуем расшифровать его. Во-первых, данные, выводимые компонентом, хранятся в переменной $_, с которой и работает регулярное выражение в блоке <%filter>. Запишем это выражение в более наглядном виде.

```
s
  {<a href="$uri/?">(.*?)</a>}
  <<Ь>$К/Ь>}
i;
```

Здесь s — оператор подстановки, следовательно, мы будем делать подстановку. Следующая часть отыскивает в `$_` ссылку, на которой мы находимся в настоящий момент, в зависимости от значения UR1. Вместе с ней захватывается весь текст между дескрипторами <a href> и </a>. Найденные данные сохраняются в $1. Затем дескрипторы ссылки заменяются дескрипторами полужирного текста, между которыми вставляется оставшийся текст. В результате у нас получается текст, содержащий только имя ссылки. Последняя строка — просто модификатор i, который указывает, что при поиске не должен учитываться регистр. В общем и целом, этот фильтр с помощью функции API `$r->uri` из `mod_perl` определяет, на какой странице мы находимся, а затем превращает ссылку на нее в обычный текст.

Все это сделано благодаря простому однострочному регулярному выражению в блоке `<%filter>`!

Надеемся, что вы поняли, какими большими возможностями обладают фильтры. Конечно, мы могли бы создать цепочку `if.. .elsif...else`, вручную определять активную страницу и отображать активную ссылку или простой текст, но это увеличило бы программу на несколько строк. Кроме того, с каждой новой добавленной ссылкой структуру `if. . .elsif.. .else` пришлось бы расширять. Если же применяется фильтр, надо только добавить новую ссылку в HTML, а фильтр сам позаботится об остальном.

# Заключение: код для примера сайта

Мы надеемся, что вы уже освоились в замечательном мире Mason. Это поистине превосходный инструмент для разработки сайтов. Mason продолжает развиваться, в него будут добавляться новые функции. Однако Джонатан Шварц, автор Mason, принял замечательное решение, отказавшись от пути "ползучего добавления функций". Это означает, что он не стал дополнять Mason все новыми и новыми возможностями, пытаясь заставить его делать *все*. Многие из нас знают, что такие попытки часто делают программу раздутой и медленной. Джонатану удалось сохранить Mason очень простым и в то же время создать очень мощный инструмент.

Эта глава завершается листингами всего кода, который образует главную страницу примера нашего сайта (см. листинги 13.3–13.18). Мы подробно рассмотрели не все компоненты, так как некоторые из них очень просты и не требуют пояснений. Листинги приведены не в том порядке, в котором они в действительности выполняются. Мы начнем с компонента верхнего уровня и затем покажем остальные компоненты в алфавитном порядке.

**Листинг 13.3. `index.html` – компонент верхнего уровня**

```
<& menu_start &>
  <& rss2html, site=>"perl-news" &>
  <& rss2html, site=>"perl.com" &>
<& menu_end &>
<& body_start &>
  <& my_news &>
  <& pictures &>
  <& links &>
<& body_end &>
```

**Листинг 13.4. Автообработчик – скрипт, который автоматически запускается для каждой обслуживаемой страницы**

```
<HTML><HEAD><TITLE><& SELF:title &></TITLE></HEAD>
<BODY BGCOLOR="<% $color %>">
<CENTER>
<img src="/images/new_header.gif" width="475" height="74">
</CENTER>
<HR WIDTH=85%>
<center>
<table width="100%" border="0" cellspacing="0"
  cellpadding="0">
<% $m->call_next %>
<& footer &>
<%method title>
  Perlguy's Pages
</%method>
<%init>
  my $color = "#f1edd3";
</%init>
```

**Листинг 13.5. body_end - компонент, завершающий таблицу, которая образует тело документа**

```
</center>
</td>
</tr>
```

Листинг 13.6. **body_start** - компонент, начинающий таблицу, которая **образует тело** документа

```
01: <td width="85%" valign="top">
02: <center>
```

Листинг 13.7. **dbhandler** — компонент, который вызывается, если **запрашиваемая** страница не существует

```
01: <font face="arial,helvetica">
02:<font size="7">
03: <center>
04: Oops!
05: </center>
06: </font>
07: <font size="5">
08: I didn't find the page you were looking for!<P>
09: Please check the URL and try again. .
10: </font>
11: </font>
```

Листинг 13.8. footer — компонент, который выводит ссылки, данные Perl Ring и завершает документ

```
01:  </tr>
02:  <tr>
03:  <td colspan="2" align="center">
04:  <br><br>
05:  <center>
06:  <hr width="85%">
07:   <brxbr>
08:   <font face=arial size=2>
09:  [
10:    <a href="/index.html">Home</a> |
11:    <a href="/articles.html">Articles</a> |
12:    <a href="/sql/index.html">DBI/SQL Tutorial</a> |
13:    <a href="/services.html">My Services</a> |
14:    <a href="/campcamel/index.html">Camp Camel</a> |
15:   <a href="/bbq/index.html">BBQ Anyone?</a> |
16:    <a href="/tattoo/Index.html">Camel Tattoo</a>
17: ]
18:  </font>
19:  <brxbr>
20:  </center>
21:  </td>
22:  </tr>
23:  <tr>
24:  <td colspan="2" align="center">
25:  <& Perl_ring &>
26:  </td>
27:  </tr>
28:  </table>
29:  </center>
30:  </body>
31:  </html>
```

```
32: <%filter>
33: my $uri = $r->uri;
34: s{<a href="$uri/?">(.*?)</a>} {<b>$1</b>}i;
35: </%filter>
```

**Листинг 13.9. links - компонент для создания раздела Links of Interest**

```
01: <& wrap_top, width=>'100%' &>
02: <table cellspacing="1" cellpadding="4" border="0"
      bgcolor="#ffffff"
03: width="100%">
04: <tr bgcolor="#cc9900">
05: <td valign="top" align="left">
06: <font face="Arial,Helvetica" size="5">
07: <b>Links of Interest</b>
08: </font>
09: </td>
10: </tr>
11: <tr>
12: <tdxfont face="arial, helvetica" size="3">
13: <p>
14: <ul>
15: <li><a href="articles.html">Articles I've Written</a>
16: <li><a href="/sql/">Perl and MySQL</a>
17: <lixa href="http://www.pm.org">The Perl Mongers</a>
18: <lixa href="http://stlouis.pm.org">Saint Louis
      Perl Mongers</a>
19: <lixa href="decss.html">DeCSS Code</a>
20: </ul>
21: </font></td>
22: </tr>
23: </table>
24: <& wrap_bottom S>
```

**Листинг 13.10. menu_end - компонент, завершающий раздел меню**

```
01: </td>
```

**Листинг 13.11. menu_start —компонент, который начинает раздел меню**

```
01: <tr>
02: <td width="15%" valign="top">
```

**Листинг 13.12. my_news - компонент для создания раздела My News**

```
01: <& wrap_top, width=>'100%' &>
02: <table cellspacing="1" cellpadding="4" border="0"
      bgcolor="#ffffff"
03: width="100%">
04: <tr bgcolor="#cc9900">
05: <td valign="top" align="left">
06: <font face="Arial,Helvetica" size="5">
07: <b>My News</b>
08: </font>
```

```
09:   </td>
10:   </tr>
11:   <tr>
12:   <td><font face="arial,helvetica" size="3">
13:   <P>
14:   % my $news = $m->file('news_text') ;
15:   <% $news %>
16:   </p>
17:   </td>
18:   </font>
19:   </tr>
20:   </table>
21:   <& wrap_bottom &>
```

**Листинг 13.13. `news_text` —текст для раздела My News**

```
01:  <b>I've Been Busy!</b><br>
02:  <p>
03:  In case some of you have e-mailed or called me, and still
have not heard back, I am sorry. I have been working on a book
and it is taking up a lot of my time. I will hopefully get back
in the swing of things once it is done.
04:  </p>
05:  <p>
06:  The book is being published by
<a href="http://www.awl.com"> Addison Wesley</a>, so
keep an eye out for it!
07:  </p>
```

**Листинг 13.14. `page_header` - компонент для создания "шапки" документа**

```
01:  <%args>
02:  $color
03:  $title => "Perlguy's Pages"
04:  </%args>
05:  <HTML><HEAD><TITLE><% $title %></TITLE></HEAD>
06:  <BODY BGCOLOR="<% $color %>">
07:  <CENTER>
08:  <img src="/images/new_header.gif" width="475"
       height="74">
09:  </CENTER>
10:  <HR WIDTH=85% SIZE=1 noshade>
11:  <table width="100%" border="0" cellspacing="0"
       cellpadding="0">
```

**Листинг 13.15. `pictures` - компонент, который создает на странице блок Pictures**

```
01:  <& wrap_top, width=>'100%' &>
02:  <table cellspacing="1" cellpadding="4" border="0"
       width="100%"
03:  bgcolor="#ffffff">
04:  <tr bgcolor="#cc9900">
05:  <td valign="top" align="left">
06:  <font face="Arial,Helvetica" size = "5">
07:  <b> Pictures </b>
```

```
08:  </font>
09:  </td>
10:  </tr>
11:  <tr>
12:  <tdxfont face="arial,helvetica" size="3">
13:  <p>
14:  <ul>
15:   <lixa href="/yapc">YAPC 1999</a>
16:   <lixa href="/yapc19100">YAPC 19100</a>
17:   <lixa href="/tpc3">The Perl Conference 3.0</a>
18:   <li><a href="/tpc4">The Perl Conference 4.0</a>
19:  <lixa href="/tattoo">Getfing My Camel Tattoo</a>
20:   <lixa href="/campcamel/1999.html">Camp Camel 1999</a>
21:  </ul>
22:  </fontx/td>
23:  </tr>
24:  </table>
25:  <& wrap_bottom &>
```

**Листинг 13.16. `rss2html` - компонент, который извлекает данные RSS из базы данных и придает им форму красивого блока новостей**

```
01: <%perl>
02: $sth->execute($site);
03: $content = $sth->fetch->[0];
04:  # Анализ содержимого RSS
05: $rss->parse($content);
06: </%perl>
07: <table bgcolor="#996600" border="0" width="200">
08: <tr>
09: <td>
10: <table cellspacing="1" cellpadding="4" bgcolor="#FFFFFF"
11: border=0 width="100%">
12: <tr>
13: <td valign="middle" align="center" bgcolor="#cc9900">
14: <font color="#000000" face="Arial,Helvetica">
15: <b>
16: <a href="<% $rss->{'channel'}->{'link'} %>">
      <% $rss->{'channel'}->{'title'} %></a>
17: </b>
18: </font>
19: </td>
20: </tr>
21: <tr>
22: <td align="center">
23: %  # Вывод изображения для канала
24: %  if ($rss->{'image'}->{'link'}) {
25:      <a href="<% $rss->{'image'}->{'link'} %>"><img
26:        src="<% $rss->{'image'}->{'url'} %>"
27:       alt="<% $rss->{'image'}->{'title'} %>" border="0"
28: %    if ($rss->{'image'}->{'width'}) {
29:        width="<% $rss->{'image'}->{'width'} %>"
30: %    }
31: %    if ($rss->{'image'}->{'height'}) {
32:        height="<% $rss->{'image'}->{'height'} %>"
33: %    }
34: ></a>
35: % }
36: </td>
```

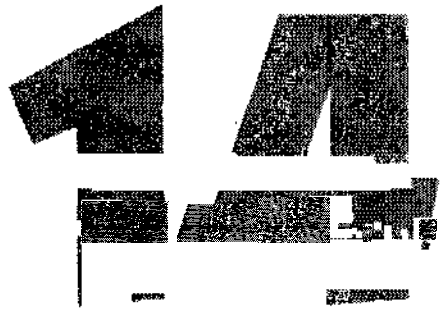
```
37:  </tr>
38:  <tr>
39:  <td><font face="arial,helvetica" size="2">
40:  <p>
41:  %  # Вывод элементов канала
42:  %    foreach my $item (@{$rss->{'items'}}) {
43:  %      next unless defined($item->{'title'}) &&
             defined($item->{'link'});
44:          <lixa href="<% $item->{ 'link'} %>">
               <% $item->{'title') %></a><BR>
45:  %    }
46:  %  # Есть ли здесь поле ввода текста?
47:  %  if ($rss->{'textinput'}->{'title'}) {
48:  <form method="get"
       action="<% $rss->{'textinput'}->{'link'} %>">
49:  <% $rss->{'textinput'}->{'description'} %><br />
50:  <input type="text"
       name="<% $rss->{'textinput'}->{'name'} %>"><br />
51:  <input type="submit"
       value="<% $rss->{'textinput'}->{'title'} %>">
52:  </form>
53:  %  }
54:  %  # Если есть элемент копирайта
55:  %  if ($rss->{'channel'}->{'copyright'}) {
56:       <pxsub><% $rss->{ 'channel'}->{ ' copyright' } %>
           </sub></p>
57:  %  }
58:  </font></td>
59:  </tr>
60:  </table>
61:  </td>
62:  </tr>
63:  </table>
64:  <%args>
65:  $5ite
66:  </%args>
67:  <%once>
68:  my $dbh;
69:  $dbh  = DBI->connect("DBI:mysql:book","book","addison")
70:    || die "Нельзя подключиться: $DBI::errstr\n"
       unless $dbh;
71:  my $sth  = $dbh->prepare(
       qq{SELECT data FROM rss WHERE site = ?});
72:  </%once>
73:  <%init>
74:  my $content;
75:  my $file;
76:  my $rss;
77:  $rss = new XML::RSS;
78:  </%init>
```

Листинг 13.17. **wrap_bottom** – компонент, который завершает обрамляющую таблицу

```
01:  </td>
02:  </tr>
03: </table>
```

**Листинг 13.18. wrap_top - компонент для создания обрамляющей таблицы**

```
01: <%args>
02: $bgcolor => "#996600"
03: $width => ''
04: </%args>
05: <table bgcolor="<% $bgcolor %>" border="0"
      width="<% $width %>">
06: <tr>
07: <td>
```



# Управление документами через Web

## Введение

Если вам нужно управлять группой документов, связанных с вашими личными делами или какой-то небольшой профессиональной задачей, с этим делом прекрасно справляется простой файловый менеджер. Но что, если вам приходится работать с 20 другими людьми, а документы должны располагаться где-то в одном месте? Файловый менеджер здесь становится бесполезным, так как он может работать только на вашем рабочем столе. Или, например, если вы открыли некоторые свои каталоги для совместного доступа, что мешает пользователю X начать работу с документом, когда с ним работает пользователь Y? Если это произойдет, тот, кто последним сохранит файл, уничтожит все изменения, сделанные предыдущим пользователем.

Кроме того, диски с совместным доступом не обладают такой "каменной" надежностью, как Web-сервер. Допустим, что каталоги совместного доступа находятся на диске пользователя Y, а он возьмет и уедет на месяц в отпуск в Антарктику! Или он может просто выключить свой компьютер, считая, что экономит этим электроэнергию, и забудет об этих общих каталогах, и по каким-то причинам не будет отвечать на телефонные звонки.

Для решения этой и массы других проблем, таких как управление версиями и резервирование документов, были созданы системы управления документами (document management systems — DMS). Системы DMS корпоративного уровня могут стоить сотни тысяч долларов и требовать дорогой аппаратуры и специальных системных администраторов, которые следили бы за их работой.

То, что какая-то вещь стоит кучу денег, вовсе не означает, что она очень хороша. Когда Брент работал на фирме "Боинг", у них была дорогая система DMS и отдель-

ная машина под UNIX, на которой эта система работала. Когда Брент стал работать на этой фирме, система работала уже шесть месяцев, и с ней были вечные проблемы. Брент пришел, увидел и написал законченную, специализированную систему DMS, которая работала на рядовой машине с Windows NT, IIS и Perl. Это стало возможным именно благодаря Perl; NT или IIS вполне можно было бы заменить другой операционной системой и Web-сервером.

Система DMS была предназначена для совместного проекта, что означало, что в этой системе находились данные других компаний — наших непосредственных конкурентов. Естественно, мы не хотели, чтобы у них был доступ к документам, предназначенным только для "Боинга": Версия системы, разработанная для "Боинга", включала следующие функции.

- Проверка выдачи и возврата файлов
- Безопасность, основанная на правах пользователей и групп
- Автоматические напоминания по электронной почте о тех документах, которые задерживаются пользователями слишком долго
- Управление версиями
- Динамически изменяющийся в зависимости от прав пользователя интерфейс

В этой главе мы с вами разработаем упрощенную версию системы DMS. Наша система будет занимать менее 500 строк кода! Это довольно большое приложение, и мы сможем показать какие-то результаты только после того, как оно будет закончено. Конечный результат определенно стоит той работы, которую потребуется для него приложить, так что давайте приступим к делу.

# План

Первое, что нужно сделать при создании проекта такого рода, — это составить план. Если ваша разработка предназначается для применения в широком кругу пользователей, постарайтесь привлечь дополнительных участников хотя бы к проектированию интерфейса и составлению списка функций. Для этой системы DMS мы выбрали следующие функциональные возможности.

- Загрузка файлов
- Просмотр файлов
- Идентификация пользователей
- Динамически изменяемая главная страница (в зависимости от прав пользователя)
- Проверка выдачи и возврата файлов
- Расширенные описания
- Безопасность на базе групп

Имея этот список функций, мыбудем знать, что нам делать дальше. Нам предстоит создать программу `auth.cgi` для выполнения идентификации, `main.cgi` — для отображения главной страницы, `upload.cgi` — для загрузки файлов и `viewer.cgi` — для просмотра файлов. Также будет создан файл `shared.pl`, в который мы будем записывать все типовые функции, которые придумаем.

Нам также понадобится пара таблиц базы данных, в которых будет храниться информация. Похоже, что мы довольно интенсивно используем базы данных, и это действительно так. Web — это превосходная вещь, но от нее немного пользы, если у вас нет информации, которую можно сохранять и отображать. Базы данных созданы как раз для хранения и представления данных, и вместе с Web они прекрасно дополняют друг друга.

В одной таблице базы данных мы будем хранить информацию о пользователях системы. Мы должны знать, кто вошел в систему и к какой группе он принадлежит. Нам понадобится и другая таблица, чтобы хранить информацию о самих файлах. Двух таблиц достаточно для того, что мы будем делать в этом приложении. Если вы планируете расширить функциональные возможности системы, то, в зависимости от целей, возможно, придется добавить новые таблицы. Ниже приведена структура двух наших таблиц в формате MySQL.

```
# Структура таблицы dms_users
CREATE TABLE dms_users (
  username    varchar(40) DEFAULT '' NOT NULL,
  password    varchar(25),
  e_mail      varchar(150),
  phone       varchar(25),
  group_id    varchar(25),
  PRIMARY KEY (username)
);
# Структура таблицы dms_files
CREATE TABLE dms_files (
  file_id int(11) DEFAULT '0' NOT NULL auto_increment,
  filename    varchar(255),
  description varchar(255),
  location    varchar(255),
  mime_type   varchar(50),
  group_id    varchar(25),
  who_to      varchar(40),
  out_date    varchar(20),
  PRIMARY KEY (file_id)
);
```

Первая таблица служит просто для сохранения некоторой информации о пользователях. В этом примере DMS нет интерфейса администрирования, позволяющего непосредственно редактировать эту таблицу. Для этой цели прекрасно подошла бы простая форма с полями ввода, но, так как мы хотим охватить лишь основные функциональные возможности системы DMS, средства администрирования не включены в ее состав. Функции администрирования были бы хорошим подспорьем к системе DMS, и, когда вы закончите эту книгу, вы сможете без труда создать их. Чтобы программа могла работать, таблица dms_users должна содержать какие-то данные, поэтому надо внести в нее информацию о нескольких фиктивных пользователях.

Для поля group_id в этом примере мы создали четыре группы: PEON, USER, PHB И BOFH. Группа PEON самого низкого уровня, а группа BOFH имеет доступ ко всем документам. Также нам потребуется создать несколько каталогов. Программы будут располагаться в каталоге cgi-bin/dms, а подкаталоги данных — в каталоге cgi-bin/dms/data. Имена подкаталогов данных совпадают с названиями групп: первый подкаталог будет носить имя cgi-bin/dms/data/PEON и т.д. Готовая структура подкаталогов должна выглядеть так:

```
cgi-bin
        /dms
            /data
                 /PEON
                 /USER
                 /PHB
                 /BOFH
```

Все страницы генерируются динамически, поэтому в обычном каталоге HTML Web-сервера нам не понадобится ни одна форма. Каталог data должен иметь права,

позволяющие "Web-пользователю" создавать файлы. "Web-пользователь" — это пользователь, под учетной записью которого работает Web-сервер. По соображениям безопасности следует также убедиться, что Web-сервер не будет обрабатывать страницы непосредственно из каталога cgi-bin. Для этого требуется специальное действие, которое вы, скорее всего, не выполнили, так что все в порядке. Все же для проверки поместите файл HTML в подкаталог cgi-bin/dms/data/PEON и попробуйте вызвать его из броузера. Даже если ввести правильный путь, Web-сервер должен возвратить ошибку "доступ запрещен" или нечто подобное.

# auth.cgi

Первая программа, которую мы создадим, — auth.cgi. Эта программа будет использоваться для входа пользователя в систему и выхода из нее, а также будет генерировать страницу входа.

```
01:  #!/usr/bin/perl -wT
02:  # auth.cgi
03:  use DBI;
04:  use strict;
05:  use CGI qw(:standard);
06:  require "./shared.pl" or die "Нельзя найти файл. $!\n";
```

**Строки 1 и 2** сообщают системе, где найти Perl, включают предупреждения и проверку на загрязнение и сообщают нам название программы. Указание названия программы позволяет легко выделить ее из множества листингов.

**Строка 3** загружает модуль **DBI,** что дает нам простой доступ к базе данных.

**Строка 4** включает строгий синтаксис, что заставляет нас писать более "чистый" код.

**Строка 5** загружает модуль CGI и импортирует стандартные функции.

**Строка *6*** загружает файл shared.pl. В этом файле мы будем хранить подпрограммы общего назначения. Многим людям очень не нравится функция *require ()*. Они предпочли бы, чтобы все файлы загружались как модули, функцией *use ()*. Я же считаю, что *require ()* быстрее и проще, и, если Ларри оставит эту функцию в языке, я буду ее применять. Нет ничего плохого в том, чтобы использовать функцию *require ()*, когда это нужно.

```
07:  my $user = param('user-name');
08:  my $pass = param('password');
09:  my $action = param('action');
10:  my $dbh = DB_Connect();
```

**В строках 7—9** объявляется несколько переменных и им присваиваются значения, переданные из формы **HTML.**

**В строке 10** вызывается подпрограмма *DB_Connect()*, которая находится в shared.pl. Эта подпрограмма возвращает дескриптор базы данных, который мы сохраняем в **$dbh.**

```
11:  Logout() if ($action eq "logout");
12:  Login_Page() unless($pass);
```

**В строке 11,** если из формы передано действие logout, вызывается функция *Logout ()*.

**В строке 12,** если из формы не передан пароль, вызывается функция *Login_page()*. Так как все пользователи должны иметь пароли, при входе в систему должен вводиться пароль. Если пароль не представлен, мы снова направляем пользователя на страницу входа. Вспомните, что *unless ()* означает "если-не".

```
13:  my $valid = Check_Login($user, $pass);
```

В **строке 13** вызывается подпрограмма *Check_Login()* и в нее передаются имя пользователя и пароль. Эта подпрограмма возвращает 0 при неправильных имени пользователя и/или пароле и 1, если данные подтверждаются.

```
14:  if ($valid) {
15:    my $cookie = Create_Cookie("1h", $user);
16:    print redirect(-uri=>"main.cgi", -cookie=>$cookie);
17:    exit;
18: }
```

**Строка 14** начинает блок if. . .else, в котором определяется, как нам поступить с данным пользователем. Сначала мы проверяем значение $valid.

**В строке 15** с помощью подпрограммы *Create_Cookie()* создается cookie. В эту подпрограмму передается значение lh (один час) и имя пользователя. Полученное в результате cookie будет гарантировать, что пользователь имеет право работать с нашей системой DMS. Для значения cookie принимается просто имя пользователя.

**Строка** 16 направляет пользователя в программу `main.cgi` и передает в нее только что созданное cookie, которое устанавливается в Web-броузере пользователя. Если cookie в броузере пользователя не разрешены, он ничего не сможет сделать в этом приложении.

**Строка 17** — выход из программы.

**Строка 18** закрывает первую часть блока if. . .else.

```
19:  else {
20:    my $time = time();
21:    print redirect(-uri=>"auth.cgi?$time");
22:    exit;
23: }
```

**Строка 19** начинает вторую часть блока if.. .else. Этот код выполняется, если данные для входа в систему по каким-то причинам недействительны.

В **строке 20** в новую переменную $time записывается текущее время.

**Строка 21** направляет пользователя вновь в эту же программу, `auth.cgi`. Время, приписанное к концу URL, необходимо только потому, что некоторые броузеры кэшируют информацию со страниц CGI. Это время будет постоянно изменяться, и броузер каждый раз будет видеть другой URL. Этим способом решается еще одна проблема, которую мы обнаружили: модуль `CGI.pm` не позволяет делать перенаправление в ту же программу, но с добавлением времени к URL это разрешается.

**Строка 22** — выход из программы.

**Строка 23** завершает блок if. . .else.

```
24:  sub Logout {
25:    my $time = time();
26:    my $cookie = Create_Cookie("-1h", "");
27:    print redirect(-uri=>"auth.cgi?$time", -cookie=>$cookie);
28:    exit;
29:  }
```

**Строка** 24 начинает подпрограмму *Logout ()*. Эта подпрограмма служит для выхода пользователя из системы и просто сбрасывает значение cookie.

В **строке 25** в новую переменную $time записывается текущее время.

В **строке 26** с помощью подпрограммы *Create_Cookie()* создается cookie с пустым значением.

**Строка** 27 снова направляет пользователя в программу `auth. cgi` и сбрасывает cookie.

**Строка 28** — выход из программы.

Строка 29 завершает подпрограмму *Logout ()*.

```
30:  sub Create_Cookie {
31:    my Sexp = shift;
32:    my $val = shift;
```

**Строка 30** начинает подпрограмму *Create_Cookie()*.

**В строках 31 и 32** значения, переданные в подпрограмму, сохраняются в переменных $exp и $val.

```
33:    my $cookie = cookie(-name => 'dms',
34:                        -value => $val,
35:                       -expires => $exp );
```

В **строках 33—35** с помощью функции *cookie ()* из CGI.pm формируется cookie. Имя cookie жестко зафиксировано в программе, так как оно не должно изменяться, но значение ($val) и срок действия (Sexp) будут непостоянны, и поэтому для них используются переменные.

```
36:     return($cookie);
37:  }
```

**В строке 36** cookie возвращается в вызывающую функцию.

**Строка 37** завершает подпрограмму *Create_Cookie ()*.

```
38:  sub Check_Login {
39:    my Suser = shift;
40:    my $pass = shift;
41:    my $data;
```

**Строка 38** начинает подпрограмму *Check_Login()*. Эта подпрограмма проверяет, соответствуют ли введенные имя пользователя и пароль тому, что хранится в базе данных.

В **строках 39—41** создается несколько переменных, которые понадобятся нам впоследствии. Переменные $user и Spass — это соответственно имя пользователя и пароль, которые передаются в функцию при ее вызове.

```
42:     my $sth = $dbh->prepare( qq{  SELECT password
43:                                   FROM dms_users
44:                                   WHERE
45:                                   usemame = ?
46:                                  } );
```

**В строках 42—46** создается дескриптор $sth, в который записывается уже готовая к применению команда SQL. Эта команда предназначена для получения пароля из таблицы dms_users.

```
47:     $sth->execute($user);
48:     Sdata = $sth->fetch;
```

**В строке 47** выполняется только что подготовленная команда. В нее передается переменная $user, которая подставляется на место метки-заполнителя ?.

**В строке 48** результат выполнения команды извлекается методом *fetch ()* и сохраняется в переменной $data. Метод *fetch ()* возвращает ссылку на массив. Каждый элемент этого массива — поле для возвращенной строки данных.

```
49:     $sth->finish;
50:     $dbh->disconnect;
```

**В строке 49** вызывается метод *finish()*. Этот метод освобождает дескриптор команды, благодаря чему база данных может правильно завершить работу.

**В строке 50** вызывается метод *disconnect ()*, который отключает нас от базы данных.

```
51:     ($data->[0] eq $pass) ? return 1 : return 0;
52: }
```

**В строке 51** мы с помощью трехместного оператора проверяем, соответствует ли нулевой элемент массива $data содержимому $pass, т.е. паролю, переданному с Web-страницы. Так как мы должны были получить только одно поле, а каждый пользователь может иметь только один пароль, то этот пароль, если он есть, должен находиться в data->[0]. Если этот пароль не совпадает с введенным или не получен по какой-то другой причине, подпрограмма возвращает 0. Если пароль подтверждается, возвращается значение 1.

Трехместный (или троичный) оператор работает следующим образом. Если выражение слева от знака ? истинно, выполняется первый элемент справа от ?. Если оно ложно, выполняется последний элемент. Для наглядности эту логику можно представить так:

```
if ($data->[0] eq $pass) {
  return 1;
}
else {
  return 0;
}
```

Трехместный оператор просто более компактен.

**Строка 52** завершает подпрограмму *Check_Login()*.

```
53: sub Login_Page {
54:   print header();
```

**Строка 53** начинает подпрограмму *Login_Page ()*. Эта подпрограмма служит только для создания формы HTML для входа пользователя на сайт.

**Строка** 54 выводит заголовок HTTP с помощью функции header *()* из CGI.pm.

```
55: print «HTML;
56:   <html><head><title>Пример DMS</title></head>
57:   <body bgcolor="#ffffff">
58:   <center>
59:   <form method="post">
60:  <p>
61:   <h2>Вас приветствует пример DMS</h2>
62:  </p>
63:  <p>
64:    <table border="1" cellspacing="0">
65:    <tr>
66:    <td>Имя пользователя:</td>
67:    <tdxinput type="text" name="username"></td>
68:    </tr>
69:    <tr>
70:    <td>Пароль:</td>
71:    <tdxinput type="password" name="password"></td>
72:    </tr>
73:    <tr align="center">
74:    <td colspan="2">
75:    <input type="submit" value="    Войти    ">
76:    </td>
77:    </tr>
78:    </table>
79:  </p>
80:   </form>
81:   </center>
82:   </body>
83:   </html>
84: HTML
```

Строки 55—84 — включенный документ, который образует форму HTML для страницы входа.

```
85:     exit;
86:  }
```

Строка 85 — выход из программы.

Строка 86 завершает подпрограмму *Create_Login()*. Итак, первая часть нашего приложения DMS уже готова. Если в базу данных уже введена какая-то тестовая информация о пользователях, этот скрипт уже можно запустить. Мы получим страницу, показанную на рис. 14.1.

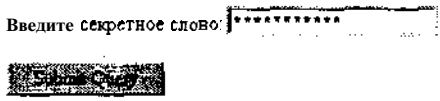



*Рис. 4.1. Экран входа в систему DMS*

После того, как мы введем еще одну программу, shared.pl, скрипт auth.cgi можно будет проверить в работе. Файл shared.pl очень невелик, и его набор не займет много времени. Если ввести неправильное имя пользователя и/или пароль, программа должна вернуть вас на страницу входа. Если же имя пользователя и пароль правильны, должна быть вызвана следующая программа main.cgi. Этой программы у нас еще нет, но, по крайней мере, мы можем убедиться, что программа auth.cgi работает.

# shared.pl

Вторая программа, которую мы рассмотрим, — это файл shared.pl. Он содержит три подпрограммы, необходимые в нескольких программах DMS.

```
01:  sub DB_Connect {
02:    my $dbh = DBI->connect("DBI:mysql:book","book","addison")
03:      or die "Ошибка при подключении к БД! $DBI::errstr\n";
04:    return($dbh);
05: }
```

Строка 1 начинает подпрограмму *DB_Connect()*. Так как shared.pl— только файл включения, а не самостоятельная исполняемая программа, для него не нужна первая строка #!/usr/bin/perl.

В строках 2 и 3 создается подключение к базе данных и полученный дескриптор базы данных сохраняется в переменной $dbh.

В строке 4 из подпрограммы возвращается дескриптор базы данных.

Строка 5 завершает подпрограмму *DB_Connect ()*.

```
06:  sub Check_Cookie {
07:    my $user = cookie ('dms');
```

Строка 6 начинает подпрограмму *Check_Cookie()*.

Строка 7 создает переменную под названием $user и присваивает ей значение, возвращенное подпрограммой *cookie ()*. В эту подпрограмму передается имя cookie, которое надо раскрыть, и она возвращает значение этого cookie, если его можно найти.

```
08:     if ($user) {
09:       return($user);
10:     }
```

**Строка 8** начинает блок if. . .else. Здесь мы проверяем, есть ли в переменной $user какое-либо значение. Если это так, следовательно, пользователь уже установил cookie, и мы выполняем блок кода.

**В строке 9,** так как cookie установлено, просто возвращается переменная $user, которая содержит имя текущего пользователя.

**Строка 10** завершает первую часть блока if...else.

```
11:     else {
12:       print redirect(-uri=>"auth.cgi");
13:       exit;
14:     }
15:   }
```

**Строка И** начинает вторую часть блока if. . .else.

**Строка 12** направляет пользователя в программу auth.cgi. Это происходит в случае, если cookie не был установлен, срок действия cookie истек или произошла какая-то другая ошибка. Тогда пользователю предоставляется новая попытка войти в систему.

**Строка 13** — выход из программы.

**Строка 14** закрывает блок if...else.

**Строка 15** завершает подпрограмму *Check_Cookie()*.

```
16:  sub Get_Group {
17:    my $user = shift;
```

**Строка** 16 начинает подпрограмму *Get_Group()*.Эта подпрограмма возвращает группу, к которой принадлежит пользователь.

В **строке** 17 значение, переданное в подпрограмму, сохраняется в переменной $user.

```
18:     my $dbh = DBI->connect("DBI:mysql:book","book","addison")
19:       or die "Ошибка при подключении к БД! $DBI::errstr\n";
```

В **строках** 18 и 19 создается подключение к базе данных для последующего получения группы. Это подключение выполняется здесь потому, что в главной программе оно оказалась бы за пределами данной подпрограммы.

```
20:     my $sth = $dbh->prepare( qq{ SELECT group_id
21:                                  FROM dms_users
22:                                  WHERE
23:                                  username = ?
24:                                } );
```

**В строках 20—24** создается и подготавливается команда SQL для получения group_id из таблицы базы данных.

```
25:     $sth->execute($user);
26:     $group = $sth->fetch->[0];
27:     redirect(-uri=>"auth.cgi") unless $group;
```

**Строка** 25 выполняет команду SQL, которую мы только что подготовили.

В строке 26 в переменную $group записывается значение 0-го элемента массива, возвращенного при вызове *fetch ()*.

**Строка** 27 возвращает пользователя на **страницу** auth. cgi, если группу определить не удалось.

```
28:     $sth->finish;
29:     $dbh->disconnect;
30:     return($group);
31:   }
```

**В строке 28** методом *finish ()* освобождается дескриптор $sth.

**В строке 29** вызывается метод *disconnect (),* который производит отключение от базы данных.

**Строка 30** возвращает значение $group.

**Строка 31** завершает подпрограмму *Get_Group{).*

```
32:    1;
```

**Строка 32** — последняя! В ней стоит только цифра 1;. Все файлы включения должны возвращать значение истины; наиболее обычный способ для этого — код "1;". Но можно поставить здесь и какой-нибудь текст, поскольку он также будет иметь значение истины. Итак, наш небольшой скрипт закончен, и мы можем запустить программу auth.cgi, чтобы проверить ее работу. Когда вы закончите эту проверку, мы перейдем к следующим скриптам нашей системы DMS.

# main.cgi

Следующая программа, main.cgi, будет выполнять основную работу в нашем приложении. Она будет играть роль "начальной страницы" для приложения, а также отображать ссылки на все документы, к которым пользователь имеет право обращаться.

Мы не собираемся делать программу main.cgi универсальной, чрезмерно продвинутой. Скорее она будет хорошей основой для более крупного и сложного приложения DMS. Вы сами можете придать странице нужный вам вид и встроить любые другие функции, какие пожелаете.

```
01:  #!/usr/bin/perl -wT
02:  # main.cgi
```

**Строки 1–2** должны быть вам хорошо знакомы. Здесь мы просто сообщаем системе, где найти Perl, включаем предупреждения и проверку на загрязнение и даем комментарий, чтобы можно было легко узнать, что это за программа.

```
03:  use strict;
04:  use CGI qw(:standard);
05:  use CGI::Carp qw(fatalsToBrowser);
06:  use DBI;
07:  require "./shared.pl" or die "Нельзя найти файл. $!\n";
```

**В строке 3** мы задаем использование строгого синтаксиса.

**Строка 4** загружает модуль CGI и импортирует стандартные функции.

**Строка** 5 загружает модуль Carp. Этот модуль позволяет отображать в броузере более конкретные сообщения об ошибках, что облегчает поиск неисправностей.

**В строке 6** загружается модуль DBI. Этот модуль требуется для доступа к базе данных.

**Строка 7** включает в программу код из файла shared.pl, который мы только что написали. Оператор require *()* очень напоминает по своему действию ключевое слово include B C.

```
08:  my $user = Check_Cookie();
09:  my $dbh = DB_Connect () ;
10:  my $group = Get_Group($user);
```

**В строке 8** вызывается подпрограмма *Check_Cookie()* и ее результат, имя пользователя, записывается в переменную Suser.

**В строке 9** вызывается подпрограмма *DB_Connect()*, которая создает подключение к базе данных. Дескриптор подключения сохраняется в переменной $dbh.

**В строке 10** вызывается подпрограмма *Get_Group()*,которая обращается к базе данных и определяет группу, к которой принадлежит данный пользователь.

```
11:  my $action = param{'action');
12:  my $file = param('file');
13:  my $out_to = param('out_to');
```

В **строках 11–13** с использованием функции CGI.pm *param()*из формы HTML извлекаются данные полей action, file и out_to. Эти значения сохраняются в одноименных переменных.

```
14:  my %files;
```

**В строке 14** просто объявляется хэш % files, который мы будем использовать позже в программе.

```
15:  my @groups - qw(PEON USER PHB BOFH.) ;
```

В **строке 15** создается массив под названием @groups, содержащий имена всех групп для этой системы DMS. Приоритет групп соответствует их порядку в массиве. Первым следует самый низкий уровень доступа (в нашем случае PEON), а последним — самый высокий уровень (BOFH).

```
16:  foreach (@groups) {
17:    $files{$_} = Get_Files($_);
18:    last if ($group eq "$_"};
19:  }
```

**Строка 16** начинает цикл *foreach ()*, который обходит все значения массива @groups. Назначение этого цикла — записать в хэш ссылки на массивы файлов для каждой из групп пользователей. Это сложно понять сразу, поэтому я попробую объяснить поподробнее. Мы заполняем хэш %files (он был создан в строке 14). В каждой итерации цикла переменная $_ получает значение текущего элемента массива. Сначала она получит значение PEON, И Т.Д. ДЛЯ значения этого элемента хэша мы вызываем подпрограмму *Get_Files (},* которая отыскивает данные (как она это делает, мы расскажем чуть позже) и возвращает ссылку на них.

**В строке 17** в текущий элемент хэша заносится ссылка, возвращенная подпрограммой *Get_Files()*. Как мы знаем, ссылка — это всего лишь скалярная переменная. Однако вместо "обычного" значения, такого как строка или число, она содержит адрес другой переменной. Эта другая переменная, на которую указывает ссылка, может содержать данные любого типа.

**В строке 18** мы прерываем выполнение цикла *foreach(),* если текущее значение массива равно $group. Иначе говоря, если пользователь принадлежит к группе USER, он получит сначала данные для группы PEON, а затем данные для группы USER. НО так как на этой итерации значения $group и $_ будут совпадать, цикл прекратится. Следовательно, пользователь не получит доступа к данным, предназначенным для следующих групп с более высоким приоритетом.

**Строка 19** завершает цикл *foreach ()*.

```
20:  Display_Page();
```

**В строке 20** вызывается подпрограмма *Display_Page()*. Эта подпрограмма отображает "начальную страницу".

```
21:  sub Get_Files {
22:    my Sgroup = shift;
23:    my @temp;
```

**Строка** 21 начинает подпрограмму *Get_Files ()*. Эта подпрограмма выдает все файлы, для которых определено имя группы, указанное при вызове.

В **строке** 22 с помощью функции *shif()* мы получаем первый — и, в данном случае, единственный — параметр, переданный в подпрограмму.

В **строке 23** объявляется переменная `@temp`.

```
24:     my $sth = $dbh->prepare( qq{ SELECT *
25:                                    FROM dms_files
26:                                   WHERE group_id = ? } );
```

**В строках 24—26** формируется команда SQL для получения из таблицы `dms_files` всех записей, поле `group_id` которых соответствует значению, переданному в подпрограмму.

```
27:   $sth->execute($group) ;
```

**Строка 27** выполняет команду SQL, которую мы только что подготовили.

```
28:     while (my $ptr = $sth->fetchrow_hashref) {
29:       push @temp, $ptr;
30:     }
```

**Строка 2?** начинает цикл *while ()*, который выполняется, пока по запросу **SQL** еще можно получить данные. Метод *fetchrow_hashref* возвращает одну строку данных в хэше. На каждой итерации цикла возвращается следующая строка данных. Указатель на хэш, содержащий данные, сохраняется в переменной `$ptr`.

**В строке 29** текущее значение `$ptr` дописывается в конец массива `@temp`. Как известно, `$ptr` это ссылка на массив, следовательно, каждый элемент создаваемого нами массива — ссылка на хэш, который содержит строку данных!

**Строка 30** завершает цикл *while()*.

```
31:     return \@temp;
32:   }
```

**Строка 31** возвращает из подпрограммы ссылку на массив `@temp`. Символ \ перед именем переменной создает ссылку на эту переменную. Этот способ позволяет нам возвращать не весь массив, а одно значение — ссылку на него.

**Строка 32** завершает подпрограмму *Get_File()*.

```
33:   sub Check_Status {
34:     my ($who_to, $date_out, $file_id) = @_;
35:     my $data;
```

**В строке 33** начинается подпрограмма *Check_Status()*. Эта подпрограмма проверяет, выдан ли пользователю указанный файл. Затем она создает страницу HTML, на которой показывает, кому выдан этот файл, или создает ссылку для возвращения файла, если вы сами — тот пользователь, которому он был выдан.

**Строка 34** принимает значения, переданные в подпрограмму, и помещает их в три новые переменные.

В строке 35 создается еще одна переменная, которую мы будем использовать позже.

```
36:     return unless($who_to);
```

**В строке 36** подпрограмма возвращает управление, если переменная `$who_to` не содержит значения. Это имеет место, если файл никому не выдан. В этом случае мы не должны ничего делать, и подпрограмма просто завершается.

```
37:     my $sth = $dbh->prepare( qq{ SELECT phone, e_mail
38:                                    FROM dms_users
39:                                   WHERE username = ? } );
```

**В строках 37—39** создается команда SQL для получения номера телефона и адреса электронной почты пользователя, которому выдан файл.

```
40:     $sth->execute($who_to);
41:     my $ptr = $sth->fetch;
```

**Строка 40** выполняет команду SQL, которую мы только что создали, и передает в нее значение `$who_to`.

**В строке 41** извлекается строка данных и ссылка на нее сохраняется в `$ptr`. Так как имя пользователя — первичный ключ таблицы, при поиске по этому полю может быть найдена только одна строка. Благодаря этому нам не нужно помещать операцию выборки данных в цикл.

```
42:     if ($who_to eq $user) {
43:       $data = "<b>Выдано:</b>";
44:       $data .= "<a href='viewer.cgi?action=checkin";
45:       $data .= "&file=$file_id'>Вернуть</a><br>";
46:     }
```

**Строка 42** начинает блок `if...else`, в котором проверяется, кому выдан документ — текущему пользователю или другому. Мы сравниваем имя пользователя, которому выдан документ (`$who_to`) с именем пользователя, который в настоящий момент работает с системой DMS (`$user`). Если `$who_to` и `$user` совпадают, то файл выдан именно данному пользователю DMS. Если дело обстоит так, мы выполняем блок кода и создаем ссылку HTML, щелчок на которой возвращает файл в систему DMS.

**Строки 43—45** — код HTML, образующий ссылку для возврата документа. Код в виде строки сохраняется в `$data`.

**Строка 46** завершает первую часть блока `if...else`.

```
47:     else {
'48:      $data = "<b>Кому выдано: </b>";
49:      '$data .= "<a href='mailto:$ptr->[1]'>$who_to</a>";
50:       Sdata .= "- Тел.: $ptr->[0]<br>";
51:     }
```

**Строка 47** начинает вторую часть блока `if...else`. Этот код выполняется, если `$user` и `$who_to` различны. Это означает, что текущий пользователь и пользователь, который получил файл — разные люди.

**Строки 48—50** — код HTML, образующий ссылку `mailto`.

**Строка** 51 закрывает блок `if...else`.

```
52:     return($data);
53:   }
```

В **строке** 52 из подпрограммы возвращается строка `Sdata`.

**Строка** 53 завершает подпрограмму *Check_Status*.

```
54:   sub Display_Page {
55:     print header;
```

**Строка** 54 начинает подпрограмму *Display_Page ()*. Эта подпрограмма служит для отображения главной страницы приложения DMS. Эта страница будет содержать заголовок и список всех файлов, к которым пользователь имеет доступ.

**Строка 55** функцией *header ()* выводит стандартный заголовок **HTTP.**

```
56:     print «HTML;
57:       <html><head>
58:       <title>Система управления документами</title>
59:       </head>
60:       <body bgcolor="#ffffff">
61:       <center>
```

```
62:      <table border="1" width="60%">
63:       <trxtd align="center">
64:      <h2>Простая система управления документами</h2>
65:      <b>Добро пожаловать, $user!</b><br>
66:       </tdx/tr>
67:       <trxtd valign='top'>
68:  HTML
```

**Строки 56—68** — включенный документ, в котором просто выводится начало генерируемой страницы HTML.

```
69:    foreach my $grp (keys %files) {
70:      next unless(@{$files{$grp}});
```

**Строка 69** начинает большой цикл *foreach(),* который в значительной степени образует основу этой программы. Этот цикл обходит все ключи, находящиеся в хэше `%files`. Каждый ключ соответствует группе PEON, USER и т.д. Мы не организуем цикла по группам в хэше `@groups`, так как в хэше %files будут только те группы, к которым принадлежит пользователь.

**8 строке 70** мы переходим к следующей итерации цикла, если в массиве не обнаруживается данных. Выражение @{$files{$grp}} — еще один пример трудно расшифровываемого кода, поэтому сейчас мы попробуем его разобрать. Значение $files{$grp} может быть, например, `$files{PEON}`. Дальше понять уже не сложно; мы видим обычный способ получения значения из хэша.

Вспомните, что хэш `%files` содержит ссылки на массивы. Следовательно, $files{$grp} — ссылка на один из этих массивов. Чтобы обратиться к массиву, мы должны разыменовать ссылку, например, так: `@ {ссылка_на_массив}`.Итак, мы получили `@{$files{$grp}}`. Иногда код Perl действительно трудно понять, но, если разбить его на отдельные компоненты, он уже не будет казаться таким сложным.

```
71:       print qq( <p><b>Область $grp</bxbr>\n );
```

**В строке 71** просто выводится имя группы, чтобы пользователь мог узнать, к какой области относятся следующие файлы.

```
72:      foreach my $ptr (@{$files{$grp}}) {
73:        my $fname;
74:        my $checked_out = Check_Status($ptr->{who_to},
75:                                       $ptr->{out_date},
                                           $ptr->{file_id});
```

В строке 72 снова применяется формат, который мы подробно разобрали в строке 70. Каждый элемент массива `@{$files{$grp}}` — ссылка на хэш, который содержит строку данных. Значение этой ссылки мы присваиваем переменной $ptr.

В строке 73 просто объявляется еще одна переменная.

**В строках 74—75** проверяется состояние текущего файла. В переменную $checked_out записываются результаты подпрограммы *Check_Status*. В эту подпрограмму передаются три параметра: имя пользователя, которому выдан файл, дата выдачи и идентификатор (`file_id`) выданного файла. Первые два параметра могут быть пустыми, что означает, что файл не выдан.

Так как Sptr — ссылка на хэш, а не сам хэш, чтобы обратиться к его данным, мы должны *разыменовать* ссылку с помощью оператора стрелки ->. Так, в выражении `$ptr->{file_id}` Sptr — ссылка на хэш, -> — разыменование ссылки, а `file_id` — ключ хэша. Если бы у нас был собственно хэш, а не ссылка на него, мы могли бы записать `$hash{file_id}`. Сами по себе ссылки — вовсе не трудное понятие, но, когда речь заходит о хэше массивов ссылок на хэши, не обойтись без дополнительных разъяснений.

Дать ясное понятие о ссылках может помочь замечательная статья Марка Джейсона Доминуса (Dominus) *Understand References Today*. Ее можно назвать "библией ссылок" Эта статья находится на сайте автора по адресу http://www.plover.com/~m.jd/-perl/FAQs/references.html; ей определенно стоит уделить внимание.

```
76:         print qq( <table bgcolor="#e0e0e0" width="100%">
              <tr><td> ) ;
77:         print qq( <b>Имя файла:</Ь> );
```

В **строках 76–77** создается начало кода HTML для списка файлов.

```
78:           $fname = qq(<a href="viewer.cgi?action=);
79:           $fname .= qq(view&file=$ptr->{file_id}">);
80:           Sfname .= qq($ptr->{filename}</a>);
```

**В строках 78–80** создается ссылка, щелкнув на которой пользователь может просмотреть файл. В этом приложении DMS мы не даем ссылку непосредственно на документ. Вместо этого мы вызываем программу `viewer.cgi`, которая выбирает файл и отправляет его пользователю, *предварительно* убедившись, что он имеет необходимые права для просмотра этого файла. В строках 79 и 80 мы снова используем оператор стрелки для разыменования ссылки и обращения к значениям хэша.

```
81:       unless($checked_out)  {
82:          Sfname  .= qq(  [<a href="viewer.cgi?action=};
83:       "  Sfname  .= qq(checkout&file=$ptr->{file_id}">);
84:        $fname  . = qq (Выдать</a>] ) ;
85:        }
```

В строке **81** проверяется, содержит ли `$checked_out` данные. Если это не так, мы выполняем блок кода, в котором создается HTML ссылки на программу `viewer.cgi`, причем в URL дополнительно передается `file_id`.

**Строки 82—84** создают HTML и сохраняют его в переменной Sfname.

**Строка 85** закрывает блок `unless`.

Функция *unless ()* требует некоторых дополнительных пояснений (для меня, по крайней мере). Она означает "если-не" (if not), это просто Perl-овский способ выражаться. Видя *unless (),* вы должны мысленно поставить на его место "если *не".* Это поможет, если вам трудно запомнить, как работает *unless ().*

```
86:         print qq( $fname<br> );
87:         print qq( <b>Описание:</b> $ptr->
              {description}<br> );
88:         print qq{ $checked_out );
89:         print qq( </td></trx/table> );
90:       }
91:     }
```

**Строки 86—89** создают HTML, который отображает сведения об одном файле. В строке 87 мы снова видим оператор стрелки, с помощью которого мы получаем описание файла.

**Строка** 90 завершает цикл *foreach ()*, начатый в строке 72.

**Строка** 91 завершает цикл *foreach (},* начатый в строке 69.

```
92:     print «HTML;
93:       </td></tr>
94:       <tr><td>
95:       <a href="upload.cgi'?action=add">
            Добавить новый документ</a><br />
96:       <a href="auth.cgi?action=logout">Выход из системы</a>
```

```
97:       </td></tr>
98:       </table>
99:       </body></html>
100:  HTML
101:  }
```

**Строки 92—100** — включенный документ, в котором выводится остальной **HTML** только что созданной нами страницы.

**Строка 101** завершает подпрограмму *Display_Page()*Теперь наша программа raain.cgi. Мы можем войти на сайт через скрипт auth.cgi или вызвав эту программу, main.cgi. Если мы не зарегистрированы, она направит нас в auth.cgi.

# upload.cgi

У нас уже вырисовывается работоспособное приложение. Необходима **еще** программа, позволяющая загружать файлы в систему DMS, и еще одна, с помощью которой мы сможем просматривать файлы из фонда этой системы. Так как не имело бы смысла сначала создавать программу просмотра, если система пуста и у нас нет возможности занести в нее документы, мы в первую очередь рассмотрим программу upload.cgi, а затем viewer.cgi, после чего наше приложение будет готово. Программа upload.cgi — самая большая в нашей системе DMS. Несмотря на это, ее создание также не должно представлять трудностей. Загрузку файлов мы уже рассмотрели в главе 7, и некоторая часть кода взята непосредственно оттуда.

```
01:  #!/usr/bin/perl -wT
02:  # upload.cgi
```

**Строки 1** и **2** — стандартные строки, которые встречаются практически в каждой нашей программе.

```
03:  use DBI;
04:  use strict;
05:  use File::Basename;
06:  use CGI qw(:standard);
07:  use CGI::Carp qw(fatalsToBrowser);
08:  require "./shared.pl" or die "Нельзя найти файл. $!\n"
```

**В строках 3—8** загружаются необходимые модули и файлы и включается прагма strict. Эти строки также должны быть очень хорошо знакомы вам. Единственное новшество — использование модуля File: :Basename.

```
09:  my $user = Check_Cookie();
10:  my $dbh = DB_Connect();
11:  my $group = Get_Group($user);
```

В **строке** 9 мы проверяем, есть ли у пользователя требуемое для этого приложения cookie. Если это не так, подпрограмма *Check_Cookie()* направляет пользователя в программу auth.cgi. Если правильное cookie имеется, подпрограмма возвращает его значение и оно сохраняется в переменной $user.

**Строка 10** создает подключение к базе данных и сохраняет дескриптор в переменной $dbh.

В **строке 11** вызывается подпрограмма *Get_Group()*и полученная группа сохраняется в переменной $group.

```
12:  param('filename') ? Process_File() : Default_Page();
```

**В строке 12** мы с помощью трехместного оператора определяем, что делать дальше. Если значение в левой части будет истинно (истина — любое значение, кроме 0 или

“”), выполняется первый оператор справа от знака ?. Если значение в левой части ложно, выполняется второй оператор. Здесь мы проверяем, передано ли что-нибудь из формы HTML в элементе `filename`.

```
13:  exit;
```

**Строка 13** — просто выход из программы. Если мы дошли до этой строки, это означает, что программа уже закончилась. Но ведь выше мы говорили, что это будет довольно большая программа, а здесь только 13 строк кода! Все дело в подпрограммах, которыми мы сейчас и займемся.

```
14:  sub Get_File_Name {
```

**Строка** 14 начинает подпрограмму *Get_File_Name ()*. Эта подпрограмма взята из главы о загрузке файлов. Она убирает всю информацию о пути и возвращает одно только имя файла.

```
15:     if ($ENV{HTTP_USER_AGENT} =~ /win/i) {
16:        fileparse_set_fstype("MSDOS");
17:     }
```

**В строках 15—17** начинается блок if. . .elsif, в котором мы определяем операционную систему текущего Пользователя. В первой части проверяется, работает ли пользователь в ОС Windows. Если это так, мы выполняем блок кода и вызываем функцию *fileparse_set_fstype()*. Эта функция входит в состав модуля File: :Basename.

```
18:     elsif ($ENV{HTTP_USER_AGENT} =~ /mac/i) {
19:        fileparse_set_fstype("MacOS");
20:     }
```

**Строки 18—20** — часть elsif нашего блока. Здесь мы проверяем, работает ли пользователь в MacOS. Если это так, мы вызываем функцию *fileparse_set_fstype()* с соответствующим параметром. Раздела else в этом блоке нет, только if и elsif. В Perl это вполне допускается. Здесь мы проверяем в HTTP_USER_AGENT только системы Windows и Macintosh, а для всех остальных ОС просто оставляем настройки по умолчанию.

```
21:     my $f_name = shift;
22:     $f_name = basename($f_name);
23:     $f_name =~ s!\s!\_!g;
```

**Строка 21** через функцию *shift ()* получает имя файла, переданное в подпрограмму.

**В строке 22** мы с помощью функции *basename ()* выделяем имя файла из полного имени.

**В строке 23** все пробелы в имени файла заменяются символами подчеркивания. Пробелы создают слишком много трудностей, и замена их на другие символы избавляет от необходимости каждый раз явно заключать имя файла в кавычки.

```
24:     return($f_name);
25:  }
```

**В строке 24** подпрограмма возвращает имя файла без пути и со знаками _ вместо пробелов.

**Строка 25** завершает подпрограмму *Get_File_Name ()*.

```
26:  sub UnTaint {
27:     my $var = shift;
```

**Строка 26** начинает подпрограмму *UnTaint ()*. Эта подпрограмма позволяет гарантировать, что отправленное имя файла не будет содержать никаких недопустимых символов.

**Строка** 27 объявляет переменную **$var** и записывает в нее значение, переданное в подпрограмму.

```
28:     if ($var =~ /^([-\@\w.]+)$/) {
29:       $var = $1;
30:   }
```

**Строка 28** начинает блок if.. .else. Здесь мы отфильтровываем недопустимые символы с помощью регулярного выражения. Допускаются буквы, цифры, символы _, -, @ и точка.

В **строке 29** переменной $var присваивается значение $1. Переменная $1 содержит все, что соответствует выражению в круглых скобках. Таким способом мы очищаем переменную.

**Строка 30** закрывает первую часть блока if. , .else.

```
31:   else {
32:     die "Имя файла загрязнено!\n";
33:   }
```

**Строка 31** начинает часть else блока if. . .else. Она выполняется, если имя файла содержит недопустимые символы.

**Строка 32** производит выход из программы с выводом сообщения об ошибке.

**Строка 33** закрывает блок if.. .else.

```
34:     return($var);
35: }
```

**Строка 34** возвращает из подпрограммы очищенную переменную.

**Строка 35** завершает подпрограмму *unTaint ()*.

```
36: sub Process_File {
37: ray  $description = param( 'description');
38:   my $me_name = param ('filename');
39:   my $area = param('area');
40:   my $mime = uploadInfo($file_name)->{'Content-Type'};
41:   my $path = "/usr/www/cgi-bin/dms/data";
```

**Строка 36** начинает подпрограмму *Process_File*. Эта подпрограмма выполняет все операции, необходимые для получения информации о файле, обеспечивает должное обновление базы данных и загрузку файла.

**В строках 37—39** создается несколько переменных и в них заносится информация, переданная из формы HTML.

В **строке 40** мы с помощью функций *uploadInfo ()* модуля CGI.pm определяем тип MIME загруженного файла. Этот тип будет сохранен в базе данных, благодаря чему при отправке файла пользователю мы сможем указать правильный тип MIME.

**Строка 41** устанавливает каталог, в котором будут храниться файлы.

```
42:     my $file = Get_File_Name($file_name);
```

**В строке 42** мы с использованием функции *Get_File_Name ()* отделяем имя файла от пути.

```
43:     $area = UnTaint($area);
44:     $file = UnTaint($file);
```

**В строках 43—44** производится очистка переменных $area и $file.

```
45:   unless($mime) { $mime = "text/plain"; }
```

В строке 45, если тип MIME не был получен из файла, он принимается как text/plain.

```
46:     my $ptr = Get_File_Info($file, $area);
```

В **строке 46** мы получаем информацию о файле из базы данных и сохраняем указатель на данные в $ptr.

```
47:   unless($ptr->{file_id}) {
```

**Строка 47** начинает блок unless...else. Здесь мы выясняем, не связан ли с этим файлом идентификатор file_id. Если такой идентификатор не найден, это должен быть новый файл, и мы загружаем его и заносим в базу данных.

```
48:       Upload_File($file, $area, $mime, $description);
49:       Add_New_File($file, Sarea, $mime, $description);
50:       print redirect(-uri=>"main.cgi");
51:       exit;
52:   }
```

В **строке** 48 вызывается подпрограмма *Upload_File()*, которую передаются имя файла, область (или группа), к которой файл относится, тип MIME и описание файла.

В **строке** 49 вызывается подпрограмма *Add_New_File()*, в которую передаются те же данные.

**Строка 50** направляет пользователя в программу main.cgi. В этот момент файл уже загружен и добавлен в базу данных, поэтому мы спокойно можем вернуть пользователя на главную страницу сайта.

Строка 51 — выход из программы.

**Строка** 52 завершает первую часть блока unless...else.

```
53:   else {
```

**Строка 53** начинает вторую часть блока unless...else.

```
54:     unless($ptr->{who_to}) {
55:       print redirect(-uri=>"main.cgi");
56:       exit;
57:     }
```

**Строка 54** начинает еще один блок unless...else. Здесь мы выясняем, не выдан ли этот файл какому-то пользователю.

**В строке 55,** если файл не выдан, пользователь направляется назад на страницу main. cgi.

**Строка 56** — выход из программы.

**Строка 57** завершает первую часть блока unless...else.

```
58:     else {                  # Файл *уже* выдан.
59:       if($ptr->{who_to} eq $user) {
60:         Upload_File($file, $area, $mime,
              Sdescription);
61:         Check_File_In($file, $area, $mime,
              $description);
62:         print redirect(-uri=>"main.cgi");
63:         exit;
64:       }
```

**Строка 58** — раздел else блока unless...else. Он выполняется, если файл выдан кому-то.

**Строка** 59 проверяет, выдан ли файл текущему пользователю. Если это так, выполняется блок кода.

**В строке 60** вызывается подпрограмма *Upload_File()* и в нее передаются все данные, необходимые для загрузки.

В строке 61 вызывается подпрограмма *Check_File_In()*. Она обновляет базу данных, лишая файл статуса выданного.

**Строка 62** возвращает пользователя в программу `main.cgi`.

**Строка** 63 — выход из программы.

**Строка 64** закрывает первую часть блока `if...else`.

```
65:         else {
66:           print redirect (-uri=>"main.cgi");
67:           exit;
68:         }
```

**Строка 65** начинает раздел `else` блока `if...else`.

**Строка 66** направляет пользователя в программу main. cgi.

**Строка** 67 — выход из программы.

Строка 68 завершает блок `if...else`.

```
69:       }  # конец unless...else
70:     }
71:     exit;
72:  }
```

**Строка 69** завершает внутренний блок `unless...else`.

**Строка 70** завершает внешний блок `unless...else`.

**Строка 71** — выход из программы.

**Строка 72** завершает подпрограмму *Process_File()*.

```
73:  sub Check_File_In {
74:    $dbh->do( " UPDATE TABLE dms_files SET
75:                (filename, group_id, mime_type, description,
                   who_to, out_date)
76:                VALUES
77:                (?, ?, ?, ?, 'NULL', 'NULL') ", {}, (@_) );
78:  }
```

**Строка 73** начинает подпрограмму *Check_File_In()*. Эта подпрограмма предназначена для обновления информации о файле в базе данных.

**В строках 74—77** формируется и выполняется команда SQL, которая производит обновление в таблице. Выражение {?, ?, ?, ?, 'NULL', 'NULL') ", {}, (@_) ) выглядит необычно, поэтому сейчасмы попробуем расшифровать его. Вопросительные знаки — это просто метки-заполнители базы данных, которые мы уже использовали много раз. NULL — пустые значения для двух полей, которые мы очищаем, чтобы программа считала, что этот файл уже возвращен пользователем. Символ двойной кавычки завершает строку, которая передается в функцию do *()*. Символы {} — пустой хэш. Этот параметр в MySQL не используется, но его тем не менее надо передать. Следующий элемент, `@_` — данные, которые мы передали в подпрограмму. Они включают четыре параметра: `$file`, `$area`, `$mime` и `$descriptipn`. В момент начала подпрограммы все они хранятся в массиве `@_`. Таким образом мы передаем четыре значения на место меток-заполнителей и уходим от необходимости создавать для них отдельные переменные.

Строка 78 завершает подпрограмму *Check_File_In()*.

```
79:  sub Add_New_File {
80:    $dbh->do( "INSERT INTO dms_files
81:               (filename, group_id, mime_type, description)
82:                VALUES
83:              (?, ?, ?, ?) ", {}, (@_) );
84:  }
```

**Строки 79—84** — подпрограмма *Add_New_File()*. Она работает почти так же, как и подпрограмма *Check_File_In()*Единственное существенное отличие— здесь не нужно передавать в базу данных последние два поля, так как они уже имеют значение NULL по умолчанию.

```
85:  sub Upload_File {
86:     my ($file, $area, $mime, $description) = @_;
87:     my $file_name = param ('filename');
88:     my $path = "/usr/www/cgi-bin/book/dms/data";
```

**Строка** 85 начинает подпрограмму *Upload_Fil()*.

**Строка** 86 считывает значения, переданные в подпрограмму, **и** сохраняет их в нескольких переменных.

Строка 87 получает имя файла, переданное из формы HTML.

**Строка** 88 устанавливает путь к каталогу, в котором будут храниться загруженные файлы.

```
89:     my $data;
```

**В строке 89** объявляется переменная $data.

```
90:     $area = UnTaint($area);
91:     Sfile = UnTaint($file);
```

**В строках 90—91** производится очистка переменных **$area** и Sfile.

```
92:     open(VAULT, ">$path/$area/$file")
93:       or die "Ошибка при открытии файла: $!\n";
```

**В строке 92** мы открываем для записи файл в подкаталоге $path/$area. Для имени файла принимается значение из $file.

Строка 93— конструкция *or die()*, которая позволяет нам убедиться, что файл открыт успешно.

```
94:     unless($mime =~ /text/)  {
95:       binmode($file_name);
96:       binmode(VAULT);
97:     }
```

В **строке** 94 мы выясняем, содержит ли тип MIME слово text. Если это слово нельзя найти, данный файл, скорее всего, двоичный, и поэтому функцией *binmode ()* мы устанавливаем двоичный режим для обоих файлов. Функция *binmode {)* не действует в системах Unix; эти строки предназначены для пользователей Windows.

**Строки 95 и 96** устанавливают двоичный режим для дескрипторов файлов $file_name И VAULT.

**Строка 97** закрывает блок unless.

```
98:     while( read($file_name, $data, 1024)  )  {
99:       print VAULT $data;
100:    }
```

**Строка 98** — цикл *while ()*, в котором мы читаем файл порциями по 1024 байта. Каждый раз данные временно сохраняются в переменной Sdata.

В **строке** 99 текущая порция данных записывается в другой файл.

**Строка 100** завершает цикл *while ()*.

```
101:    close VAULT;
102: }
```

**Строка 101** закрывает дескриптор файла **VAULT**.

**Строка 102** завершает подпрограмму *Upload_File ()*.

```
103: sub Get_File_Info {
104:   my ($file, $group) = @_;
```

**Строка 103** начинает подпрограмму *Get_File_Info ()*, которая служит для получения из базы данных информации о файле.

В **строке 104** считываются значения, переданные при вызове подпрограммы.

```
105:   my $sth = $dbh->prepare( qq{ SELECT * FROM dms_files
                      WHERE
106:                  ((filename = ?) AND (group_id = ?))  } );
```

**В строках 105—106** мы создаем и подготавливаем команду SQL для получения данных о файле из базы.

```
107:   $sth->execute($file, $group);
```

**Строка 107** выполняет команду SQL, которую мы только что создали.

```
108:   my $ptr = $sth->fetchrow_hashref;
```

**Строка 108** извлекает из базы строку данных. Должна быть получена только одна строка, и ссылка на хэш, содержащий ее, возвращается в переменную $ptr.

```
109:   return($ptr);
110: }
```

**Строка 109** возвращает из подпрограммы указатель $ptr.

**Строка 110** завершает подпрограмму *Get_File_Info*.

```
111: sub Default_Page {
112:   my $options;
```

**Строка 111** начинает подпрограмму *Default_Page()*.

**В строке 112** объявляется переменная $options.

```
113:   {
114:     $options = qq{ <option value="PEON">
             Рабочий</option> };
115:       last if($group eq "PEON");
116:     $options .= qq{ <option value="USER">
             Пользователь</option> };
117:       last if($group eq "USER"};
118:    $options .= qq{ <option value="PHB">
             Управляющий</option> };
119:      last if($group eq "PHB");
120:    $options .= qq{ <option value="BOFH">
             Босс</option> };
121:  }
```

**Строки 113—121** представляют некоторый интерес. Здесь мы хотим отобразить на странице загрузки раскрывающийся список, в котором пользователь должен будет выбрать уровень прав доступа. Мы последовательно записываем в строку код пунктов списка, начиная с самого низкого уровня доступа и до уровня, который имеет сам пользователь, для чего на каждом шаге сравниваем группу пользователя $group с только что выведенной группой. Если эти группы совпадают, мы выходим из блока. Все эти операторы должны быть заключены в блок, иначе последний из них не будет выполнен. В результате у нас получается строка с кодом раскрывающегося списка, который будет выведен на странице загрузки.

```
122:   print header;
```

**Строка 122** выводит стандартный заголовок **HTTP.**

```
123:    print «HTML;
124:      <html><head>
125:      <title>Пример DMS - загрузка файла</title></head>
126:       <body bgcolor="#ffffff">
127:      <center>
128:      <form method="post" ENCTYPE="multipart/form-data">
129:     <p>
130:     <h2>Добро пожаловать на страницу загрузки</h2>
131:    </p>
132:    <p>
133:     <table border="1" cellspacing="0">
134:      <tr>
135:      <td>Имя файла:</td>
136:      <td><input type="file" name="filename"></td>
137:     </tr>
138:      <tr>
139:      <td>Область:</td>
140:      <td>
141:      <select name="area">
142: HTML
```

**Строки 123—142** — включенный документ, в котором выводится начало страницы загрузки.

```
143:    print $options;
```

**Строка 143** выводит переменную $options. Эта переменная содержит код пунктов раскрывающегося списка.

```
144:    print <<HTML;
145:      </select>
146:      </td>
147:      </tr>
148:      <tr>
149:      <td>Описание:</td>
150:      <td>
151:      <textarea name="description" cols="40" rows="4"
                          wrap="physical"x/textarea>
152:      </td>
153:      </tr>
154:      <tr align="center">
155:      <td colspan="2"><input type="submit"
            value="    Загрузить файл    "></td>
156:      </tr>
157:      </table>
158:      </p>
159:      </form>
160:      </center>
161:      </body>
162:      </html>
163:    HTML
```

**Строки 144—163** — включенный документ, в котором выводится остальная часть HTML страницы загрузки.

```
164:    exit;
165: }
```

**Строка 164** — выход из программы.

**Строка 165** завершает подпрограмму *Default_ Page()*.

Теперь наша страница загрузки готова. Вы можете проверить ее в работе, если хотите. Нам осталось написать еще одну программу, чтобы полностью завершить систему DMS. Этот последний скрипт предназначен для того, чтобы пользователи могли просматривать файлы, находящиеся в системе DMS. Он называется `viewer.cgi` и довольно невелик, занимая всего около 75 строк.

# viewer.cgi

Программа `viewer.cgi` не только предоставляет файлы пользователям, но и дает возможность проверить, выдан ли документ пользователю или уже возвращен.

```
01:  #!/usr/bin/perl -wT
02:  # viewer.cgi
03:  use DBI;
04:  use strict;
05:  use CGI qw(:standard);
06:  require "./shared.pl" or die "Нельзя найти файл. $!\n";
```

**Строки 1–6** вам уже многократно приходилось видеть. Их назначение такое же, что и в предыдущих программах.

```
07:  my $file = param('file');
08:  my $action = param('action');
```

**Строки 7 и 8** принимают данные из формы HTML и сохраняют их в переменных.

```
09:  my $user = Check_Cookie();
10:  my $dbh = DB_Connect();
```

В **строке** 9 мы проверяем cookie, чтобы узнать, вошел ли пользователь в систему. Результат сохраняется в переменной `$user`.

**Строка** 10 создает подключение к базе данных и сохраняет дескриптор в переменной $dbh.

```
11:  my $f_ptr = Get_File_Info_ID($file);
12:  my $group = Get_Group($user);
```

**В строке 11** мы получаем указатель на информацию о файле и записываем его в переменную `$f_ptr`.

**В строке 12** определяется группа, к которой принадлежит пользователь, и сохраняется в переменной $group.

```
13:  my %g_hash = ( 'BOFH' => 4, 'PHB' => 3,
14:                 'USER' => 2, 'PEON' => 1);
```

**В строках 13-14** создается хэш `%g_hash`, заполненный численными значениями групп. По этим значениям мы будем определять, имеет ли пользователь должные права для просмотра файла. Чем больше число, тем больше прав у пользователя.

```
15:  my $u_val = $g_hash{$group};
16:  my $f_val = $g_hash{$f_ptr->{group_id}};
```

**В строке 15** определяется численное значение группы пользователя. Например, если текущий пользователь принадлежит к группе `PHB`, значение `$u_val` для него будет 3.

В **строке 16** мы получаем значение `group_id`, которое связано с этим файлом в базе данных. Это значение присваивается переменной $f_val.

```
17:  print redirect(-uri=>"auth.cgi")
18:    unless($u_val >= $f_val);
```

**Строки 17–18** направляют пользователя на страницу auth.cgi, если значение группы пользователя меньше значения группы файла. В системе профессионального уровня в этом месте целесообразно выводить страницу, сообщающую пользователю о причинах проблемы.

```
19:  if (Saction eq "view") {
20:     View_File();
21:     exit;
22:  }
```

**Строка 19** начинает блок if.. .elsif..elsif__else, в котором выясняется, какое действие должно быть выполнено. В первом условии мы сравниваем параметр $action со значением view.

**Строка 20** выполняется, если выбрано действие view; здесь мы вызываем подпрограмму *View_File()*.

**Строка 21** — выход из программы.

**Строка 22** закрывает первую часть блока.

```
23:  elsif (Saction eq "checkout") {
24:     Check_Out();
25:     print redirect(-uri=>"main.cgi");
26:     exit;
27:  )
```

**В строке 23** проверяется, не выбрано ли действие checkout. Если это так, выполняется блок кода.

**В строке 24** вызывается подпрограмма *Check_Out()*, которая выдает файл текущему пользователю.

**Строка 25** возвращает пользователя в программу main. cgi.

**Строка 26** — выход из программы.

**Строка 27** закрывает эту часть блока.

```
28:  elsif ($action eq "checkin") {
29:     Check_In();
30:     print redirect(-uri=>"main.cgi");
31:     exit;
32:  }
```

В строке 28 проверяется, не выбрано ли действие checkin. Если это так, выполняется блок кода.

В **строке** 29 вызывается подпрограмма *Check_Int()*, которая обновляет базу данных и учитывает файл как возвращенный.

**Строка** 30 направляет пользователя в программу main. cgi.

Строка **31** — выход из программы.

Строка 32 закрывает эту часть блока.

```
33:  else {
34:     print redirect(-uri=>"main.cgi");
35:     exit;
36:  }
```

**Строка** 33 выполняется, если ни одно из предыдущих условий не выполнено.

В строке 34 мы должны просто вернуть пользователя назад на главную страницу.

**Строка** 35 — выход из программы.

Строка 36 завершает весь блок if.. .elsif.. .elsif.. .else.

```
37:  sub Check_Out {
38:    my $time = time();
```

**Строка 37** начинает подпрограмму *Check_Out()*.

**В строке 38** в переменную **$time** записывается текущее время.

```
39:    $dbh->do( qq{ UPDATE dms_files
40:                   SET
41:                  who_to = '$user',
42:                   out_date = '$time'
43:                   WHERE
44:                  file_id = '$file' } );
```

В **строках 39-44** мы формируем команду SQL для обновления таблицы базы данных и выполняем ее.

```
45:    print redirect(-uri=>"main.cgi");
46:  }
```

**Строка 45** направляет пользователя в программу **main.cgi**.

**Строка 46** завершает подпрограмму *Check_Out()*.

```
47:  sub Check_In {
```

**Строка 47** начинает подпрограмму *Check_In()*.

```
48:    $dbh->do( qq{ UPDATE dms_files
49:                   SET
50:                  who_to = NULL,
51:                  out_date = NULL
52:                   WHERE
53:                  file_id = '$file' } );
```

**В строках 48-53** формируется и выполняется команда SQL для обновления базы данных.

```
54:    print redirect(-uri=>"main.cgi");
55:  )
```

**Строка 54** направляет пользователя в программу main. cgi.

**Строка 55** завершает подпрограмму *Check_In()*.

```
56:  sub View_File {
57:    my $data;
58:    my $filepath =
         "data/$f_ptr->{group_id}/$f_ptr->{filename}";
```

**Строка 56** начинает подпрограмму *View_File()*. Эта подпрограмма служит для того, чтобы выбрать требуемый документ и отправить его пользователю.

В **строке 57** создается переменная $data, которая понадобится нам позже.

В **строке** 58 определяется имя файла и каталог, в котором он хранится.

```
59:    print redirect(-uri=>"main.cgi")
         unless $f_ptr->{filename};
60:    open(FILE, $filepath)
         or die "Ошибка при открытии файла! $!\n";
```

**Строка 59** направляет пользователя на главную страницу, если файла с таким именем не существует.

**Строка 60** открывает файл для чтения.

```
61:    print header ($f_ptr->{mime_type});
```

В **строке** 61 выводится заголовок с типом MIME, который был зафиксирован при загрузке файла.

```
62:     while( read(FILE, $data, 1024) ) {
63:       print $data;
64:     }
```

**Строки 62–64** образуют цикл *while()*, в котором файл считывается порциями по 1024 байта, которые сохраняются в переменной $data а затем выводятся.

```
65:     close FILE;
66:   }
```

**Строка 65** закрывает файл.

**Строка 66** завершает подпрограмму *View_File()*.

```
67:   sub Get_File_Info_ID {
68:     my $file_id = shift;
```

**Строка** 67 начинает подпрограмму *Get_File_Info_ID()*. Эта подпрограмма извлекает из базы данных сведения о файле, идентификатор которого передается в **URL.**

**В строке 68** объявляется переменная $file_id и в нее записывается значение, переданное в подпрограмму.

```
69:     my Ssth = $dbh->prepare( qq{ SELECT *
70:                                  FROM dms_files
71:                                  WHERE file_id - ? } );
```

**Строки 69—71** формируют и подготавливают команду SQL, которая выбирает из таблицы информацию о файле.

```
72:     $sth->execute($file_id);
```

**В строке 72** выполняется команда SQL, которую мы только что создали.

```
73:     my $ptr = $sth->fetchrow_hashref;
74:     return Sptr;
75:   }
```

**В строке 73** из базы извлекается строка данных и сохраняется в новой переменной под названием $ptr.

В **строке 74** значение $ptr возвращается из подпрограммы.

**Строка 75** завершает подпрограмму *Get_File_Info_ID()*.

# Листинги

Листинг 14.1. **auth.cgi**

```
01: #!/usr/bin/perl -wT
02: # auth.cgi
03: use DBI;
04: use strict;
05: use CGI qw(:standard);
06: require "./shared.pl" or die "Нельзя найти файл. $!\n";
07: my $user = param('user-name');
08; my Spass = param('password');
09: my $action = param('action');
10: my $dbh = DB_Connect();
11: Logout() if ($action eq "logout");
12: Login_Pagef) unless($pass);
13: my $valid = Check_Login($user, Spass);
14: if (Svalid) {
15:   my $cookie = Create_Cookie("lh", $user);
16:   print redirect(-uri=>"main.cgi", -cookie=>$cookie);
```

```
17:     exit;
18:   }
19:   else {
20:     my $time = time();
21:     print redirect(-uri=>"auth.cgi?$time");
22:     exit;
23:   }
24:   sub Logout {
25:     my $time = time();
26:    my $cookie = Create_Cookie("-1h", " " ) ;
27:     print redirect(-uri=>"auth.cgi?$time", -cookie=>$cookie);
28:     exit;
29:   }
30:   sub Create_Cookie {
31:     my Sexp = shift;
32:     my $val = shift;
33:     my $cookie = cookie(-name => 'dms',
34:                         -value => $val,
35:                         -expires => $exp );
36:     return($cookie);
37:   }
38:   sub Check_Login {
39:     my $user = shift;
40:     my $pass = shift;
41:     my $data;
42:     my $sth = $dbh->prepare( qq( SELECT password
43:                                  FROM dms_users
44:                                  WHERE
45:                                  usemame = ?
46:                                } ) ;
47:     $sth->execute($user);
48:     $data = $sth->fetch;
49:     $sth->finish;
50:     $dbh->discormect;
51:     ($data~>[0] eq $pass)  ? return 1  : return 0;
52:   }
53:   sub Login_Page {
54:     print header!);
55:   print «HTML;
56:     <html><head><title>Пример DMSe</title></head>
57:     <body bgcolor="#ffffff">
58:     <center>
59:     <form method="post">
60:    <p>
61:     <h2>Вас приветствует пример DMS</h2>
62:    </p>
63:    <p>
64:     <table border="1" cellspacing="0">
65:     <tr>
66:     <td>Имя пользователя:</td>
67:     <td><input type="text" name="username"></td>
68:     </tr>
69:     <tr>
70:     <td>Пароль:</td>
71:      <td><input type="password" name="password"x/td>
72:     </tr>
73:     <tr align="center">
74:     <td colspan="2">
75:     <input type="submit" value="    Войти    ">
76:     </td>
77:     </tr>
```

```
78:     </table>
79:     </p>
80:     </form>
81:     </center>
82:     </body>
83:     </html>
84:   HTML
85:     exit;
86:   }
```

**Листинг 14.2. shared.pl**

```
01:  sub DB_Connect {
02:    my $dbh = DBI->connect("DBI:mysql:book","book","addison")
03:      or die "Ошибка при подключении к БД! $DBI::errstr\n";
04:    return($dbh);
05:  }
06:  sub Check_Cookie {
07:    my Suser = cookie('dms');
08:    if ($user) {
09:      return($user) ;
10:    }
11:    else {
12:      print redirect(-uri=>"auth.cgi");
13:      exit;
14:    }
15:  }
16:  sub Get_Group {
17:    my $user = shift;
18:    my $dbh = DBI->connect("DBI:mysql:book","book","addison")
19:      or die "Ошибка при подключении к БД! $DBI::errstr\n";
20:    my $sth = $dbh->prepare( qq{ SELECT group_id
21:                                 FROM dms_users
22:                                 WHERE
23:                                 username = ?
24:                               } ) ;
25:    $sth->execute($user) ;
26:    $group = $sth->fetch->[0];
27:    redirect(-uri=>"auth.cgi") unless $group;
28:    $sth->finish;
29:    $dbh->disconnect;
30:    return($group);
31:  }
32:  1;
```

**Листинг 14.3. main.cgi**

```
01:  #!/usr/bin/perl -wT
02:  # main.cgi
03:  use strict;
04:  use CGI qw(:standard) ;
05:  use CGI::Carp qw(fatalsToBrowser);
06:  use DBI;
07:   require "./shared.pl" or die "Нельзя найти файл. $!\n";
08:  my $user = Check_Cookie();
09:  my $dbh = DB_Connect();
10:  my Sgroup = Get_Group($user);
11:  my $action = param('action');
12:  my $file   = param('file');
```

```
13:  my $out_to = param('out_to');
14:  my %files;
15:  my @groups - qw(PEON USER PHB BOFH);
16:  foreach (@groups) {
17:    $files{$_} = Get_Files($_);
18:    last if (Sgroup eq "$_");
19:  }
20:  Display_Page();
21:  sub Get_Files {
22:    my $group = shift;
23:    my @temp;
24:    my $sth = $dbh->prepare( qq{ SELECT *
25:                                 FROM dms_files
26:                                WHERE group_id = ? } );
27:    $sth->execute($group);
28:    while {my $ptr = $sth->fetchrow_hashref) {
29:      push @temp, $ptr;
30:    }
31:    return \@temp;
32:  }
33:  sub Check_Status {
34:    my ($who_to, $date_out, $file_id} = @_;
35:    my Sdata;
36:    return unless($who_to);
37:    my $sth = $dbh->prepare( qq{ SELECT phone, e_mail
38:                                 FROM dms_users
39:                                WHERE username = ? } );
40:    $sth->execute($who_to);
41:    my $ptr = $sth->fetch;
42:    if ($who_to eq $user) {
43:      $data = "<b>Выдано:</b>";
44:      $data .= "<a href='viewer.cgi?action=checkin";
45:      $data .= "&file=$file_id'>Вернуть</a><br>";
46:    }
47:    else {
48:      $data = "<b>Кому выдано:</b>";
49:      Sdata .= "<a href='mailto:$ptr->[1]'>$who_to</a>";
50:      $data .= "~ Тел.: $ptr->[0]<br>";
51:    }
52:    return($data);
53:  }
54:  sub Display_Page {
55:    print header;
56:    print «HTML;
57:      <html><head>
58:      <title>Система управления flOKyMeHTaMM</title>
59:      </head>
60:      <body bgcolor="#ffffff">
61:      <center>
62:      <table border="1" width="60%">
63:      <tr><td align="center">
64:      <h2>Простая система управления документами</h2>
65:      <b>Добро пожаловать, $user!</bxbr>
66:      </td></tr>
67:      <trxtd valign='top'>
68:  HTML
69:    foreach my $grp (keys %files) (
70:      next unless(@{$files{$grp}});
71:      print qq ( <p><b>Область $grp</bxbr>\n ) ;
72:      foreach my Sptr (@{$files{$grp}}) {
73:        my $fname;
```

```
74:        my $checked_out = Check_Status($ptr->{who_to},
75:                                       $ptr->{out_date},
                                          $ptr->{file_id});
76:         print qq( <table bgcolor="#e0e0e0" width="100%">
              <tr><td> );
77:        print qq( <b>Имя файла:</b> );
78:          $fname = qq(<a href="viewer.cgi?action=);
79:          $fname .= qq(view&file=$ptr->{file_id}">);
80:          $fname .= qq($ptr->{filename}</a>);
81:        unless($checked_out) {
82:          $fname .= qq( [<a href="viewer.cgi?action=);
83:          Sfname .= qq(checkout&file=$ptr->{file_id}">) ;
84:          Sfname .= qq(Выдать</a>]) ;
85:        }
86:        print qq( $fname<br> );
87:        print qq( <Ь>Описание:</b> $ptr->
             tdescription}<br> );
88:        print qq( $checked_out );
89:        print qq( </td></tr></table> );
90:      }
91:    }
92:     print «HTML;
93:       </tdx/tr>
94:       <trxtd>
95:       <a href="upload.cgi'?action=add">
            Добавить новый документ</a><br />
96:      <a href="auth. cgi?action=logout">Выход из системы</a>
97:       </td></tr>
98:       </table>
99:       </body></html>
100:  HTML
101: }
```

**Листинг 14.4. upload.cgi**

```
01:  #!/usr/bin/perl -wT
02:  # upload.cgi
03: use DBI;
04:  use strict;
05:  use File::Basename;
06:  use CGI qw(:standard);
07:  use CGI::Carp qw(fatalsToBrowser);
08:  require "./shared.pl" or die "Нельзя найти файл. $!\n"
09:  my $user = Check_Cookie();
10:  my $dbh  = DB_Connect();
11:  my $group = Get_Group($user);
12:  param('filename') ? Process_File() : Default_Page();
13:  exit;
14:  sub Get_File_Name {
15:    if ($ENV{HTTP_USER_AGENT} =~ /win/i) {
16:       fileparse_set_fstype("MSDOS");
17:    }
18:    elsif ($ENV{HTTP_USER_AGENT} =~ /mac/i) {
19:       fileparse_set_fstype("MacOS");
20:    }
21:    my $f_name = shift;
22:    $f_name    = basename ($f_name) ;
23:     $f_name    =~s!\s!\_!g;
24:    return($f_name);
25: }
```

```
26:  sub UnTaint {
27:  ray  $var = shift;
28:    if ($var =~  /^([-\@\w.]+)$/)  (
29:       $var = $1;
30:     }
31:     else {
32:       die "Имя файла загрязнено!\n";
33:     }
34:      return($var);
35:  }
36:  sub Process_File {
37:  ray  Sdescription = param('description');
38:     my $me_name = param ('filename');
39:     my $area = param('area');
40:     my $mime = uploadInfo($file_name)->{'Content-Type'};
41:  ray  Spath = "/usr/www/cgi-bin/dms/data";
42:     my Sfile = Get_File_Name ($file_name);
43:     $area = UnTaint($area);
44:     $file = UnTaint($file);
45:     unless($mime)  { $mime = "text/plain"; }
46:     my $ptr = Get_File_Info($file, $area);
47:     unless($ptr->{file_id})  {
48:       Upload_File($file, $area, $mime, $description);
49:       Add_New_File($file, $area, $mirae, Sdescription);
50:        print redirect(-uri=>"main.cgi");
51:        exit;
52:     }
53:     else {
54:       unless($ptr->{who_to})  {
55:          print redirect(-uri=>"main.cgi");
56:          exit;
57:        }
58:        else {                  # Файл *уже* выдан.
59:         if($ptr->{who_to} eq $user)  {
60:           Upload_File($file, $area, $mime,
                Sdescription);
61:           Check_File_In($file, Sarea, $mime,
                Sdescription);
62:            print redirect(-uri=>"main.cgi");
63:          • exit;
64:          }
65:          else {
66:            print redirect  (-uri=>"main.cgi");
67:            exit;
68:          }
69:       }  # конец unless...else
70:     }
71:     exit;
72:  }
73:  sub Check_File_In {
74:     $dbh->do(  " UPDATE TABLE dms_files SET
75:                  (filename, group_id, mime_type, description,
                     who_to, out_date)
76:                  VALUES
77:                 (?,?,?,?,  'NULL',  'NULL')  ",  {},  (@_)  );
78:  }
79:  sub Add_New_File {
80:     $dbh->do(  "INSERT  INTO dms_files
81:                  (filename, group_id, mime_type, description)
82:                  VALUES
83:                (?,  ?,  ?,  ?)  ",  {},  (@_)  );
```

```
84: }
85:  sub Upload_File {
86:    my ($file, $area, $mime, $description) = @_;
87:    my $file_name = parara ('filename');
88:    my $path = "/usr/www/cgi-bin/book/dms/data";
89:    my $data;
90:    $area = UnTaint($area);
91:    $file = UnTaint($file);
92:    open(VAULT, ">$path/$area/$file")
93:      or die "Ошибка при открытии файла: $!\n";
94:    unless($mime =~ /text/) {
95:      binmode($file_name);
96:      binmode(VAULT);
97:    }
98:   while( read($file_name, $data, 1024) ) {
99:      print VAULT $data;
100:  }
101:   close VAULT;
102: }
103: sub Get_File_Info {
104:  ray  ($file, $group) = @_;
105:   my $sth = $dbh->prepare( qq{ SELECT * FROM dms_files
                    WHERE
106:                  ((filename = ?) AND (group_id = ?}) } );
107:   $sth->execute($file, $group);
108:   my $ptr = $sth->fetchrow_hashref;
109:   return($ptr);
110: }
111: sub Default_Page {
112:   my $options;
113:  {
114:     $options = qq{ <option value="PEON">
           Рабочий</option> };
115:       last if($group eq "PEON");
116:     $options .= qq{ <option value="USER">
           Пользователь</option> };
117:       last if($group eq "USER");
118:     $options .= qq{ <option value="PHB">
           Управляющий</option> };
119:       last if($group eq "PHB");
120:     $options .= qq{ <option value="BOFH">
           Босс</option> };
121:  }
122:   print header;
123:   print «HTML;
124:     <htmlxhead>
125:     <title>Пример DMS - загрузка файла</title></head>
126:     <body bgcolor="#ffffff">
127:     <center>
128:     <form method="post" ENCTYPE="multipart/form-data">
129:    <p>
130:     <h2>Добро пожаловать на страницу загрузки</h2>
131:    </p>
132:    <p>
133:     <table border="1" cellspacing="0">
134:     <tr>
135:     <td>Имя файла:</td>
13 6:  <td><input  type="file"  name="filename"></td>
137:     </tr>
138:     <tr>
139:     <td>Область:</td>
```

```
140:      <td>
141:      <select name="area">
142: HTML
143:    print $options;
144:    print «HTML;
145:      </select>
146:      </td>
147:      </tr>
148:      <tr>
149:      <td>Описание:</td>
150:      <td>
151:      <textarea name="description" cols="40" rows="4"
                        wrap="physical"></textarea>
152:      </td>
153:      </tr>
154:      <tr align="center">
155:      <td colspan="2"xinput type="submit"
            value="   Загрузить файл   "></td>
156:      </tr>
157:      </table>
158:      </p>
159:      </form>
160:      </center>
161:      </body>
162:      </html>
163:  HTML
164:    exit;
165:  }
```

**Листинг 14.5. viewer.cgi**

```
01:  #!/usr/bin/perl -wT
02 : # viewer.cgi
03: use DBI;
04:  use strict;
05:  use CGI qw(:standard);
06:  require "./shared.pl" or die "Нельзя найти файл. $!\n";
07:  my $file = param('file');
08:  my $action = param('action');
09:  my $user = Check_Cookie () ;
10:  my $dbh = DB_Connect();
11:  my $f_ptr = Get_File_Info_ID($file) ;
12:  my Sgroup = Get_Group($user);
13:  my %g_hash = ( 'BOFH' => 4, 'PHB' => 3,
14:                 'USER' => 2, 'PEON' => 1);
15:  my $u_val = $g_hash{$group};
16:  my $f_val = $g_hash{$f_ptr->{group_id}};
17:  print redirect(-uri=>"auth.cgi")
18:    unless($u_val >= $f_val);
19:  if ($action eq "view") {
20:    View_File();
21:    exit;
22: }
23:  elsif {Saction eq "checkout") (
24:    Check_Out();
25:    print redirect(-uri=>"main.cgi");
26:    exit;
27: }
28:  elsif ($action eq "checkin") {
29:    Check_In();
```

```
30:    print redirect(-uri=>"main.cgi");
31:    exit;
32:  }
33:  else {
34:    print redirect(-uri=>"main.cgi");
35:    exit;
36:  }
37:  sub Check_Out {
38:    my $time = time();
39:    $dbh->do( qq{ UPDATE dms_files
40:                  SET
41:                  who_to = '$user',
42:                  out_date = '$time'
43:                  WHERE
44:                  file_id = '$file' } );
45:    print redirect(-uri=>"main.cgi");
46:  }
47:  sub Check_In {
48:    $dbh->do( qq{ UPDATE dms_files
49:                  SET
50:                  who_to = NULL,
51:                  out_date = NULL
52:                  WHERE
53:                  file_id = '$file' } ) ;
54:    print redirect(-uri=>"main.cgi");
55:  }
56:  sub View_File (
57:    my $data;
58:    my $filepath =
       "data/$f_ptr->{group_id}/$f_ptr->{filename}";
59:    print redirect(-uri=>"main.cgi")
         unless $f_ptr->{filename};
60:    open(FILE, $filepath)
         or die "Ошибка при открытии файла! $!\n";
61:    print header ($f_ptr->{mime_type});
62:   while( read(FILE, $data, 1024) ) <
63:      print $data;
64:    }
65:    close FILE;
66:  }
67:  sub Get_File_Info_ID {
68:    my $file_id = shift;
69:    my $sth = $dbh->prepare( qq( SELECT *
70:                                 FROM dms_files
71:                                WHERE file_id = ? } );
72:    $sth->execute($file_id);
73:    my $ptr = $sth->fetchrow_hashref;
74:    return $ptr;
75:  }
```


# Динамическая обработка изображений

## Введение

В этой главе мы рассмотрим некоторые базовые методы обработки изображений с использованием Perl. Как будет показано в примерах, динамическую обработку изображений можно реализовать и с помощью специализированных скриптов для уже имеющихся изображений. Примеры этой главы, охватывая некоторые основные функции, которыми вы можете воспользоваться, показывают, как легко выполнить эту задачу с помощью Perl. В этих примерах вы научитесь быстро создавать диаграммы, формировать эскизы, обрабатывать изображения фильтрами, создавать анимированные изображения и вставлять в изображения текст и графические фигуры.

В первую очередь мы уделим внимание модулям Perl `GD.pm`[1] и `Image::`Magick.[2] Модуль GD — это API к библиотеке `gd`, написанный на Perl Линкольном Штайном. Библиотека gd предназначена для обработки изображений и написана на C Томасом Боутеллом (Boutell). Она дает разработчикам возможность создавать элементарные геометрические фигуры, такие как многоугольники, дуги и линии, а также добавлять текст и цвет. Эта библиотека не охватывает всех возможностей работы с изображениями, но в решении многих задач приносит большую пользу. Модуль image: `:Magick`, с другой

---

[1] *Для работы этого модуля должна быть установлена библиотека `gdlib`. Последние требования можно найти в документации.*

[2] *Этот модуль Per/ предоставляет API к графическому пакету Image Magick. Этот пакет можно бесплатно загрузить на http://www.wizards.dupont.com/cristy/ImageMagick.html; руководство по установке см. в документации.*

стороны, предоставляет **API** к пакету программ Image Magick. Этот пакет — самостоятельная графическая программа широкого назначения, обладающая более расширенными возможностями, например, фильтрами.

# Вставка фигур и текста

Иногда требуется вставить в существующие изображения графические фигуры или текст. Практическим примером этого может служить программа, которую я недавно разработал. Это была система Web-безопасности[3], служебные экраны которой были построены на базе изображения. Это изображение напоминало пульт дистанционного управления (ДУ) и так и называлось. Каждая виртуальная машина в этой системе могла иметь собственный ДУ (все ДУ использовали один и тот же набор скриптов), вследствие чего, в случае многочисленных открытых ДУ, было трудно разобраться Б общей картине и невозможно узнать, какой ДУ какой компьютер представляет. Чтобы прояснить эту ситуацию, на изображение ДУ динамически накладывался текст, содержаший доменное имя виртуальной машины и версию системы безопасности. Благодаря этому пользователи могли легко идентифицировать любой из множества открытых ДУ. Все это было сделано с использованием модуля Perl `GD.pm`.

Пример скрипта, который мы рассмотрим, не предназначен для такой серьезной цели, как система Web-безопасности, но использует такие же принципы, концепцию и методы. Этот скрипт открывает существующий файл изображения, накладывает на него некоторые элементарные фигуры (кружки) и помещает в них текст. Название этого скрипта— `add_text.cgi`. В этом примере я помешаю на фотографию моей очаровательной маленькой дочери[4] пузырьковую выноску и вставляю в нее текст. Моя задача — научить вас добавлять в изображение элементарные фигуры и текст.

```
01:  #!/usr/bin/perl -wT
02:  use strict;
03:  use CD;
04:  no strict 'subs';
```

**В строках 1—4** определяется путь к Perl и загружаются прагма `strict` и модуль GD. Строка 4 указывает, что прагма строгого синтаксиса не должна обращать внимания на отдельные слова (bare words), которые могут выглядеть как имена подпрограмм. Причина в том, что иногда мы будем использовать отдельные слова как дескрипторы файлов, как, например, в строке 6. Если бы мы не сделали этой оговорки, прагма `strict` сгенерировала бы ошибку и прекратила выполнение скрипта.

```
05:  open (PNG, "kyla_smile.png") || die "$!";
06:  my $image = newFromPng CD::Image(PNG) || die "$!";
07:  close PNG;
```

В строках 5, 6 и 7 открывается изображение, которое следует обработать (в данном случае — `kyla_smile.png`), и создается новый объект изображения, связанный с дескриптором этого файла. На **строку 6** надо обратить особое внимание. Здесь мы передаем в метод *`newFromPng()`*, импортированный из GD, параметр `GD::Image(PNG)`. Здесь `PNG` — дескриптор открытого файла изображения. Итак, `GD::Image` читает указанный файл и передает результат в метод *`newFromPng()`*, который в GD служит для декодирования изображения. Если, например, наше изображение имело бы формат XBM, мы должны были бы вызывать для декодирования метод *`newFromXBM()`*. Полученная в результате ссылка на объект сохраняется в переменной `$image`.

---

[3] *Это совершенно отдельное приложение, и, чтобы его рассмотреть, потребовалась бы еще одна книга.*

[4] *Это, конечно, субъективное мнение.*

```
08:  my $white = $image->colorAllocate(255,255,255);
09:  my $black = $image->colorAllocate(0,0,0);
```

**В строках 8 и 9** определяются два цвета: белый (white) и черный (black). Мы уже затрагивали эту тему в главе 7. Метод *colorAllocate()* добавляет новые цвета в таблицу цветов изображений. В него передаются три параметра — значения красной, зеленой и синей компонент (RGB) нового цвета. Возвращаемое значение — индекс этого цвета в таблице цветов изображений; к нему можно обращаться по имени переменной, стоящей слева от знака =. Если в изображении предполагается использовать какой-то цвет, его *необходимо* предварительно определить в таблице цветов.

```
10:  $image->arc(50,50, 95,75, 0,360,$black);
11:  $image->fillToBorder(50, 50, $Ыаск, Swhite) ;
12:  $image->arc(60,95,2 5,25,0, 3 60,$Ыаск) ;
13:  $image->fillToBorder(60,95,$black,$white);
14:  $image->arc(70, 125, 20 , 20 , 0 , 360, $Ыаск) ;
15:  $image->fillToBorder(70,125,$black,$white);
```

**В строках 10—15** мы создаем три разных овала. Каждая из этих фигур создается двумя строками; в первой определяется граница эллипса и ее цвет. Во второй строке созданная фигура заполняется одним цветом. В нашем случае фигуры будут иметь черные границы и будут заполнены белым цветом. В результате получится что-то, напоминающее пузырьки. Для определения размеров и цвета контура мы используем метод *arc()*. В этот метод передается семь параметров. Первые два — координаты X и Y центра эллипса. Третий и четвертый параметры — соответственно ширина и высота эллипса. Пятый и шестой параметры определяют начало и конец дуги эллипса, которая будет построена. Угол дуги отсчитывается в градусах против часовой стрелки. Диапазон от 0 до 360 означает, что эллипс будет построен полностью. Седьмой и последний параметр — цвет границы эллипса, которая будет черной. Результатом выполнения этой строки будет нарисованный на изображении эллипс с черным контуром.

Обычно пузырьковые выноски имеют непрозрачный белый фон и черную границу, поэтому мы должны заполнить построенные эллипсы. Это выполняется с помощью метода *fillToBorder()*. В этот метод передаются четыре параметра. Первый и второй определяют координаты X и Y начальной точки заливки. Третий параметр — цвет границы, пространство до которой будет залито. Метод заполняет плоскость одним цветом, пока не будет достигнута граница, имеющая цвет, указанный в этом параметре. Существует аналогичный метод *fill ()*, также заполняющий плоскость от указанной начальной точки, но только пока не будет достигнут пиксел другого цвета. С помощью *fillToBorder ()* разработчик легко может заполнять одним цветом эллипсы и другие фигуры. Последний параметр — цвет самой заливки. Этот цвет, как и цвет границы, указываемый в третьем параметре, должен быть предварительно определен в таблице цветов изображения. После заполнения трех эллипсов мы получаем готовые "пузырьки".

```
16:  $image->string(gdMediumBoldFont,10,45,
       "Привет, папа!",$black);
```

**Строка 16** помещает на изображение строку текста. Для этого мы используем метод *String ()*. Он принимает пять параметров, первый из которых — тип шрифта для текста. В данном случае мы использовали gdMediumBoldFont, другие возможные шрифты — gdTinyFont, gdSmallFont, gdLargeFont и gdGiantFont. Третий И четвертый параметры метода — координаты X и Y *начала* текста в изображении. Далее следует собственно текстовая строка, которую надо вывести, и, наконец, — цвет текста.

```
17:  print "Content-type: image/png\n\n";
```

**В строке 17** выводится строка типа содержимого заголовка **HTTP.** Так как в броузер передается изображение PNG, мы устанавливаем тип **MIME** image/png, чтобы клиент знал, как обработать поступившие данные.

```
18:  binmode STDOUT;
```

**В строке 18** функцией Perl *binmode()* для стандартного дескриптора STDOUT устанавливается двоичный режим. Вследствие этого данные изображения будут выведены в двоичном виде. Многие системы не различают двоичную и текстовую форму файла и читают и записывают данные обоих типов одним и тем же способом. Однако для некоторых систем, таких как Win32, текстовые файлы отличаются от двоичных. Эта строка повышает мобильность скрипта в предположении, что он может быть запущен в системе любого типа.

```
19:  print $image->png;
```

В **строке** 19 данные изображения записываются в стандартный вывод STDOUT, т.е. в броузер. Метод *png()* объекта изображения преобразует его данные в формат PNG. В этом примере мы выводим изображение непосредственно в STDOUT, не создавая нового файла изображения. Однако разработчик может при желании открыть дескриптор файла и записать данные в файл. Окончательный результат этого примера показан на рис. 15.1.


*Рис. 15.1. Результат работы скрипта add_text.cgi*

# Создание динамической диаграммы

В предыдущем разделе вы научились некоторым элементарным методам обработки изображений с помощью GD. В этом разделе мы покажем, как создать с помощью GD::Graph динамическую диаграмму. В данном примере будет построена столбчатая диаграмма, но этот метод позволяет также создавать линейные, точечные, секторные, линейно-точечные, фигурные и смешанные диаграммы. Он может быть очень полезен, если требуется быстро и без усилий создать диаграмму. Пример этого раздела вполне реален; в нем используются данные, собранные в одном из предыдущих примеров книги. В главе 9 мы пробовали отслеживать количество показов баннеров и щелчков на них. В этом разделе мы усовершенствуем созданное ранее приложение и на основе собранных данных построим динамическую диаграмму, отображающую количество показов и щелчков на пяти баннерах с наиболее высокой популярностью. Этот пример продемонстрирует, как показать информацию из базы данных более наглядно. Наш опыт в сфере IT свидетельствует, что пользователям гораздо больше нравится видеть диаграммы и графики, чем простой текст.

Следующий скрипт называется hits.cgi; его можно будет вызвать непосредственно с сайта через URL http://www.you.com/cgi-bin/hits.cgi, и он отобразит свои результаты прямо в броузере. Этот результат — динамическое изображение; в скрипте не используется и не создается никакой физический файл. Закончив разбор этого скрипта, мы покажем, как можно сохранить изображение в файле, чтобы скрипт мог не только создавать диаграммы, но и сохранять их для последующего сравнения с другими диаграммами.

```
01:  #!/usr/bin/perl -wT
02:  use strict;
03:  use CGI qw(:standard);
04:  use GD::Graph::bars;
```

```
05:  use DBI;
06:  my $dbname = 'book';
07: my $dbhost = 'localhost';,
08:  my @res;
09:  my $dsn = "DBI:mysql:database=$dbname;host=$dbhost";
10:  my $dbh=DBI->connect($dsn,"guest","cheese");
11:  if (!defined($dbh)) {
12:     print header;
13:     print "\nошибка:
          Не удается подключиться к базе данных MySQL:\n";
14:     print DBI->errmsg;
15:     print "-" x 25;
16:     exit;
17: }
```

**В строках 1—17** для вас не должно быть ничего нового. Указав путь к Perl в первой строке, мы загружаем прагму `strict`, модули `CGI.pm`, `GD::Graph::bars` и `DBI.pm`. Впервые в этом примере мы будем использовать для создания диаграммы модуль `GD::Graph::bars`. Модуль `CGI.pm` играет в этом скрипте небольшую роль; он служит лишь для отображения заголовка HTTP и может показаться лишним, но мы оставили его, так как он очень удобен. Также мы используем модуль `DBI` ,pm для получения необходимой информации из базы данных. Как можно видеть здесь, скрипт пробует установить подключение к базе данных и отображает сообщение об ошибке, если это не удается.

```
18:  my $sth = $dbh->prepare(qq{select IMAGE, CLICKS,
       IMPRESSIONS from image_tracker order by CLICKS
       desc limit 5});
```

**В строке 18** с помощью метода *prepare()* из DBl.pm подготавливается команда SQL. Эта команда выбирает из базы данных название изображения, количество щелчков и количество показов изображения. Данные, взятые из таблицы image_tracker[5] сортируются по полю CLICKS в порядке убывания. В результате мы получаем в первую очередь записи с наиболее высоким количеством щелчков. Так как нас интересуют только пять баннеров с наибольшим количеством щелчков, мы ограничиваем число возвращаемых записей числом 5. Дескриптор подготовленной команды записывается в переменную $sth.

```
19:  $sth->execute;
```

**В строке 19** запрос SQL выполняется методом *execute()*.

```
20:  while (my @results = $sth->fetchrow_array> {
21:     push (ires, @results;
22:  }
```

В **строках** 20, 21 и 22 возвращенные строки из базы данных извлекаются методом *fetchrow_array()*. Каждая полученная строка сохраняется в массиве `@results`. В **строке** 21 эти результаты функцией Perl *push ()* дописываются в массив @res, который был объявлен в строке 8. Когда цикл закончится, `@res` будет содержать 15 элементов — по три элемента для каждой записи. Использование этой структуры мы рассмотрим в следующих строках.

```
23:  $dbh->disconnect;
```

**Строка 23** производит отключение от базы данных. Нам уже не нужно сохранять подключение открытым, поэтому мы закрываем его. Следует подчеркнуть, что во всех случаях рекомендуется закрывать подключение к базе данных сразу же после того, как

[5] *Схему этой таблицы можно найти в главе 8.*

из нее будет получена необходимая информация. Чтобы лучше понять, как эта информация будет использоваться в следующих строках, полезно рассмотреть структуру массива @res. Она показана в листинге 15.1 в форме псевдокода.

### Листинг 15,1 Структура массива @res

```
@res = ("foo.png", 150, 175,
        "bar.png", 100, 110,
        "baz.png", 75, 150,
        "zog.png", 72, 88,
        "xyzzy.png", 50, 99);
24: my @data = (
25:  [$res[0],$res[3],$res[6],$res[9],$res[12]],
26:  [$res[1],$res[4],$res[7],$res[10],$res[13]],
27:  [$res[2],$res[5],$res[8],$res[11],$res[14]],
28: );
```

**В строках 24—28** создается новый массив под названием @data. В нем мы перегруппировываем данные массива @res. Новый массив образуют три элемента — три безымянных массива. Первый из этих безымянных массивов содержит названия изображений. Эти данные мы используем для меток на оси X диаграммы. Второй безымянный массив содержит количество щелчков, а третий — количество показов. Данные двух этих массивов будут откладываться на диаграмме по оси Y. Наша мысль станет более понятной, когда мы немного позже придем к методу *GD: -.Graph:: Plot ().*

```
29: my $my_graph - new CD::Graph::bars(640, 480);
```

**В строке 29** мы создаем новый объект GD: :Graph::bars и записываем ссылку на него в переменную $my_graph. В метод *new()* передаются два параметра — ширина и высота нового изображения. По умолчанию, если параметры не передаются, принимаются размеры нового изображения 400x300.

```
30: $my_graph->set_legend("Щелчки", "Показы");
```

**В строке 30** с помощью метода *set_legend()* объекта диаграммы устанавливаются названия меток для легенды. Диаграммы не обязательно должны иметь легенду, но в данном случае мы используем два цвета (какие, вы скоро увидите) и сравниваем два набора данных —- щелчки и показы. Легенда сообщает тому, кто видит диаграмму, какие столбцы изображают щелчки, а какие — показы. Также эта строка демонстрирует, как поместить легенду в диаграмму. Порядок, в котором указаны параметры в этом методе, соответствует порядку расположения безымянных массивов в @data, т.е., сначала идут щелчки, а за ними — показы. Тот же порядок будет соблюден, когда мы определим цвета для столбцов.

```
31: $my_graph->set(
```

**Строка 31** начинает вызов метода *set().* В этот метод мы передадим довольно много пар параметр-значение, поэтому каждая пара будет показана и разобрана в отдельной строке. Метод *set()* служит для установки параметров диаграммы. Эти параметры включают следующие[6];

```
32:     dclrs => [ qw(lgreen lyellow) ],
```

В **строке** 32 устанавливается параметр dclrs (сокращенное datacolors). В этом параметре указывается ссылка на массив названий цветов, которые будут использованы для наборов данных. Здесь также важен порядок, поскольку первый элемент этого

[6] *Список всех параметров, а также информацию о новых параметрах можно найти в документации по модулю.*

массива указывает цвет для первого набора данных (в нашем случае — щелчков), а следующий — цвет для следующего набора данных (показов). В этой диаграмме столбцы щелчков будут изображены светло-зеленым цветом (lgreen), а столбцы показов — светло-желтым (lyellow).[7]

```
33:     title => "Статистика самых популярных баннеров",
```

В **строке** 33 через параметр `title` определяется заголовок диаграммы. Он помещается по центру вверху изображения диаграммы.

```
34:     x_label => "Баннеры",
```

**Строка 34** определяет текст названия оси X. Для этого служит параметр `x_label`.

```
35:     y_label => "Количество",
```

**В строке 35** параметру y_label присваивается строковое значение "Количество". Этот параметр подобен предыдущему, только устанавливает название для оси Y.

```
36:     long_ticks => 1,
```

**В строке 36** устанавливается значение 1 (истина) для параметра `long_ticks`. В результате на осях будут отображены засечки.

```
37:     x_ticks => 0,
```

**В строке 37** устанавливается значение 0 для параметра `x_ticks`. Если этот параметр имеет значение 1 (как по умолчанию), на оси X будут изображены линии сетки.

```
38:     x_label_position => '.5',
```

**В строке 38** параметру `x_label_position` присваивается значение 0.5. Этот параметр устанавливает положение названия на оси X. Его значение должно быть в пределах от 0 до 1; при 0 название помещается на левом краю оси, а при 1 — на правом. Здесь мы установили значение .5, и название будет расположено посредине оси. По умолчанию для названия оси X принято положение 0.75.

```
39:     y_label_position => '.5',
```

**В строке 39** устанавливается такое же положение для названия оси Y.

```
40:     bgclr => 'white',
```

В **строке** 40 устанавливается значение параметра `bgclr` — цвет фона диафрагмы. В данном случае мы выбираем белый цвет (white).

```
41:     transparent => 0,
```

**В строке 41** параметру `transparent` присваивается значение 0. По умолчанию этот параметр имеет значение истины (т.е. ненулевое), и тогда цвет фона диаграммы считается прозрачным. Эта строка присутствует здесь только для демонстрации, так как мы уже определили цвет фона как белый.

```
42:     interlaced => 1,
```

**Строка 42** устанавливает для параметра `interlaced` значение истины. В результате GD::Graph генерирует чересстрочное изображение. Чересстрочный формат изображений может быть полезен при использовании графики в Web. При загрузке такого изображения на Web-страницу оно сразу развертывается на всем своем пространстве, постепенно приобретая все большую четкость и детализацию. Когда чересстрочное изображение создается, пикселы сохраняются не в том порядке, в котором они следуют в

[7] *Чтобы увидеть все доступные цвета, наберите* `perldoc GD::Graph::colour`. *Этот модуль устанавливается в составе пакета* `GD::Graph`.

изображении, и именно так передаются в броузер. Благодаря такой методике пользователь видит, как изображение постепенно "проявляется", а не выводится сверху вниз, как при обычном формате. Часто чересстрочные изображения дают больший размер файла, но зато они могут создать у пользователя иллюзию, будто загрузка происходит быстрее, чем на самом деле.

*43:* `x_labels_vertical => 1,`

**В строке 43** для параметра `x_labels_vertical` устанавливается значение 1. В этом случае метки на оси X будут отображены по вертикали, а не, как обычно, по горизонтали. Это может быть очень полезно, если метки включают длинные строки текста, например, URL, имена файлов или полные имена людей.

44: `lg_cols => 2,`

**Строка 44** управляет легендой, которую мы определили в строке 30. Значение 2 здесь указывает на две колонки, которые будут сформированы в легенде. Так как в нашем примере легенда состоит из двух элементов, они будут отображены рядом в двух колонках. Значение 1 отобразило бы элементы легенды один под другим.

45: `bar_spacing => 8,`

В **строке** 45 с помощью параметра `bar_spacing` определяется расстояние в пикселах между столбцами на графе. По умолчанию это значение — 0; этот параметр эффективен только для столбцовых диаграмм. Здесь мы устанавливаем расстояние в восемь пикселов, что даст нам довольно узкий маленький промежуток.

46: `shadow_depth => 4,`

**Строка 46** дает нам возможность с большей пользой применить этот промежуток из восьми пикселов, который мы только что установили. Параметр `shadow_depth`, здесь установленный в четыре пиксела, создаст на диаграмме тень от столбцов. Положительное значение создает тень, падающую направо и вниз. Отрицательное значение направляет тень налево и вверх.

47: `shadowclr => 'red',`

В **строке** 47 для только что созданной тени с помощью параметра `shadowclr` устанавливается красный цвет.

**48: );**

**Строка 48** завершает список параметров для метода `set()`. Эти 16 параметров определяют внешний вид диаграммы. Параметров может быть больше, чем необходимо для простой диаграммы, но не все из них программист может свободно использовать. Некоторые параметры связаны с конкретным типом создаваемой диаграммы, а другие глобальны. Мы настоятельно рекомендуем вам уделить немного времени документации по модулю, чтобы изучить все доступные параметры, поскольку через какое-то время их число может возрасти. Также рекомендуется поэкспериментировать с этим скриптом, изменяя, удаляя и добавляя параметры, чтобы изучить их влияние на диаграмму.

49: `my $format = $my_graph->export_format;`

**В строке 49** с помощью метода *export_format()* мы получаем формат этого изображения. Метод вызывает библиотеку GD и возвращает тип создаваемого изображения. Формат сохраняется в переменной `$format` и будет использован в строке 50.

50: `print header("image/$format");`

**В строке 50** мы выводим заголовок HTTP для показа изображения. Как можно заметить, в заголовке выводится значение переменной `$format`, созданной в строке 49. Это дает броузеру возможность узнать, какой файл поступает и как его надо обрабатывать.

```
51:  binmode STDOUT;
```

**В строке 51** для дескриптора вывода STDOUT устанавливается двоичный режим, так как данные изображения — двоичные.

```
52:  print $my_graph->plot(\@data)->$format();
```

**Строка 52** — последняя в скрипте. Именно здесь используются все параметры, которые мы установили. В этой строке в метод `plot()` передается ссылка на массив `@data`, содержащий все данные для построения диаграммы. Рассмотрим, что именно делает метод `plot()`. Он считывает данные координат из строк 24—28 и создает диаграмму на основе этих данных, указанного типа диаграммы и параметров. Готовая диаграмма выводится в файл, в данном случае в STDOUT, который направляет данные непосредственно в броузер, используя метод, соответствующий формату изображения. Если изображение — файл PNG, то будет применен метод *png()*. Нельзя вывести данные PNG, например, как файл **JPEG.** Это невозможно, так как их алгоритмы сжатия различны и несовместимы. Примерный результат выполнения этого скрипта показан на рис. 15.2.

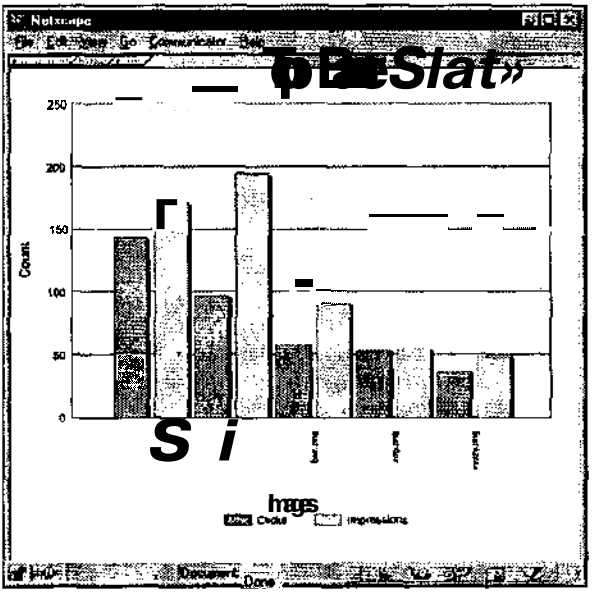



*Рис. 15.2. Вывод динамической диаграммы*

На этом рисунке мы видим, что скрипт `hits.cgi` сгенерировал очень неплохую столбцовую диаграмму. Все метки размещены там, где мы хотели, снизу по центру видна легенда с двумя колонками. Как показано в легенде, зеленые (более темные на рисунке) столбцы представляют щелчки, а желтые (более светлые) — показы. Мы считаем, что модуль GD: : Graph действительно может генерировать замечательные диаграммы, что позволяет дополнить Web-приложения такими интересными функциями, как графическое отображение в динамике статистики посещений, данных пользователей, коммерческой информации, сведений о трафике и любых других данных, которые только могут понадобиться в вашем приложении. Учтите, что с использованием этого модуля все диаграммы можно сохранить на диске и использовать позже. Например, планировщик событий cron может еженедельно создавать диаграммы на основе журналов Web-сервера и сохранять их. Здесь открываются прекрасные возможности для использования различных типов диаграмм.

# Создание эскизов изображений

Еще в главе 9 вы научились создавать эскизы изображений с помощью модуля `Apache::Album`. В этом разделе мы рассмотрим способ, которым это выполняется. Пример скрипта `make_thumbnails.cgi`, который мы разберем, использует основные принципы этого модуля и демонстрирует создание эскизов для изображений в каталоге и отображение их с помощью скрипта CGI. Также он показывает методику создания эскизов на тот случай, если вы захотите использовать эти функции в других скриптах Perl, не CGI.

Для работы этого скрипта должен быть установлен модуль `image: :Magick`, для которого в свою очередь требуется программный пакет Image Magick. Это свободно распространяемый пакет для создания изображений и их обработки, а модуль Perl `Image: :Magick` предоставляет интерфейс к его API. Таким образом, разработчик фактически может пользоваться возможностями этой программы, не вызывая ее напрямую. Существуют и другие модули Perl, предоставляющие интерфейс к другим программам обработки изображений, например, GIMP, которая позволяет разработчикам выполнять над изображениями такие же операции.

```
01:  #!/usr/bin/perl -wT
02:   use strict;
03:  use CGI qw(:standard);
04:   use Image::Magick;
```

**Строки 1–4** начинают скрипт, указывая путь к Perl, устанавливая предупреждения, проверку на загрязнение и строгий синтаксис и загружая модули `CGI` и `Image: : Magick`.

```
05:   print header,
06:   start_html('Эскизы')
07:  h1('Эскизы изображений');
```

Строки 5, 6 и 7 начинают код HTML создаваемой Web-страницы. В строке 5 методом *header ()* модуля `CGI.pm` отображается заголовок HTTP. Следующие строки начинают собственно HTML и выводят строку заголовка с текстом "Эскизы изображений". Все это уже должно быть вам прекрасно известно.

```
08:  unless (opendir(IN,"./")) {
09:     print "Нельзя открыть каталог ($!)";
10:     exit;
11: }
```

**Строки 8–11** — условное выражение, в котором мы выполняем два действия. Во-первых, мы пробуем открыть текущий рабочий каталог, и, во-вторых, мы выводим сообщение об ошибке и прекращаем выполнение скрипта, если открыть дескриптор не удается. Если же мы смогли открыть каталог, мы получаем дескриптор `IN`.

```
12:   my @files = grep { !/^tn_/ and /\.jpe?g$/ } readdir(IN);
13:   closedir(IN);
```

**Строка 12** создает массив подмножества файлов в каталоге. Это выполняется с помощью функции *grep()*. Функция *grep()* выполняет операцию, указанную в регулярном выражении, над каждым из элементов, возвращаемых функцией *readdir ()*, т.е. над всеми файлами в каталоге. Регулярное выражение выделяет файлы, которые имеют расширение `.jpeg` или `.jpg`, но не начинаются с префикса `tn_`. Далее в скрипте мы будем присваивать этот префикс (сокращение от "thumbnails") файлам эскизов. Благодаря этому мы сможем отличать изображения-оригиналы от эскизов. В результате выполнения этой строки массив `@files` будет заполнен именами всех изображений, которые заканчиваются на `.jpeg` или `.jpg`, но не начинаются с `tn_`. Затем **строка 13** закрывает дескриптор `IN`. Теперь мы знаем имена всех файлов, которые нас интересуют.

```
14:  foreach (@files) {
15:   unless <-e "tn_$_" && ( (stat (" ./tn_$_") ) [9] >
                                (stat($_))[9])) {
```

**Строка** 14 начинает цикл *foreach()* по всем именам файлов в массиве `@files`. В **строке** 15 начинается блок условия `unless` *(),* в котором мы выясняем, имеет ли уже изображение свой эскиз. Как было сказано выше, имена файлов эскизов имеют префикс `tn_`. В первую очередь мы проверяем существование файла оператором `-e`, а затем определяем время последнего изменения как файла изображения-оригинала, так и файла эскиза. Время последнего изменения содержится в девятом элементе массива, возвращаемого функцией *stat ().* Эта часть условного выражения будет истинной, если файл эскиза имеет более позднее время последнего изменения, чем файл оригинала. Это означает, что оригинал изображения не изменился с момента создания эскиза. Если это так, весь блок `unless` пропускается и эскиз сразу отображается в броузере (это происходит в строке 25), после чего проверяется следующее изображение. Если же версия эскиза оказывается более старой, чем последнее изменение оригинала, этот блок выполняется.

```
16:       my $image = new Image::Magick;
```

**Строка 16** создает новый объект `Image::Magick`. Этот объект создается для каждого файла, для которого требуется сгенерировать эскиз. Ссылка на объект сохраняется в переменной `$image`. Через этот объект мы будем вызывать методы для обработки изображения.

```
17:       print STDERR "Невозможно прочитать $_"
            if $image->Read($_);
```

**В строке 17** мы пробуем прочесть данные изображения с помощью метода `Image::Magick` *Read().* Если этот метод завершается неудачно, он возвращает значение истины. В этом случае мы не получаем необходимых данных, и поэтому в стандартный дескриптор ошибок (STDERR) ВЫВОДИТСЯ сообщение об ошибке. Эта методика позволяет фиксировать ошибку (скорее всего, в журнале ошибок Web-сервера), не прекращая выполнения скрипта. В метод *Read()* можно передавать список имен файлов, а также шаблон имени, например, `*.jpg`. Также этот метод может принимать открытый дескриптор файла, для чего надо вызвать его в форме `$image->Read(file => VTILEHANDLE)`. Формат файла изображения метод *Read()* определяет самостоятельно, считывая заголовок файла, который предшествует данным изображения. Это позволяет передавать в метод открытый дескриптор файла.

```
18:       my ($o_width, $o_height) = $image->Get('width',
                                                 'height');
```

**В строке 18** методом `Get` *()* определяются ширина и высота изображения-оригинала. Эти данные сохраняются в скалярах `$o_width` и `$o_height`. Префикс `o_` в именах переменных будет означать, что речь идет об оригинале, а префикс `tn_` обозначает эскиз.

```
19:       my $tn_width = $o_width * .25;
```

В **строке** 19 в переменную `$tn_width` записывается ширина в пикселах будущего эскиза. Мы принимаем, что эскизы будут вчетверо меньше оригиналов, поэтому ширина изображения-оригинала умножается на .25. Например, если ширина оригинала составляет 400 пикселов, то `$tn_width` будет равно 100.

```
20:       my $ratio = $o_width / $o_height;
```

В **строке** 20 отношение ширины изображения-оригинала к его высоте записывается в переменную `$ratio`. Считая, что размер оригинала был 400 на 200 пикселов, мы получаем `$ratio` равным 2.

```
21:     my $tn_height = $tn_width / $ratio;
```

В строке 21 вычисляется высота эскиза. Ширину эскиза мы делим на отношение ширины к высоте. В нашем примере высота эскиза будет составлять 100 / 2 = 50 пикселов.

```
22:     $image->Sample(width => $tn_width,
                       height => $tn_height);
```

В строке 22 методом *Sample ()* создается эскиз. Фактически этот метод не создает нового изображения, а лишь масштабирует данные оригинала до требуемого размера эскиза. Теперь наш объект изображения содержит данные эскиза. Ширина и высота эскиза передаются в метод через соответствующие переменные.

```
23:     $image->Write("tn_$_");
24: }
```

В строке 23 данные изображения с помощью метода *Write()* записываются на диск. Имя нового файла — имя оригинала, перед которым ставится префикс tn_. Так, если имя файла оригинала gouda.jpg, то файл эскиза получает имя tn_gouda.jpg.

Строка 24 завершает блок *unless ()*.

```
25:     print qq(<A HREF="$_"><IMG SRC=" ./tn_$_"
          ALT="$_"></A><BR>$_<P>);
26: }
```

В строке 25 HTML выводится в клиент. Этот HTML образует изображение-эскиз, связанное с гиперссылкой на изображение-оригинал. Затем после разрыва строки отображается имя оригинала.

Строка 26 завершает блок *foreach()*. Этот цикл выполняется для каждого файла изображения в каталоге, имеющего нужное расширение.

```
27: print qq(</BODY></HTML>);
```

Строка 27 завершает код HTML нашей страницы. На этот момент все требуемые эскизы уже созданы и вся галерея эскизов отображена в броузере.

В этом примере вы научились нескольким вещам. Во-первых, вы познакомились с модулем Image::Magick и узнали, как через него использовать возможности программы Image Magick. также вы научились читать содержимое каталога и отбрасывать ненужные файлы с помощью функции *grep()*.Мы рассмотрели создание физических[8] эскизов изображений, а также показали, как можно ускорить генерацию эскизов, выбирая только файлы, измененные после своих эскизов. Такое приложение может послужить основой, которую можно дополнить функциями, подобными тем, которые мы рассмотрели в модуле Apache::Album. Его также можно объединить с другими методами, приведенными в примерах этой книги, и получить в результате полный аналог Apache::Album, не требующий привлечения mod_perl.

# Применение к изображениям фильтров Image::Magick

В этом разделе мы продолжим рассматривать методы обработки изображений с помощью модуля Image::Magick. Здесь мы разберем новые методы, которые позволяют получить довольно забавный результат, а именно фильтры. Графический фильтр — это набор команд, который выполняется над изображением и производит на него некото-

[8] *Физических - в отличие от предыдущего примера, где данные изображения выводились непосредственно в броузер, без создания файла.*

рый эффект. Сами эти команды входят в состав пакета, в данном случае — Image Magick; разработчику нужно знать только название фильтра, например, "вихрь" (swirl), "рельеф" (emboss) и "отражение" (flip). Очень важно, чтобы разработчик прочел документацию и узнал, какие фильтры предлагает программа, а также попробовал их в работе, изучив действие каждого фильтра по отдельности и в сочетании с другими.

Пример скрипта `filter.cgi`, который мы сейчас рассмотрим, отображает Web-страницу со списком различных фильтров и их параметров. Это лишь часть доступных фильтров, но она позволяет получить представление об их возможностях.

```
01: #!/usr/bin/perl -wT
02: # filter.cgi
03: use strict;
04: use CGI qw(:standard);
05: use Image::Magick;
```

**Строки 1–5** образуют традиционное начало скрипта.

```
06: my %filters = (Charcoal => 'amount',
07:                Oil Paint => 'radius',
08:                Spread => 'amount',
09:                Solarize => 'factor',
10:                Swirl => 'degrees',
11:                Implode => 'factor',
12:                Flip => undef,
13:                Emboss => undef
14:               );
```

В **строках 6–14** формируется хэш `%filters`. Ключи этого хэша соответствуют фильтрам, которые можно будет вызвать на будущей Web-странице. Значения под этими ключами — параметры, которые надо будет передать в соответствующий фильтр Image Magick. Например, фильтр `oilPaint` имеет параметр `radius`, определяющий радиус кисти, которой обрабатывается изображение. Напротив, фильтр `Flip` не имеет параметров — он просто зеркально отражает изображение. Если фильтр не имеет никаких параметров, ему соответствует значение `undef`. Почему мы так сделали, станет ясно чуть позже. Этот хэш предназначен для создания Web-страницы и правильного вызова соответствующих фильтров.

```
15: my $filename = "kyla_smile.jpg";
16:  (my $filtered = $filename) =~ s!(\.\w*$)!
                                   ${filename}_filtered$1!i;
```

**Строка 15** определяет имя файла, который мы будем использовать. Так как этот пример лишь демонстрирует использование фильтров, имя файла жестко фиксируется в `$filename`. Позже, в качестве упражнения, вы можете дополнить этот скрипт любым из способов выбора имени файла, которые показаны в этой книге. Подробнее об этом мы расскажем в конце главы. В **строке 16** определяется переменная `$filtered`. Имя нового файла будет состоять из имени `$filename` без расширения, за которым будет следовать суффикс `_filtered`и, наконец, расширение. Хотя в строке 15 задано статичное значение переменной, этот механизм будет работать и в случае, если значение `$filename` определяется динамически.

```
17: if (!param('doit')) {
18:   print header,
19:   start_html ("Графические фильтры");
20:   print h2(param('error')) if param('error');
21:   print h2("Выберите фильтр"),
22:   start_form(-action=>"filter.cgi");
```

**Строки 17—22** начинают блок условия. Можно заметить, что в создаваемой нами форме имеется скрытый элемент под названием doit. Если этот параметр имеет значение, мы знаем, что форма была отправлена и нужно применить какой-то фильтр. В строках **18—22** с помощью методов модуля CGI.pm начинается код HTML и открывается Web-форма. **В строке 20,** если был передан параметр error, выводится дополнительный второй заголовок. Подробно мы объясним этот прием позже; пока достаточно знать, что этот параметр существует, если где-то в скрипте произошла ошибка, определенная программистом.

```
23:     for (sort keys %filters) {
24:       print qq($_ <INPUT TYPE="radio" NAME="filter"
            VALUE="$_"> );
25:       defined($filters{$_})
26:       ? print qq($filters{$_}
          <INPUT TYPE="text" NAME="$_" SIZE=3xP>)
27:       : print qq(<P>);
28:     }
```

В строке 23 начинается цикл, который обходит все ключи хэша %filters. Каждая итерация цикла начинается с вывода в **строке** 24 кода HTML, создающего переключатель со значением названия фильтра. **Строки 25—27** — одно условное выражение, разбитое на три строки для наглядности. В этом условии мы проверяем, записано ли под ключом хэша % filters определенное значение. Если это так, на странице отображается текстовое поле, в котором пользователь может ввести определенное значение параметра для этого фильтра. Например, когда обрабатывается элемент хэша для фильтра OilPaint, отображается переключатель с текстом "OilPaint" и текстовое поле длиной в три символа с надписью "radius" — названием параметра, который надо ввести. Если данный элемент хэша имеет неопределенное значение, как, например, элемент Flip, вместо текстового поля и надписи выводится просто дескриптор <p>.

```
29:     print submit(-value=>"Изменить изображение",
                     -name=>"doit"),
30:       end_form,
31:       end_html;
```

Строки 29, 30 **и** 31 завершают Web-форму и страницу HTML. Пока Web-форма не отправлена, это будет последним отображением в броузере. Результат показан на рис. 15.3.

```
32:   } else {
33:     my $filter = param('filter') ||
                     error("Выберите фильтр");
34:     my $amount = param($filter);
```

Строки 32, 33 **и** 34 начинают вторую часть условного выражения, которая выполняется, если скрипт должен не отображать Web-форму, а применить выбранный фильтр к изображению. **В строке 33** определяется тип фильтра, переданный из формы в параметре filter. Если такой параметр был передан, его значение записывается в переменную filter. Если нет, скрипт вызывает подпрограмму *error(),* которая будет показана позже. Обратите внимание, что в эту подпрограмму передается строка сообщения об ошибке. В **строке** 34 переменной $amount присваивается значение, переданное из формы в параметре, название которого соответствует названию фильтра. Выше, в строке 26, мы создавали для фильтра поле ввода, если этому фильтру, как, например, OilPaint, требовался параметр. Имя этого поля ввода совпадало с названием данного фильтра. Если фильтр не имеет параметра и поля ввода, как, например, фильтр Flip, то переменная $amount не будет иметь значения.

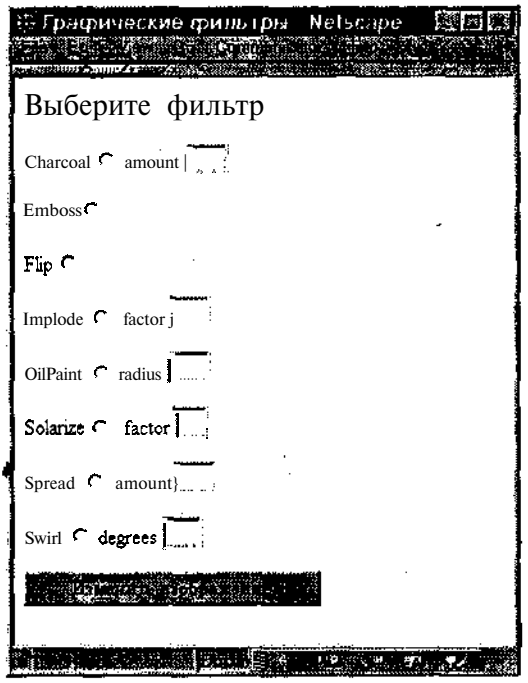



*Рис. 15.3. Форма для выбора графического фильтра*

```
35:     my $q = new Image::Magick;
```

В строке 35 устанавливается ссылка на новый объект image: `:Magick`. Через этот объект мы будем управлять файлом изображения.

```
36:     error("Невозможно прочитать файл изображения")
          if  $q->Read($filename);
```

Строка 36, если метод *Image::Magick:Read()* не может прочесть данные из $filename, вызывает подпрограмму *error ()* и передает в нее строку сообщения об ошибке.

```
37:     defined($filters{$filter})
38:       ? $q->$filter($filters{$filter} -> $amount)
39:       : $q->$filter();
```

Строки 37, 38 и 39 — на самом деле одна строка программы, разбитая на три строки для наглядности. Здесь мы вызываем фильтр в одной из двух форм в зависимости от того, принимает ли данный фильтр параметр. В строке 37 определяется, имеет ли элемент хэша `%fliters` для выбранного фильтра определенное значение. Вспомните, что фильтры, не имеющие параметра, например, Flip, имеют в хэше неопределенное значение `undef`. Если значение в хэше определено, строка 38 вызывает метод с соответствующим параметром. На рис. 15.3 показан результат применения выбранного фильтра Swirl со значением параметра 75. После отправки формы строка 38, если подставить в нее значения, приобрела бы следующий вид.

```
$q->Swirl(degrees => 75)
```

Если значение в хэше не определено, в строке 39 вызывается метод без параметра. В этот момент модуль Image: `:Magick` вызывает метод программы Image Magick, который и выполняет обработку изображения, после чего новые данные изображения сохраняются в объекте $q.

```
40:    $q->Write($filtered) ;
```

**В строке 40** новые данные изображения записываются в файл `$filtered`. После этого мы будем иметь новое изображение не только в памяти, но и в файле.

```
41:     print redirect($filtered);
42:  }
```

**Строка 41** направляет броузер на новое изображение. Это производится с помощью метода *redirect ()* из модуля `CGI.pm`. Итак, мы прочли изображение, обработали его, создали файл с новыми данными и передали его пользователю. **Строка 42** завершает блок if.

```
43:  sub error {
44:    my Serror = CGI::escape(@_) ;
45:     print "Расположение: filter.cgi?error=$error\n\n";
46:     exit;
47:  }
```

**Строки 43—47** составляют подпрограмму *error()*. Эта подпрограмма принимает как параметр строку сообщения об ошибке. В **строке 44** это сообщение преобразуется в кодировку URI с помощью метода *CGI::escape ()*. После перекодировки **строка 45** возвращает броузер в данный скрипт с параметром `error` и сообщением об ошибке в кодировке URI в качестве значения. Когда скрипт обнаруживает этот параметр в строке 20, он отображает сообщение в броузере. **Строка 46** обеспечивает выход из скрипта. Перенаправление броузера на другой адрес на самом деле не приводит к прекращению работы скрипта, и поэтому его надо явно закончить, так как после ошибки скрипт уже не должен работать, Наконец, **строка 47** завершает скрипт. На рис. 15.4 показано действие фильтра Swirl с параметром 75.

*Рис. 15.4. Эффект фильтра Swirl*

# Анимированные изображения

В предыдуших разделах мы рассмотрели методы доступа к изображениям и изменения их с помощью `Image::Magick`. В этом разделе мы сделаем еще один шаг вперед и создадим анимированное изображение. Такое изображение представляет собой

файл, содержащий несколько изображений, называемых также "кадрами" (scenes). Недавно появился новый графический формат — Multiple-Image Network Graphics, или MNG[9], основанный на формате PNG, но позволяющий включать в один файл много изображений и имеющий расширенные возможности, которые PNG не поддерживает. Программа Image Magick может работать с изображениями MNG, но они не поддерживаются распространенными броузерами и поэтому неприменимы для Web-приложений. Здесь можно использовать другой формат — широко применяемый GIF89a. Следующий пример, скрипт animate.cgi, покажет нам, как создать динамическое изображение. Этот пример будет повторно читать одно и то же изображение и применять к нему фильтр Swirl с постепенно увеличивающимся параметром degrees (углом закручивания). Значительная часть этого скрипта уже знакома вам.

```
01:  #!/usr/bin/perl -wT
02:  use strict;
03:  use CGI qw(:standard) ;
04:  use Image::Magick;
```

**Строки 1—4** не нуждаются в пояснении.

```
05:  my $image = new Image: :Magick;
```

**Строка 5** создает ссылку на новый объект Image: :Magick.

```
06:  my $i =10;
```

**В строке 6** объявляется и инициализируется переменная $i, которую мы будем использовать в следующем блоке for.

```
07:  for (1..20) {
08:    $image->Read("kyla_smile.jpg");
09:    $image->Swirl(degrees => $i);
10:    $i += 10;
11:  }
```

**Строки 7—11** — блок **for,** в котором изображение читается и к нему применяется фильтр. **Строка 8** считывает данные изображения, а **строка 9** обрабатывает их фильтром со значением параметра degrees, равным текущему значению $i. На каждой итерации цикла в объект $image добавляется новое изображение. **В строке 10** происходит приращение параметра фильтра на 10 на каждой итерации цикла. После завершения цикла объект Image: : Magick будет содержать данные для 20 изображений.

```
12:  $image->Set(loop=>0) ;
13:  $image->Write("kyla_ani.gif");
14:  print redirect("kyla_ani.gif");
```

В **строке** 12 для изображения устанавливается атрибут loop. По умолчанию этот атрибут имеет значение 1, и тогда все кадры изображения будут показаны по одному разу, а на последнем кадре анимация закончится. Если же этот параметр имеет значение 0, кадры будут отображаться в бесконечном цикле. **Строки 13 и 14** записывают файл на диск и направляют броузер на новое изображение, В результате мы получим непрерывно изменяющееся изображение; к сожалению, его невозможно показать на бумаге.

Как вы, вероятно, заметили, многокадровые изображения (хотя и входящие в один физический файл, как в этом примере) можно читать и не создавая каждый раз новый объект Image::Magick. Мы уже говорили, что метод *Read()* может читать несколько файлов сразу, что очень полезно при создании анимированных изображений. Например, если у нас есть несколько файлов или кадров, которые образуют последовательность, их можно прочесть в одном вызове метода, создав таким образом многокадровую анимацию. Если запустить предыдущий пример, можно заметить, что кадры отображаются без перерыва между ними.

[9] *Произносится "минг". См. http://www.libpng.org/pub/mng/.*

Но предположим, что у нас есть последовательность изображений, из которых надо создать анимацию с перерывом между кадрами. Если заменить строки 6–11 в скрипте animate.cgi следующим фрагментом кода, мы получим чтение нескольких файлов за один раз и установку интервала между кадрами с помощью нового параметра delay.

```
$image->Read(qw(image_a.gif image_b.gif image_c.gif
                image_d.gif));
$image->Set(delay => 200, loop => 0);
```

В этой главе вы научились создавать эскизы, применять к изображениям фильтры, создавать анимированные изображения, строить диаграммы и дополнять изображения текстом и фигурами. Это основные приемы, которые разработчик может применять при создании и обработке изображений. Если необходимы более совершенные методы, лучше всего, конечно, обратиться к документации по модулям.

# Упражнения

- Измените скрипт make_thumbnails.cgi так, чтобы он мог обходить дерево каталогов и создавать эскизы всех найденных изображений.
- Постройте в скрипте hits.cgi диаграмму другого типа. Например, круговая диаграмма может показать распределение щелчков или показов между баннерами.
- Интересное приложение можно создать на основе скрипта filter.cgi. Эта программа должна принимать файл, загруженный в броузер, преобразовывать его фильтром и возвращать новое изображение. В главе 11 было показано, как возвратить в броузер двоичные данные, чтобы пользователь мог сохранить их на диске как файл.

# Листинги

**Листинг 15.2. Скрипт add_text.cgi**

```
01: #!/usr/bin/perl -wT
02: use strict;
03: use CD;
04: no strict 'subs';
05: open (PNG, "kyla_smile.png") || die "$!";
06: my $image = newFromPng CD::Image(PNG) || die "$!";
07: close PNG;
08: my $white = $image->colorAllocate(255, 255,255);
09: my $black = $image->colorAllocate(0,0,0);
10: $image->arc(50,50,95,75,0,36,0,$black);
11: $image->fillToBorder(50,50,$black,$white);
12: $image->arc(60,95,25,25,0,360,$black);
13: $image->fillToBorder(60,95,$black,$white);
14: $image->arc(70,125,20,20,0,360,$black);
15: $image->fillToBorder(70,125,$black,$white);
16: $image->string(gdMediumBoldFont,10,45,
      "Привет, папа!",$black);
17: print "Content-type: image/png\n\n";
18: binmode STDOUT;
19: print $image->png;
```

Листинг 15.3. Скрипт **hits**.cgi

```
01: tt!/usr/bin/perl -wT
02:  use strict;
03:  use CGI qw(:standard) ;
04:  use GD::Graph::bars;
05:  use DBI;
06:  my $dbname = 'book';
07:  my Sdbhost - 'localhost';
08:  my @res;
09:  my $dsn = "DBI:mysql:database=$dbname;host=$dbhost";
10:  my $dbh=DBI->connect($dsn,"guest","cheese");
11:  if (!defined($dbh)) {
12:    print header;
13:    print "\пошибка:
         Не удается подключиться к базе данных MySQL:\n";
14:    print DBI->errmsg;
15:    print "-" x 25;
16:    exit;
17:  }
18:  my $sth = $dbh->prepare(qq{select IMAGE, CLICKS,
       IMPRESSIONS from image_tracker order by CLICKS
       desc limit 5}};
19:  $sth->execute;
20:  while (my ©results = $sth->fetchrow_array) 1
21:    push ©res, @results;
22:  }
23:  $dbh->disconnect;
24:  my @data - (
25:   [$res[0],$res[3],$res[6],$res[9],$res[12]],
26:   [$res[1],$res[4],$res[7],$res[10],$res[13]],
27:   [$res[2],$res[5],$res[8],$res[11],$res[14]],
28:  );
29:  my $my_graph = new CD::Graph::bars(640, 480);
30:  $my_graph->set_legend("Щелчки", "Показы");
31:  $my_graph->set(
32:    dclrs => [ qw(lgreen lyellow) ],
33:    title => "Статистика самых популярных баннеров",
34:    x_label => "Баннеры",
35:    y_label => "Количество",
36:    long_ticks => 1,
37:    x_ticks => 0,
38:    x_label_position => '.5',
39:    y_label_position => '.5',
40:    bgclr => 'white',
41:    transparent => 0,
42:    interlaced => 1,
43:    x_labels_vertical => 1,
44:    lg_cols => 2,
45:    bar_spacing => 8,
46:    shadow_depth => 4,
47:    shadowclr => 'red',
48:  );
49:  my $format = $my_graph->export_format;
50:  print header("image/$format");
51:  binmode STDOUT;
52:  print $my_graph->plot(\@data)->$format();
```

**ЛИСТИНГ 15.4. Скрипт make_thumbnails.cgi**

```
01: #!/usr/bin/perl -wT
02: use strict;
03: use CGI qw(:standard);
04: use Image::Magick;
05: print header,
06: start_html('Эскизы')
07: h1('Эскизы изображений');
08: unless (opendir(IN,"./")) {
09:   print "Нельзя открыть каталог ($!)";
10:   exit;
11: }
12: my @files = grep { !/^tn_/ and /\.jpe?g$/ } readdir(IN);
13: closedir(IN);
14: foreach (@files) {
15:  unless (-e "tn_$_" && ( (stat ("./tn_$_")} [9] >
                                (stat($_))[9])) {
16:     my $image = new Image::Magick;
17:     print STDERR "Невозможно прочитать. $_"
          if $image->Read($_);
18:     my ($o_width, $o_height) = $image->Get('width',
                                                'height');
19:     my $tn_width = $o_width * .25;
20:     my $ratio = $o_width / $o_height;
21:     my $tn_height = $tn_width / $ratio;
22:     $image->Sample(width => $tn_width,
                       height => $tn_height};
23:      $image->Write("tn_$_");
24:   }
25:   print qq(<A HREF="$_"><IMG SRC=" ./tn_$_"
        ALT="$_"X/A><BR>$_<P>);
26: }
27: print qq(</BODY></HTML>);
```

**Листинг 15.5. Скрипт filter.cgi**

```
01: #!/usr/bin/perl -wT
02: # filter.cgi
03: use strict;
04: use CGI qw(:standard);
05: use Image::Magick;
06: my %filters = (Charcoal => 'amount',
07:                Oil Paint => 'radius',
08:                Spread => 'amount',
09:                Solarize => 'factor',
10:                Swirl => 'degrees',
11:                Implode => 'factor',
12:                Flip => undef,
13:                Emboss => undef
14:               );
15: my $filename = "kyla_smile.jpg";
16:  (my $filtered = $filename) =~ s!(\.\w*$)!
                                   ${filename}_filtered$1!i;
17: if (!param('doit')) {
18:   print header,
19:   start_html ("Графические фильтры");
```

```
20:    print h2(param('error')) if param('error');
21:    print h2("Выберите фильтр"),
22:    start_form(-action=>"filter.cgi");
23:    for (sort keys %filters) {
24:      print qq(S_ <INPUT TYPE="radio" NAME="filter"
           VALUE="$_"> );
25:      defined($filters{$_})
26:      ? print qq($filters{$_}
         <INPUT TYPE="text" NAME="$_" SIZE=3xP>)
27:       : print qq(<P>);
28:    }
29:    print submit(-value=>"Изменить изображение",
                    -name=>"doit"),
30:     end_form, •
31:      end_html;
32: } else {
33:    my Sfilter = param('filter') ||
                    error("Выберите фильтр");
34:    my $amount = param($filter);
35:    my $q = new Image::Magick;
3 6: error("Невозможно прочитать файл изображения")
       if $q->Read($filename);
37:    defined($filters{$filter}).
38:      ? $q->$filter($filters{$filter} => $amount)
39:      : $q->$filter();
40:    $q->Write($filtered);
41:    print redirect($filtered);
42: }
43: sub error {
44:    my $error = CGI::escape(@_);
45:    print "Расположение: filter.cgi?error=$error\n\n";
46:    exit;
47: }
```

**ЛИСТИНГ 15.6. Скрипт animate.cgi**

```
01: #!/usr/bin/perl -wT
02: use strict;
03: use CGI qw(:standard);
04: use Image::Magick;
05: my $image = new Image::Magick;
06: my Si =10;
07: for (1..20) {
08:    $image->Read("kyla_smile.jpg");
09:   $image->Swirl(degrees => $i);
10:   Si += 10;
11: }
12: $image->Set(loop=>0);
13: $image->Write("kyla_ani.gif");
14: , print redirect("kyla_ani.gif");
```

Глава 16

# RSS и XML

## XML и RSS — краткий обзор

Язык XML (Extensible Markup Language) — это язык разметки для структурированных документов. Структурированным называется документ, содержащий различные элементы, такие как изображения и текст, а также указание на то, какой это элемент. Например, письмо содержит такие элементы, как адрес, тело и подпись, которые играют в документе различные роли. Если в письме имеется какое-то указание на то, что делает каждая часть каждого из этих элементов, документ является структурированным. Структуру имеет большинство документов, а XML позволяет определить эту структуру удобным и стандартным способом. Документы HTML — структурные, так как они содержат элементы и дескрипторы, определяющие роль того элемента.

Однако HTML и XML — не одно и то же. Язык HTML состоит из известного набора дескрипторов, а в XML такого ограничения нет. Язык XML — не только способ разметки документов; это, по сути, метаязык, который дает разработчикам возможность самим описать такую разметку. Разработчики документов и приложений XML, использующих эту разметку, сами определяют дескрипторы для документа и устанавливают отношения между ними. Назначение XML — позволить разработчикам использовать собственные дескрипторы при создании структурированных документов для Web. В этой главе мы рассмотрим некоторые основы документов XML и расскажем, как использовать RSS, вариант языка XML, для Web.

## Структура документа XML

По внешнему виду документ XML очень близок к документу HTML, и поэтому его легко проанализировать и понять. В листинге 16.1 показан простой документ XML.

**Листинг 16.1. Пример документа XML**

```
<?xml version="1.0"?>
<Zappa>
<quote>Good night Cleveland, wherever you are!</quote>
<quQte>Shoot low, they're riding Shetlands.</quote>
</Zappa>
```

Здесь не обнаруживается ничего необычного. Первая строка объявляет, что это документ XML, а также версию применяемого XML. Эта строка не обязательна, но ее рекомендуется включать, чтобы документ был построен по всем правилам. Затем создается контейнер с названием "Zappa". В этом контейнере содержатся два элемента, выделенные дескрипторами "quote". Наконец, контейнер "Zappa" закрывается. Этот пример примитивен; на самом деле XML работает гораздо сложнее, чем это кажется на первый взгляд, однако для того, чтобы рассмотреть все его аспекты, потребуется целая книга (и есть книги, которые только этому и посвящены). Тем не менее, данный пример дает достаточно информации, чтобы разобраться в последующих примерах и быстро начать работу с XML и его вариантами.

## Портал новостей на базе RSS

Всего несколько лет назад в Netscape было создано нечто, что можно было бы назвать первым Web-порталом. Они разработали технологию My News Network[1], или MNN, которая давала каждому пользователю Netscape возможность располагать многими функциями получения новостей и поиска в сети прямо на собственной начальной странице. Пользователи могли выбирать сайты, с которых они хотят получать сводки новостей, и отображать их. Этим сводкам было дано наименование "каналы", которое сейчас стало общепринятым. Для работы каналов Netscape специальный сервер должен был периодически извлекать структурированные документы XML с сайтов поддержки и обновлять содержимое каналов. Чтобы файлы на всех сайтах поддержки имели одинаковую структуру, Netscape разработала формат RSS (RDF Site Summary). Этот формат основан на XML и RDF[2] (Resource Description Format — формат описания ресурсов) и определяет язык разметки, который должны использовать разработчики. Однако область применения RSS не ограничивается только каналами Netscape. Благодаря модулю XML::RSS программист может форматировать в таких файлах каналов любые данные для Web.

Прежде чем перейти к использованию XML: :RSS, рассмотрим сначала язык разметки формата RSS. В RSS можно применять ограниченный набор тэгов. Основным контейнером в документе является канал (channel). В этом контейнере содержится несколько элементов, при помощи которых можно определить содержимое. Три основных элемента— заголовок.(title), ссылка (link) и описание (description). Канал может также содержать изображение (image), блок ввода текста (textinput) и другие контейнеры. Попробуем разобрать примерный документ RSS.

```
<?xml version="1.0" encoding="windows-1251"?>
<!DOCTYPE rss PUBLIC
  "-//Netscape Comraunications//DTD RSS 0.91//EN"
  "http://my.netscape.com/publish/formats/rss-0.91.dtd">
<rss version="0.91">
```

Этот раздел включает три объявления. В первом объявляется, что это документ XML, как мы видели в предыдущем примере. Второе объявление — атрибут DOCTYPE документа.

---

[1] *http://my.netscape.com*

[2] *Это формат, используемый для описания метаданных в Web.*

В третьем объявляется, что это документ RSS версии 0.91. Версия RSS 0.90 была представлена Netscape в 1999 г., так что эта версия еще остается довольно новой. Теперь, когда мы определили тип создаваемого документа, можно приступать непосредственно к созданию.

```
<channel>
```

Эта строка открывает контейнер-канал. Все содержимое вплоть до закрытия этого контейнера будет его частью. Канал — главный контейнер во всех документах RSS.

```
<title>Мои новости</title>
<link>http://news.me.com</link>
<description>Мои новости для вас!</description>
<language>ru</language>
<copyright>Copyright 2000++, Me</copyright>
<pubDate>Суббота, 14:40:45 9 июня 2001 г.</pubDate>
<lastBuildDate>Суббота, 14:40:45 9 июня 2001 г.</lastBuildDate>
<managingEditor>me@me.com</managingEditor>
<webMaster>me@me.com</webMaster>
```

Здесь определяется список элементов. Три главных и необходимых элемента: title — заголовок канала, link — местоположение Web-сайта для этого канала и description — описание канала. Остальные перечисленные элементы необязательны для включения в документ, но могут давать полезную информацию. Мы не будем приводить отдельное описание каждого элемента, так как их названия достаточно хорошо объясняют их назначение.

```
<image>
<title>Мои новости</title>
<url>http://news.me.com/my_news.gif</url>
<link>http://news.me.com</link>
<width>119</width>
<height>30</height>
</image>
```

В этом фрагменте кода показан необязательный контейнер image. Этот контейнер содержит информацию об изображении для эмблемы канала. И в этот раз, благодаря простоте формата RSS, название каждого элемента очевидно объясняет назначение его данных. Последняя строка фрагмента закрывает контейнер.

```
<item>
<title>Человек ест сыр - MPEG, 11.00</title>
<link>http://news.me.com/news/story2.html</link>
</item>
```

Контейнер item чрезвычайно важен, так как он определяет содержимое канала. Этот контейнер необходим; канал может содержать несколько контейнеров item. Если, например, у нас есть пять сообщений новостей, должны быть соответственно пять контейнеров item. Этот контейнер содержит два элемента, причем оба обязательны. Первый элемент определяет заголовок сообщения, а второй — ссылку на него.

```
<textinput>
<title>Поиск в новостях</title>
<description>Поиск в архивах</description>
<name>text</name>
<link>http://news.me.com/search.cgi</link>
</textinput>
```

Контейнер textinput создает поле ввода текста, которое можно использовать для поиска на сайте. Этот контейнер необязателен, так как некоторые сайты могут не иметь возможности поиска. Если же сайт имеет функцию поиска, этот контейнер — превосходный способ предоставить ее пользователям, которые смогут выполнять поиск на вашем сайте с любой страницы, где отображается этот канал.

```
</channel>
</rss>
```

Документ завершается закрытием контейнера-канала и дескриптора RSS. Эта операция эквивалентна закрытию документа HTML с помощью дескриптора </HTML>. Теперь наш документ RSS готов. На основе этого документа можно создать канал Netscape[3], который сможет поместить на свою страницу или в Web-приложение любой разработчик. Теперь, когда мы знаем структуру документа RSS, мы можем рассмотреть способы использования этого файла, а затем самостоятельно создать такой документ с помощью Perl.

## Портал новостей для начальной страницы

Имея общее представление о документах RSS и их структуре, можно перейти к тому, как с помощью этих файлов создать динамическое содержимое страницы и сформировать на его основе специализированный портал. Самый простой способ использования файлов формата RSS в Perl предоставляет модуль XML::RSS, написанный Джонатаном Айзенцопфом (Eisenzopf). Этот модуль позволяет программисту без труда получать данные, содержащиеся в файле формата RSS, используя объектно-ориентированный подход. В этом разделе мы разберем приложение, которое выбирает файлы RSS, отображает каналы на Web-странице и дает пользователю возможность добавлять новые каналы и выбирать, какие из имеющихся каналов должны быть отображены на Web-странице.

В первую очередь следует установить, где мы будем получать файлы каналов. На Web-сайте xmlTree[4] представлен каталог значительной части XML-содержимого в Web, и значительную часть этого содержимого составляют файлы RSS. На этом сайте можно провести поиск по типу канала, который вас интересует, и найти местоположение файлов RSS[5] для этих каналов. Теперь, когда мы знаем, где найти файлы каналов, можно приступать к скрипту.

Вначале мы создадим таблицу, которая будет содержать информацию о каналах. Для каждого канала будем хранить три элемента данных: URL файла RSS, имя канала и должен ли канал быть отображен на Web-странице.

**Листинг 16.2. Команда SQL для создания таблицы RDF**

```
CREATE TABLE rdf (
  URL varchar(250) NOT NULL,
  Name varchar(250) NOT NULL,
  Selected int(11)
);
```

Чтобы облегчить анализ последующего скрипта, ниже мы поместили команды SQL, которые вставляют в таблицу записи для четырех каналов. Также это поможет дать представление о данных таблицы.

```
INSERT INTO rdf VALUES
  ('http://slashdot.org/slashdot.rdf','Slashdot',0);
INSERT INTO rdf VALUES
  ('http://www.news.perl.org/perl-news-short.rdf',
   'Perl News',1);
INSERT INTO rdf VALUES
  ('http://freshmeat.net/backend/fm.rdf, 'Freshmeat',1);
INSERT INTO rdf VALUES
  ('http://www.securityfocus.com/topnews-rss.html',
   'Security Focus',1);
```

[3] *Файл RDF следует зарегистрировать в Netscape, чтобы MNN знала, где его найти.*

[4] *www.xmtree.com*

[5] *Многие из этих файлов имеют расширение .rdf, хотя это не обязательно.*

Затем нам нужен способ получения файлов RSS на локальной машине, чтобы модуль XML::RSS мог проанализировать их. Для этой цели мы создадим скрипт fetch, который можно будет запускать из командной строки или из планировщика событий через регулярные интервалы.

```
01: #!/usr/bin/perl -w
02: # fetch
03: use strict;
04: use File::Basename;
05: use DBI;
06: use LWP::Simple qw(mirror);
```

**Строки 1-6** определяют путь к Perl и загружают необходимые модули. Метод *mirror ()* из модуля LWP::Simple послужит нам для получения удаленных файлов RSS. На локальной машине каждый файл будет сохранен с первоначальным именем. Для определения этого имени мы применим метод *basename ()* из File::Basename.

```
07: my $RDF_DIR = './rdf';
```

**Строка 7** инициализирует переменную $RDF_DIR. Ее значение — каталог, в который мы будем записывать найденные файлы RSS.

```
08: my $dbh = DBI->connect("dbi:mysql:book","user","password");
09: my $sth = $dbh->prepare(qq{select URL from rdf});
10: $sth->execute or die $DBI::errstr;
```

**Строка 8** выполняет подключение к базе данных. В **строке** 9 подготавливается запрос, который выбирает из таблицы все URL для файлов RSS. В **строке** 10 этот запрос выполняется или выводится сообщение об ошибке из DBI.pm.

```
11: while (my $url = $sth->fetchrow) {
12:   my $name = basename($url);
13:   mirror($url, "./$RDF_DIR/$name");
14: }
```

В **строках 11–14** выполняется цикл по последовательности результатов, полученных из базы данных. Значение, которое возвращается и сохраняется в $url — отдельный URL. **В строке 12** $url передается в метод *basename ()*, который извлекает из этого URL имя файла. Это значение сохраняется в $name. Главная работа производится в **строке 13.** Метод *mirror()* принимает URL в своем первом параметре и обращается к указанной в нем удаленной Web-странице. Второй параметр — расположение, в которое эта страница копируется. В результате выполнения этого цикла все доступные файлы RSS будут сохранены на локальной машине. Максимальную пользу этот скрипт приносит, если запускается через регулярные промежутки времени и загружает наиболее свежие версии файлов RSS.

```
15: $dbh->disconnect;
```

В **строке** 15 производится отключение от базы данных.

Задача следующей части нашего приложения — извлечь пользу из загруженных файлов RSS. Скрипт index.cgi, который рассматривается ниже, создает на Web-странице блоки каналов, в которых отображаются данные из файлов RSS.

```
01: #!/usr/bin/perl -wT
02: # index.cgi
03: use strict;
04: use CGI qw(:standard end_ul end_table);
05: use CGI::Carp qw(fatalsToBrowser);
06: use File::Basename;
07: use DBI;
08: use XML::RSS;
```

```
09: my $RDF_DIR = './rdf';
10; my $dbh = DBI->connect ("dbi:mysql:book","user", "password")
      or print $DBI::errstr;
11:  my $sth = $dbh->prepare(qq{select URL from rdf
      where Selected = 1});
12:  $sth->execute;
```

**В строках 1-12** появляется только один новый элемент — модуль XML: :RSS. С помощью этого модуля мы получаем из файлов **RSS** в каталоге $RDF_DIR нужную информацию. Также мы извлекаем из базы данных все **URL**, для которых поле Selected имеет значение 1, что указывает на то, что данный канал должен быть отображен на Web-странице.

```
13: print header,
14:    start_html("Моя начальная страница")*
15:    п2("Мои любимые сайты");
16: print start_table ({cellpadding=>0, cellspacing=>0,
      border=>0,•width=>'100%'}),
17:    td;
```

**Строки 13-17** начинают код HTML страницы, При выводе HTML интенсивно применяются методы модуля **CGI.pm.**

```
18:  ту $count = 1;
19:  my @html = ('</TD><TD>', '</TD><TR><TD>');
```

**В строках 18 и 19** инициализируются две переменные, которые будут служить нам как своего рода переключатель. Каналы будут отображены в двух колонках, и в каждом случае мы должны знать, надо ли для начала новой строки таблицы выводить дескриптор <TR>. Так как этот дескриптор должна иметь каждая вторая колонка, с помощью переменной 5count можно легко определять, какой из двух кодов в массиве **@html** надо использовать.

```
20:  while (my $url = basename($sth->fetch rów)) {
21:    my $rss = new XML::RSS;
```

В **строке** 20 начинается цикл, который обходит все **URL**, полученные из базы данных. Каждая строка результатов обрабатывается методом *basename ()*, который выделяет из **URL** имя файла. Например, **URL** `http://slasdhot.org/slashdot.rdf` будет сокращен до `slashdot.rdf`. Полученное имя файла записывается в $url. **В строке 21** создается новый объект XML: :RSS. Ссылка на него помещается в переменную $rss.

```
22:     eval {$rss->parsefile("$RDF_DIR/$url")};
23:     warn "Не удается проанализировать $url - $@"
          and next if $@;
```

**В строке 22** выполняется метод *parsefile ()*. Этот метод открывает указанный локальный файл RSS и анализирует его. Мы вызываем **метод** через функцию *eval ()*, так как, если RSS поврежден, может возникнуть исключение. Благодаря *eval ()* это исключение перехватывается и выполнение скрипта продолжается. Если обнаружено исключение, **строка 23** вьшодит предупреждение к STDERR И переходит к следующей итерации цикла.

```
24:    my $last_mod = scalar
         localtime((stat("$RDF_DIR/$url"))[9]);
```

**В строке 24** переменная $last_mod инициализируется скалярным значением времени последнего изменения файла RSS.

```
25:     print start_table({cellpadding=>0,
          cellspacing=>2, border=>5, width=>'75%'}),
26:       td({valign=>'CENTER', bgcolor=>'#C0C0C0'});
```

**Строки 25 и 26** начинают таблицу HTML, в которой будет отображен канал.

```
27:    $rss->{image}{url}
28:     ? print img({src=>$rss->{image}{url}})
29:     : print strong($rss->{channel}{title});
```

**Строки 27, 28 и 29** — одна строка, разделенная натрое для наглядности. В строке 27 мы проверяем значение `$rss->{image}{url}`. Если оно существует, это означает, что файл RSS содержит контейнер image, и в **строке** 28 содержащееся в нем изображение выводится в броузер. Если же такого контейнера нет, отображается заголовок канала.

```
30:     print ul;
```

**В строке 30** выводится дескриптор <UL>. Сообщения файла RSS будут отображены как ненумерованный список.

```
31:    for (@{$rss->{items}}) {
32:      print li(a({href=>$_->{link}}, $_->{title}));
33:    }
```

**В строках 31—33** выполняется цикл по всем пунктам (сообщениям) в контейнере-канале. Выражение $rss->{items} — это ссылка на массив, и, как таковая, она разыменовывается. **В строке 32** пункт выводится в броузер. Каждый пункт отображается как гиперссылка на URL полного сообщения. Текст этой гиперссылки — заголовок сообщения. Значения URL и заголовка взяты из элементов link и title контейнера item.

```
34:    print end_ul;
```

**Строка 34** выводит дескриптор </UL>.

```
35:    if ($rss->{textinput}{link}) {
36:       print $rss->{textinput}{description},
           start_form(-method => 'GET',
37:                     -action => $rss->{textinput}{link}),
38:        textfield(-name => $rss->{textinput}{name}),
39:        end_form;
40:    }
```

**В строках 35—40** обрабатывается контейнер textinput, если он обнаружен в документе. Если в **строке 35** оказывается, что элемент link контейнера textinput существует (имеет значение истины), в остальной части блока выводится соответствующая форма.

```
41:    print qq(Последнее обновление: $last_mod<BR>) ,
42:      end_table;
43:    $html[$count^=1];
44: }
```

**В строке 41** в броузере отображается дата последнего изменения, полученная в строке 24. **Строка 42** закрывает таблицу канала, а **строка** 43 делает одну очень интересную вещь. Помните, как в строках 18 и 19 мы инициализировали переменные @html и $count и дали им роль переключателя? Этот переключатель срабатывает в строке 43. Здесь мы в зависимости от результата операции XOR над значением $count и единицей выводим код HTML, содержащий либо не содержащий дескриптор новой строки таблицы <TR>. Чем больше начальное значение, которое мы присвоим $count, тем больше колонок мы получим. Наконец, **строка** 44 завершает цикл while ().

```
45: print end_table,
46:   end_html;
```

**Строки 45 и 46** завершают скрипт, выводя закрывающие дескрипторы таблицы и HTML. В результате выполнения этого скрипта мы получаем сгенерированную Web-страницу, подобную показанной на рис. 16.1.

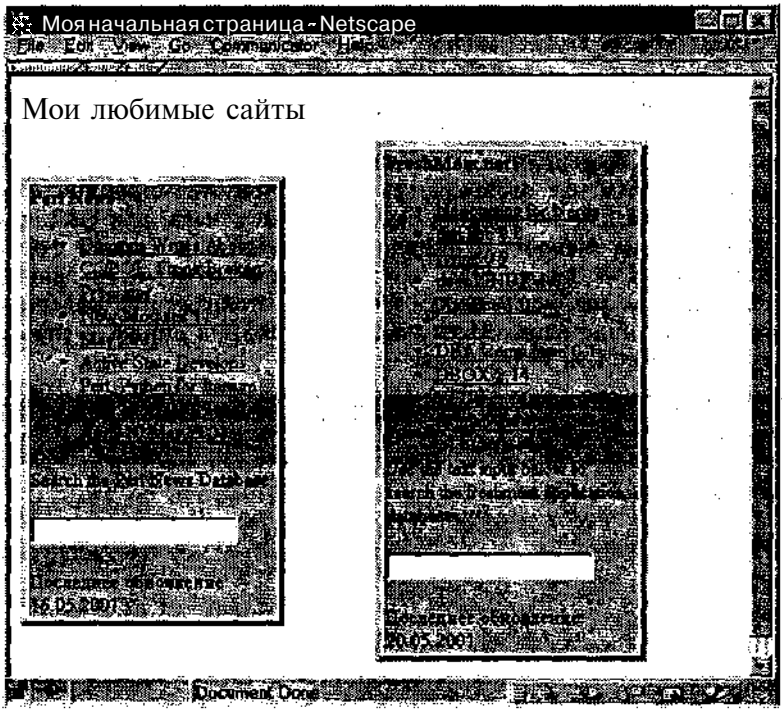



*Рис. 16.1. Каналы на Web-странице*

Теперь наше приложение обладает возможностями получения свежих файлов RSS и отображения данных канала на Web-странице. Нам еще необходимы функции добавления файлов RSS в базу данных и выбора файлов, которые мы хотим отобразить. Все это будет реализовано в следующем скрипте `admin.cgi`.

```
01:  #!/usr/bin/perl -wT
02:  # admin.cgi
03:  use strict;
04:  use CGI qw(:standard);
05:  use CGI::Carp qw(fatalsToBrowser);
06:  use DBI;
07:  my $dbh = DBI->connect("dbi:mysql:book","user","password");
08:  param('Submit') ? add_new() : show_form();
09:  $dbh->disconnect;
```

В строках 1–9 нет ничего нового. Самая важная из них — **строка 8.** Здесь мы выясняем, был ли в скрипт передан параметр Submit. Если это так, следовательно, форма для внесения изменений была отправлена, и мы вызываем подпрограмму *add_new()*.Эта подпрограмма делает в базе данных необходимые изменения и затем повторно отображает форму подпрограммой *show_form()*. Если же такой параметр не был передан, подпрограмма *show_form()*вызывается немедленно. Строка 9 производит отключение от базы данных.

```
10:  sub show_form {
11:    my $sth = $dbh->prepare(qq{select * from rdf});
12:    $sth->execute;
13:    print header,
14:       start_html ("Моя страница - Параметры"),
```

```
15:     • h2("Выберите любимые сайты");
16:     print start_form(-method => 'POST',
                         -action => 'admin.cgi');
```

**Строки 9—16** начинают подпрограмму *show_form()*. Эта подпрограмма служит только для одной цели: отображения Web-формы в броузере. Форма будет состоять из списка всех каналов в базе данных, каждый из которых будет иметь флажок, указывающий, будет ли этот канал отображаться на главной Web-странице. Также форма будет иметь поля ввода для добавления имени нового канала, для URL файла RSS и флажок отображения на главной странице. Примерный вид готовой формы показан на рис. 16.2. **Строка 11** — запрос SQL, который извлекает из базы данных всю информацию. Эти данные мы будем использовать для отображения сведений о каналах начиная со строки 17. Еще одна заслуживающая внимания строка — строка **16,** которая начинает собственно код Web-формы.

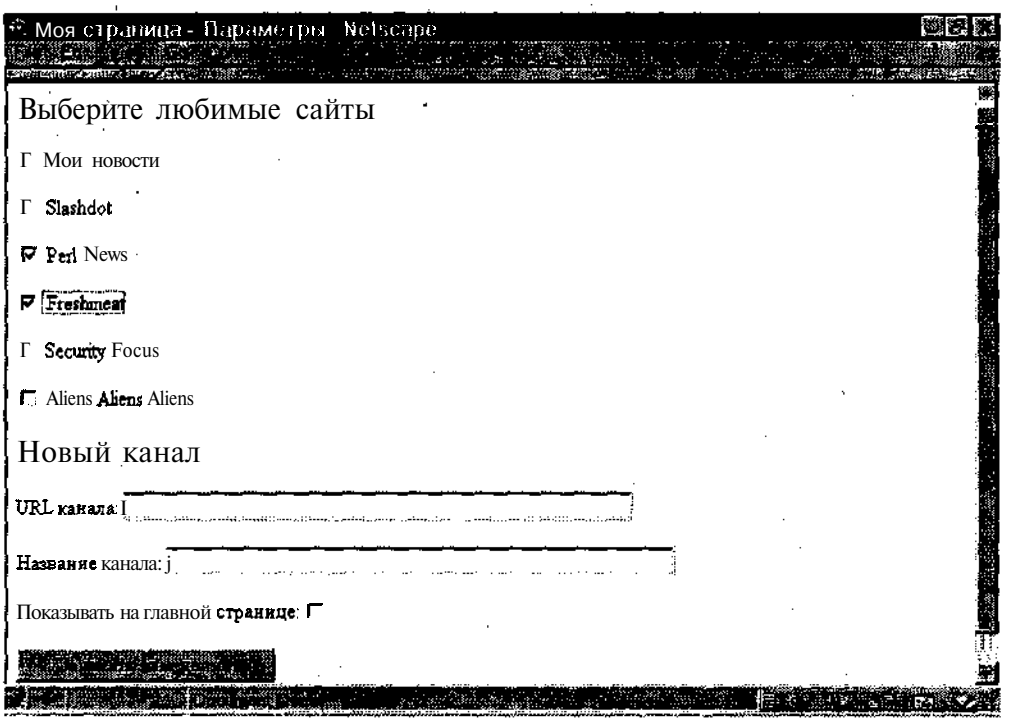



*Рис. 16.2. Страница управления каналами*

```
17:     while (my $data,= $sth->fetchrow_hashref) {
18:       my $checked = $data->{Selected} ? "CHECKED" : "";
19:       print checkbox(-name => 'Selected',
20:                      -checked => $checked,
21:                      -value => $data->{Name},
22:                      -label => $data->{Name},
23:                     ),
24:         p;
25:     }
```

**Строки 17—25** — цикл по набору данных, возвращенному запросом SQL, который был выполнен в строке 12. Переменная $data, в которую возвращает значение метод *fetchrow_hashr()*, является ссылкой на хэш, содержащий данные определенной

строки. Каждый канал в базе данных имеет поле Selected, которое указывает, должен ли канал быть отображен на главной Web-странице. Возможные значения этого поля — 0 и 1, что соответствует запрету отображения и отображению канала. **В строке 18** проверяется значение этого поля и на его основе устанавливается значение переменной `$checked`. Если канал должен быть отображен (`$data->{Selected}` имеет значение истины), в переменную `$checked` заносится строка "CHECKED", а если нет — пустая строка. **В строках 19—23** отображается флажок для данного канала. Через параметры метода *checkbox ()* из `CGI.pm` мы можем указать название флажка — Selected, его начальное состояние (через переменную $checked), а также значение и надпись. Оба последних параметра, value и label, принимают название канала, взятое из `$data->{Name}`. Эти операции выполняются для каждого канала в базе данных. **Строка 25** закрывает блок *while ()*.

```
26:   print h2("Новый канал") , p,
27:     "URL канала: ", textfield(-name=>'URL', -size=>50), p
28:     "Название канала: ", textfield(-name=>'Name',
        -size=>50), p,
29:     "Показывать на главной странице: ",
30:     checkbox(-name=>'new-Selected', -label=>''), p,
31:     submit(-name=>'Submit', -value=>'Записать изменения'),
32:     end_form, end_html;
33: }
```

**Строки 26—32** — один большой оператор *print()*, в котором отображаются три элемента формы для вставки в базу данных нового канала. Первый элемент — текстовое поле с именем URL (созданное методом *texfield()*), в которое будет введен URL файла RSS. **В строке 28** создается второе текстовое поле с именем Name. В него пользователь будет вводить название нового канала. **Строка 30** формирует флажок new-Selected, состояние которого будет определять, должен ли канал отображаться. Вероятно, вы заметили, что имя всех элементов формы, кроме этого, совпадает с именем поля в базе данных. Для флажка выбрано другое имя, чтобы после отправки формы его значение не смешивалось со значениями других установленных флажков. Все остальные установленные флажки будут получены из формы как массив, а этот будет отдельным. **Строка 31** помещает в форму кнопку отправки. Для этой кнопки определено значение, поэтому в строке 8 можно будет выяснить, была ли эта форма отправлена для обновления или просто вызвана. **Строка 32** завершает оператор *print ()*, форму и Web-страницу.

```
34: sub add_new {
35:   my $qry_select = qq(update rdf set Selected = 1
                          where );
36:   my $qry_deselect = qq(update rdf set Selected = 0
                          where );
```

**Строка 34** начинает подпрограмму *add_new()*. **В строках 35 и 36** определяются две переменные, образующие начало двух команд SQL UPDATE. При анализе данных мы определяем, какие каналы выбраны, но никак не можем узнать, какие не были выбраны. Поэтому после выявления выбранных каналов мы должны выполнить два запроса. Первый из них обновит записи выбранных каналов, установив для их полей Selected значение 1, а второй— записи невыбранных, установив значение 0. Эти запросы дополняются в строках 39 и 40.

```
37:   my @selected = param('Selected');
```

**В строке 37** формируется массив `@selected` из значений всех переданных из формы параметров с именем Selected. Эти элементы содержат имена каналов, для которых пользователь установил на форме флажки отображения.

```
38:    $qry_select .= qq(Name = '$_' or ) for @selected;
39:  $qry_deselect .= qq(Name <> '$_' and ) for @selected;
```

**В строках 38 и 39** продолжается формирование команд SQL. К обеим переменным **в цикле for** дописываются все значения из `@selected`.

```
40:    $qry_deselect =~ s! and $!!;
41:    $qry_select =~ s! or $!!;
```

**В строках 40 и 41** в сформированные запросы вносится последнее исправление. В конце каждой строки запроса после выполнения цикла оказывается одно лишнее "and" или "or", которое перед выполнением запроса надо удалить.

```
42:    my $sth = $dbh->prepare($qry_select);
43:    $sth->execute or print $DBI::errstr;
44:    $sth = $dbh->prepare($qry_deselect);
45:    $sth->execute or print $DBI::errstr;
```

**В строках 42—45** запросы подготавливаются и выполняются. После этого состояние базы данных будет соответствовать тому, что указал в форме пользователь.

```
46:   if (param('URL') && param('Name')) {
47:     my $url = param('URL') ;
48:      my $name = param('Name');
49:     my $display = param('new~Selected') ? 1 : 0;
50:      $sth = $dbh->prepare(qq{insert into rdf
           {URL, NAME, SELECTED)
         values ('$url', '$name', Sdisplay)});
51:     $sth->execute or print $DBI::errstr;
52:   }
53:    show_form;
54: }
```

**Строки 46—54** завершают скрипт. В строке 46 проверяется, были ли переданы в скрипт параметры URL И Name. Мы проверяем наличие обоих параметров, чтобы не допустить появления канала без имени и без URL. Если эти параметры присутствуют, в базу данных добавляется новая запись, создавая таким образом новый канал. После этого вызывается подпрограмма *show_form()* и на экране отображается форма с обновленной информацией.

Три скрипта, показанные в этом разделе, демонстрируют приемы создания файла RSS, а также его анализа, чтения и отображения. Эти приемы можно применить для подачи или получения информации в Web-приложениях. Но приложение можно дополнить и другими функциями, которые упоминаются в упражнениях в конце этой главы. Популярность XML растет, и этот язык представляет собой превосходное дополнение в наборе инструментов программиста. Овладев основными методиками, описанными в этой главе, вы уже можете применять Perl и XML для совершенствования Web-приложений.

# Создание файла RSS

Теперь, когда вы научились использовать файлы RSS и ознакомились с их структурой, самое время показать, как можно самостоятельно создать файл канала. Чтобы файл RSS приносил в приложении какую-то пользу, необходимо, чтобы самый существенный элемент канала, а именно новости, которые вы хотите распространить, был доступен в Web. Для примера мы предположим, что эта информация хранится в текстовом файле. Конечно, ваша информация может находиться в базе данных или иметь форму файлов HTML в особом каталоге, или представлять источник данных какого-то другого типа.

Следующий пример, `make_rss`, не является скриптом CGI. Это скрипт командной строки, который можно запускать периодически через планировщик, чтобы автомати-

чески создавать файл RSS. Конечно, этот код можно выполнять и как CGI, но если ваши новости изменяются ежеминутно, наиболее целесообразный способ — автоматическое выполнение через определенные интервалы.

```
01: #!/usr/bin/perl -wT
02: # make_rss
03: use strict;
04: use XML::RSS;
```

Строки 1–4 определяют путь к Perl, устанавливают строгий синтаксис и загружают модули XML: :RSS.

```
05: my $FILE = 'news.txt';
06: my $RDF_DIR = './rdf';
```

В строках 5 и 6 создаются две скалярные переменные, которые мы будем использовать позже. **В строке 5** в `$FILE` заносится имя текстового файла, содержащего данные новостей. Этот файл, news.txt, содержит текстовые записи, состоящие из URL сообщения новостей, разделителя—вертикальной черты и описания сообщения. В строке 6 создается переменная $RDF_DIR. Эта переменная содержит имя каталога, в котором должен быть создан файл RSS.

```
07: my $rdf = new XML::RSS;
```

Строка 7 создает новый объект XML: **:RSS.** Ссылка на этот объект сохраняется в $rdf.

```
08: $rdf->channel (title => 'Мои новости',
09:                link => 'http://news.me.com',
10:                language => 'ru',
11:                description => 'Мои новости для вас!',
12:                copyright => 'Copyright 2000++, Me',
13:                pubDate => scalar localtime(time),
14:                lastBuildDate => scalar localtime(time),
15:                managingEditor => 'me@me.com',
16:                webMaster => 'me@me.com'
17:               );
```

В строках 8—17 с помощью метода *channel ()* определяется некоторая часть основных сведений о самом канале и начинается контейнер-канал. Выше в этой главе уже демонстрировался готовый файл RSS файл, в котором можно увидеть, как каждый из этих параметров будет представлен в конечном документе.

```
18: $rdf->image(title => 'Мои новости',
19:             url => 'http://news.me.com/my_news.gif',
20:             link => 'http://news.me.com',
21:             height => 30,
22:             width => 119
23:            );
```

В строках **18—23** создается контейнер image. Этот контейнер необязателен, но если у канала есть какая-то эмблема, этот код поможет вставить ее.

```
24: open(FILE, $FILE) || die "Нельзя открыть $FILE ($!)";
25: while (<FILE>) {
26: my ($url, $desc) = split /\|/;
27: $rdf->add_item(title => $desc,
28:                link => $url
29:                );
30: }
```

В строках **24—30** из текстового файла извлекаются данные и на их основе создаются контейнеры item. **Строка 24** открывает файл для чтения или выходит с ошибкой. Строки 25—30 — цикл *while ()* по всем строкам файла. **В строке 26** прочитанные дан-

ные строки разделяются по символу вертикальной черты (|) на две части, которые заносятся в переменные $url и $desc. В строках 27 и 28 методом *add_item()* в контейнер-канал добавляется новый пункт. Как можно было видеть в примере файла RSS, контейнер пункта item содержит два элемента: заголовок (title) и ссылку (link). Чтобы создать новый пункт, достаточно вызвать метод *add_item()* и передать в него пары параметр-значение для этих элементов. Когда этот цикл закончится, контейнер-канал будет содержать все пункты.

```
31:  $rdf->textinput(title => 'Поиск в новостях',
32:                  description => 'Поиск в архивах',
33:                  name => 'text',
34:                  link => 'http://news.me.com/search.cgi'
35:                 );
```

**В строках 31—35** методом *textinput()* создается одноименный контейнер. Этот контейнер необязателен; он образует поле ввода текста, которое пользователь может применить для поиска на сайте. В значении параметра link передается URL скрипта для поиска, а в остальных параметрах — название, заголовок и описание соответствующей ссылки. После этого в объекте XML::RSS оказывается законченный канал.

```
36:  $rdf->save("$RDF_DIR/my_news.rdf");
```

**Строка 36** завершает скрипт и методом save() записывает файл на диск. Этого достаточно — у нас появился готовый файл RSS! Скрипт подобного типа можно составить и написать за короткое время, и он почти или вообще не нуждается в сопровождении.

Теперь вы овладели всем циклом RSS начиная с создания источника данных и до использования файла RSS для Web.

# Упражнения

- Описанное нами приложение позволяет получать список каналов и добавлять новые. Однако оно не имеет функции удаления каналов. Дополните скрипт admin.cgi этой функцией.
- Создайте собственный канал. Если ваша компания выпускает собственные новости или вы просто хотите дать ссылки на ваши любимые сайты, сформируйте с помощью модуля XML::RSS ваш собственный файл RSS. Затем добавьте этот канал в базу данных и посмотрите на результат.
- На базе приложения, созданного в этой главе, и других материалов из этой книги постройте многопользовательскую систему каналов новостей.

# Листинги

### Листинг 16.3. Скрипт fetch

```
01:  #!/usr/bin/perl -w
02:  # fetch
03:  use strict;
04:  use File::Basename;
05:  use DBI;
06: use LWP::Simple qw(mirror),-
07:  my $RDF_DIR = './rdf';
08:  my $dbh = DBI->connect("dbi:mysql:book","user","password");
09:  my $sth = $dbh->prepare(qq{select URL from rdf});
10:  $sth->execute or die $DBI::errstr;
```

```
11:  while (my $url = $sth->fetchrow) 1
12:    my $name = basename($url);
13:    mirror($url, "./$RDF_DIR/$name");
14:  }
15:  $dbh->disconnect;
```

**Листинг 16.4. Скрипт index.cgi**

```
01:  #!/usr/bin/perl -wT
02:  # index.cgi
03:  use strict;
04:  use CGI qw(:standard end_ul end_table);
05:  use CGI::Carp qw(fatalsToBrowser);
06:  use File::Basename;
07:  use DBI;
08:  use XML::RSS;
09:  my $RDF_DIR = './rdf';
10:  my $dbh = DBI->connect("dbi:mysql:book","user","password")
       or print $DBI::errstr;
11:  my $sth = $dbh->prepare(qq{select URL from rdf
       where Selected =1});
12:  $sth->execute;
13:  print header,
14:    start_html("Моя начальная страница"),
15:    h2("Мои любимые сайты");
16:  print start_table({cellpadding=>0, cellspacing=>0,
       border=>0, width=>'100%'}),
17:    td;
18:  my $count = 1;
19:  my @html = ('</TD><TD>', '</TD><TR><TD>');
20:  while (my $url = basename($sth->fetch row)) {
21:    my $rss = new XML::RSS;
22:    eval {$rss->parsefile("$RDF_DIR/$url")};
23:    warn "Не удается проанализировать $url - $@"
         and next if $@;
24:    my $last_mod = scalar
         localtime((stat("$RDF_DIR/$url"))[9]);
25:    print start_table({cellpadding=>0,
         cellspacing=>2, border=>5, width=>'75%'}),
26:      td({valign=>'CENTER', bgcolor=>'#C0C0C0'});
27:    $rss->{image}.{url}
28:     ? print img({src=>$rss->{image}{url}})
29:     : print strong($rss->{channel}{title});
30:    print ul;
31:    for (@{$rss->{items}}) {
32:      print li(a({href=>$_->{link}}, $_->{title}));
33:    }
34:    print end_ul;
35:    if ($rss->{textinput}{link}) {
36:      print $rss->{textinput}{description},
           start_form(-method => 'GET',
37:                   -action => $rss->{textinput}{link}),
38:        textfield(-name => $rss->{textinput}{name}),
39:        end_form;
40:    }
41:    print qq(Последнее обновление: $last_mod<BR>),
42:      end_table;
```

```
43:     $html[$count^=1];
44:  }
45:  print end_table,
4  6:  end_html;
```

**ЛИСТИНГ 16.5. Скрипт admin.cgi**

```
01:  #!/usr/bin/perl -wT
02 : # admin.cgi
03:  use strict;
04:  use CGI qw(:standard);
05:  use CGI::Carp qw(fatalsToBrowser);
06:  use DBI;
07:  my $dbh = DBI->connect("dbi:mysql:book","user","password");
08: param( 'Submit') ? add_new () : show_form();
09:  $dbh->disconnect;
10:  sub show_form {
11:   my $sth = $dbh->prepare(qq{select * from rdf});
12:    $sth->execute;
13:    print header,
14:      start_html ("Моя страница - Параметры"),
15:      h2("Выберите любимые сайты");
16:    print start_form(-method => 'POST',
                        -action => 'admin.cgi');
17:    while (my $data = $sth->fetchrow_hashref) {
18:      my Schecked - $data->{Selected} ? "CHECKED" : "";
19:      print checkbox(-name => 'Selected',
20:                     -checked => Schecked,
21:                     -value => $data->{Name},
22:                     -label => $data->{Name},
23:                    ),
24:        p;
25:    }
26:    print h2("Новый канал") , p,
27:      "URL канала: ", textfield(-name=>'URL', -size=>50), p
28:      "Название канала: ", textfield(-name=>'Name',
          -size=>50), p,
29:      "Показывать на главной странице: ",
30:      checkbox(-name=>'new-Selected', -label=>''}, p,
31:      submit(-name=>'Submit', -value=>'Записать изменения'),
32:      end_form, end_html;
33:  }
34:  sub add_new {
35:    my $qry_select = qq(update rdf set Selected = 1
                           where );
36:    my $qry_degelect = qq(update rdf set Selected = 0
                           where );
37:    my @selected = param('Selected');
38:    $qry_select .= qq(Name = '$_' or ) for @selected;
39:    $qry_deselect .= qq(Name <> '$_' and ) for @selected;
40:    $qry_deselect =~ s! and $!!;
41:    $qry_select =~ s! or $!!;
42:    my $sth = $dbh->prepare($qry_select);
43:    $sth->execute or print $DBI::errstr;
44:   $sth = $dbh->prepare ($qry_deselect);
45:    $sth->execute or print $DBI::errstr;
46:   if (param('URL') && param('Name')) {
```

```
47:     my $url = param('URL');
48:     my $name = param('Name');
49:     my $display = param('new-Selected') ? 1 : 0;
50:     $sth = $dbh->prepare(qq{insert into rdf
          (URL, NAME, SELECTED)
        values ('$url', '$name', $display)});
51:     $sth->execute or print $DBI::errstr;
52:   }
53:   show_form;
54: }
```

Листинг 16.6. **Скрипт make_rss**

```
01: #!/usr/bin/perl -wT
02: # make_rss
03: use strict;
04: use XML::RSS;
05: my $FILE = 'news.txt';
06: my $RDF_DIR = './rdf';
07: my $rdf = new XML::RSS;
08: $rdf->channel (title => 'Мои новости',
09:                 link => 'http://news.me.com',
10:                 language => 'ru',
11:                description => 'Мои новости для вас!',
12:                copyright => 'Copyright 2000++, Me',
13:                pubDate => scalar localtime(time),
14:                lastBuildDate => scalar localtime(time),
15:                managingEditor => 'me@me.com',
16:                webMaster => 'me@me.com'
17:               );
18: $rdf->image(title => 'Мои новости',
19:             url => 'http://news.me.com/my_news.gif',
20:             link => 'http://news.me.com',
21:             height => 30,
22:             width => 119
23:            );
24: open(FILE, $FILE) II die "Нельзя открыть $FILE ($!)';
25: while (<FILE>) {
26: my ($url, $desc) = split /\|/;
27: $rdf->add_item(title => $desc,
28:         link => $url
29:               );
30: }
31: $rdf->textinput(title => 'Поиск в новостях',
32:                 description => 'Поиск в архивах',
33:                 name => 'text',
34:                 link => 'http://news.me.com/search.cgi'
35:                 );
36: $rdf->save("$RDF_DIR/my_news.rdf");
```


# Коды сервера

| 100–199 Подтверждение, что запрос обрабатывается | | |
|---|---|---|
| **Код** | **Значение** | **Пояснение** |
| **100** | **Continue** (Продолжить) | Запрос закончен и процесс может продолжаться. |
| 101 | Switching Protocols (Переключение протоколов) | Принят запрос на переключение протоколов (например, с HTTP на FTP). |

| 200–299 Запрос выполнен | | |
|---|---|---|
| **Код** | **Значение** | **Пояснение** |
| 200 | OK | Транзакция завершена успешно. |
| 201 | Created (Ресурс создан) | Завершена транзакция POST и создан новый URL. |
| 202 | Accepted (Запрос принят) | Запрос принят, но сервер еще обрабатывает его. |
| 203 | Non Authoritative Неавторитетный) | Принятая информация в заголовке объекта исходит не от первоначального сервера, а от третьего лица. |
| 204 | No Content (Содержимое отсутствует) | Полученный запрос не указывает информацию, которую надо направить в ответ. |
| 205 | Reset Content (Восстановить содержимое) | Сервер выполнил запрос, и агент пользователя должен восстановить вид документа, из которого был отправлен запрос. |
| 206 | Partial Information (Содержимое неполно) | Возвращенная информация может иметь форму, которую клиент не поддерживает. |

## 300–399 Запрос перенаправлен

| Код | Значение | Пояснение |
|---|---|---|
| 300 | Multiple Choices (Несколько вариантов) | Запрашиваемый адрес указывает более чем на один объект. |
| 301 | Moved Perm. (Постоянное перемещение) | Запрашиваемым данным был присвоен новый URL. |
| 302 | Moved Temp. (Временное перемещение) | Запрашиваемым данным был присвоен новый URL, и перенаправление может быть изменено. |
| 303 | See Other (См. другой) | Для перенаправления требуется протокол, отличный от указанного в первоначальном запросе. |
| 304 | Not Modified (Не изменено) | Клиент послал условный запрос GET, но документ не изменился после даты и времени, указанных в запросе. |
| 305 | Use Proxy (Используйте прокси) | Сообщает серверу, что к требуемому документу нужно обращаться через прокси-сервер. |

## 400–499 Запрос неправильный

| Код | Значение | Пояснение |
|---|---|---|
| 400 | Bad Request (Ошибочный запрос) | В запросе применен неправильный синтаксис. |
| 401 | Unauthorized (Неавторизованный запрос) | Запрос не прошел проверки подлинности. |
| 402 | Payment Required (Требуется оплата) | Запрос может быть выполнен, только если клиент согласится заплатить за транзакцию. |
| 403 | Forbidden (Запрещено) | Запрос запрещен. Проверка подлинности не помогла. Обычно запрашиваемый объект защищен элементом в списке ACL, связанном с каталогом или файлом. |
| 404 | Not Found (Ресурс не найден) | Сервер не смог найти запрашиваемый URL |
| 405 | Method Not Allowed (Метод не разрешен) | Метод, применяемый для обращения к файлу, не разрешается. |
| 406 | Not Acceptable (Неприемлемый) | Запрашиваемая страница существует, но вы не можете ее увидеть, так как ваша система не поддерживает формат, для которого сконфигурирована страница. |
| 407 | Proxy Authorization Needed (Требуется авторизация прокси) | Чтобы запрос был выполнен, он должен пройти проверку подлинности. |
| 408 | Time Out (Превышено время ожидания) | Превышено время ожидания ответа. |
| 409 | Conflict (Конфликт) | Слишком много пользователей запрашивает этот файл одновременно. Сервер перефужен. Попробуйте еще раз. |
| 410 | Gone (Потеряно) | Здесь должна быть страница, но ее нет. |
| 411 | Length Required (Требуется длина) | В запросе отсутствует заголовок Content-Length. |

## 400-499 Запрос не завершен

| Код | Значение | Пояснение |
|---|---|---|
| 412 | Precondition Failed (Не выполняется заданное условие) | Запрашиваемая страница задает некоторое предварительное условие, которому ваш запрос не удовлетворяет. |
| 413 | Request Too Large (Длина запроса слишком велика) | Запрашиваемые данные слишком велики, чтобы их можно было обработать. |
| 414 | URI Too Large (Длина URI-адреса в запросе слишком велика) | Введен слишком длинный URL. |
| 415 | Unsupported Media Type (Неподдерживаемый тип носителя) | Страница имеет неподдерживаемый тип носителя; например, это специальный файл, созданный для определенной программы. |

## 500-599 Внутренние ошибки сервера

| Код | Значение | Пояснение |
|---|---|---|
| 500 | Internal Error (Внутренняя ошибка) | Сервер встретил неожиданное условие, которое препятствует ему выполнить запрос. |
| 501 | Not Implemented (Не реализовано) | Сервер не поддерживает функцию, необходимую для выполнения запроса. |
| 502 | Service Unavailable (Служба недоступна) | Когда сервер обращается к какой-то другой службе, это сообщение указывает, что другая служба не ответила в течение заданного времени. |
| 503 | Service Temporarily Overloaded (Служба временно перегружена) | Служба не смогла обработать запрос вследствие перегрузки большим количеством запросов. |
| 504 | Gateway Timeout (Превышено время ожидания в шлюзе) | Превышено время ожидания в шлюзе. |
| 505 | HTTP Version Not Supported (Версия HTTP не поддерживается) | Запрашиваемый протокол HTTP не поддерживается. |

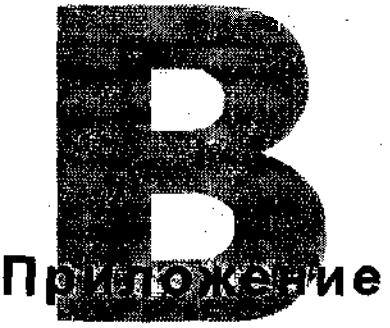
Приложение

# Переменные окружения

Ниже приводится частичный список наиболее часто используемых переменных окружения. В документации по вашему Web-серверу можно найти другие переменные, а также исключить те, которые на нем не присутствуют. В главе 4 приведен скрипт, позволяющий просмотреть все переменные окружения, поддерживаемые вашим Web-сервером.

| Переменная | Пояснение |
|---|---|
| AUTH_TYPE | Протокол, применяемый для проверки подлинности. Устанавливается только при наличии какой-то проверки подлинности, например, основной (Basic Authentication) или краткой (Digest Authentication). |
| CONTENT_LENGTH | Длина в байтах сообщения, передаваемого через входной поток. |
| DOCUMENT_ROOT | Путь на сервере к корневому каталогу дерева документов Web. |
| GATEWAYJNTERFACE | Версия спецификации CGI, которую реализует Web-сервер. |
| HTTP_REFERER | Адрес URL, с которого посетитель пришел на страницу. Если он использовал закладку своего броузера или ввел URL вручную, эта переменная имеет пустое значение. |
| HTTP_USER_AGENT | Название и версия клиента, используемого для просмотра страницы. |
| PATH INFO | Дополнительная информация пути, переданная клиентом. Это может быть например, каталог, указанный после скрипта, например, `/hello/world` в составе адреса `http://you.com/script.cgi/hello/world`. |
| PATH_TRANSLATED | Перевод PATHJNFO в путь на сервере. Это может быть полный путь к DOCUMENT_ROOT, за которым следует относительный путь. Например, `/usr/local/httpd/htdocs/foo/bar`. |
| QUERY_STRING | Информация запроса, переданная вызывающим URL. Это данные, которые следуют после вопросительного знака за именем скрипта. |
| REMOTE_ADDR | IP-адрес запрашивающего клиента. |

| Переменная | Пояснение |
|---|---|
| REMOTE_HOST | Имя компьютера, делающего запрос. Оно устанавливается, только если для сервера разрешен обратный просмотр. |
| REMOTE_USER | Имя пользователя, если пользователь проходит проверку подлинности для обращения к защищенному скрипту. |
| REQUEST_METHOD | Метод, используемый для запроса. Обычно GET, POST или HEAD. |
| SCRIPT_NAME | Виртуальный путь к выполняемому скрипту. |
| SERVER_NAME | Имя или IP-адрес Web-сервера. |
| SERVER_PORT | Порт прослушивания Web-сервера. |
| SERVER_PROTOCOL | Название и версия протокола запроса, например, HTTP/1.0. |
| SERVER_SOFTWARE | Название и версия программы Web-сервера. |




# Форматы POSIX::strftime()

| Формат | Пояснение |
|---|---|
| %a | Сокращенное название дня недели в соответствии с настройками языка. |
| %A | Полное название дня недели в соответствии с настройками языка. |
| %b | Сокращенное название месяца в соответствии с настройками языка. |
| %B | Полное название месяца в соответствии с настройками языка. |
| %c | Предпочтительное представление даты и времени в соответствии с настройками языка. |
| %C | Номер столетия (год/100) как целое число с двумя цифрами. (Единая спецификация UNIX.) |
| %d | Число месяца в десятичном виде (в диапазоне от 01 до 31). |
| %D | Эквивалент %m/%d/%y. (Единая спецификация UNIX.) |
| %e | Как %d, число месяца в десятичном виде, но ведущий нуль заменен пробелом. (Единая спецификация UNIX.) |
| %E | Модификатор: используйте альтернативный формат, см. ниже. (Единая спецификация UNIX.) |
| %G | Год со столетием в десятичном виде ло ISO 8601. Год из четырех цифр с учетом номера недели по ISO (см. %V). Формат и значение года совпадают с %y, с тем исключением, что, если номер недели по ISO относится к предыдущему или следующему году, то используется этот год вместо текущего. (По местному времени.) |
| %д | Как %G, но без столетия — т.е. год из двух цифр (00-99). (По местному времени.) |
| %h | Эквивалент %b. (Единая спецификация UNIX.) |

| Формат | Пояснение |
|---|---|
| %H | Час в десятичном виде по 24-часовому счету (в диапазоне от 00 до 23). |
| %I | Час в десятичном виде по 12-часовому счету (в диапазоне от 01 до 12). |
| %j | День года в десятичном виде (в диапазоне от 001 до 366). |
| %к | Час в десятичном виде по 24-часовому счету (в диапазоне от 0 до 23); перед часом из одной цифры ставится пробел. (См. также %H.) (По местному времени.) |
| %l | Час в десятичном виде по 12-часовому счету (в диапазоне от 1 до 12); перед часом из одной цифры ставится пробел. (См. также %I.) (По местному времени.) |
| %m | Месяц в десятичном виде (в диапазоне от 01 до 12). |
| %M | Минуты в десятичном виде (в диапазоне от 00 до 59). |
| %п | Символ новой строки. (Единая спецификация UNIX.) |
| %O | Модификатор: используйте альтернативный формат, см. ниже. (Единая спецификация UNIX.) |
| %p | "AM" либо "PM" в зависимости от значения времени или соответствующие строки согласно настройкам языка. Полдень считается "PM", а полночь — "AM". |
| %P | Как %p, но в нижнем регистре: "am" или "pm" либо соответствующие строки согласно настройкам языка. (GNU) |
| %г | Время в записи "AM" или "PM". В POSIX соответствует "%I:%M:%S %p". (Единая спецификация UNIX.) |
| %R | Время в 24-часовой записи (%H:%M). (Единая спецификация UNIX.) Отображение секунд см. в версии %T ниже. |
| %s | Количество секунд с начала эпохи, т.е. с 00:00:00 UTC 01.01.1970. (По местному времени.) |
| %S | Секунды в десятичном виде (в диапазоне от 00 до 59). |
| %t | Символ табуляции. (Единая спецификация UNIX.) |
| %T | Время в 24-часовой записи (%H:%M:%S). (Единая спецификация UNIX.) |
| %u | День недели как десятичное число в диапазоне от 1 до 7, понедельник — 1. См. также %w. (Единая спецификация UNIX) |
| %U | Номер недели текущего года — десятичное число в диапазоне от 00 до 53, причем первое воскресенье года считается первым днем недели 01. См. также %V и %W. |
| %V | Номер недели текущего года по ISO 8601:1988 — десятичное число в диапазоне от 01 до 53, причем неделей 01 считается первая неделя, имеющая в текущем году не менее четырех дней, а первым днем недели считается понедельник. См. также %U и %W. (Единая спецификация UNIX.) |
| %w | День недели как десятичное число в диапазоне от 0 до 6, воскресенье — 0. См. также %u. |
| %W | Номер недели текущего года — десятичное число в диапазоне от 00 до 53, причем первый понедельник года считается первым днем недели 01. |
| %x | Предпочтительное представление даты в соответствии с настройками языка, без времени. |

| Формат | Пояснение |
|---|---|
| %X | Предпочтительное представление времени в соответствии с настройками языка, без даты. |
| %y | Год без столетия как десятичное число (в диапазоне от 00 до 99). |
| %Y | Год как десятичное число, включая столетие. |
| %z | Часовой пояс как сдвиг относительно среднего времени по Гринвичу. Необходимо для генерации дат в соответствии с требованиями RFC 822 (в формате "%a, %d %b %Y %H:%M:%S %z"). (GNU.) |
| %Z | Часовой пояс, его название или сокращение. |
| %+ | Дата и время в формате date(1). (По местному времени.) |
| %% | Символ "%". |

# Приложение Г

# Общедоступная лицензия

## ОБЩЕДОСТУПНАЯ ЛИЦЕНЗИЯ GNU (GENERAL PUBLIC LICENSE GNU)

Версия 2, июнь 1991 г.

Copyright (C) 1989, 1991 Free Software Foundation, Inc.

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

Копирование и распространение точных копий данного документа разрешается всем, но вносить в него какие либо изменения запрещается.

### Преамбула

Лицензии для большей части программ составлены так, чтобы лишить вас свободы в их передаче и модификации. В противоположность этому Общедоступная лицензия GNU предназначена для того, чтобы гарантировать вашу свободу в распространении и изменении свободно используемых программ, обеспечивая их свободное применение всеми пользователями. Эта Общедоступная лицензия применяется к большей части программ Free Software Foundation и ко всем иным программам, авторы которых передают их в свободное использование. (Некоторые другие программы Free Software Foundation подлежат действию Общедоступной лицензии GNU для библиотек). Вы также можете распространить ее на свои программы. Когда речь идет о свободно используемых программах, имеется в виду свобода, а не цена. Наши Общедоступные лицензии сформулированы так, чтобы гарантировать вам возможность свободно распространять копии свободно используемых программ (и получать плату за эту услугу, если пожелаете), получать исходные коды или иметь возможность их получить, изменять программы или применять их части для создания новых свободно используемых программ и знать, что вы можете это делать.

Для зашиты ваших прав нам необходимо установить ограничения, которые запрещают кому бы то ни было нарушать эти права, или просить вас отказаться от таких прав. Эти ограничения имеют своим следствием для вас определенную ответственность, если вы распространяете копии программ или модифицируете их.

Например, если вы распространяете копии таких программ бесплатно или за вознаграждение, вы должны передать получателям права на них, которые вы имели. Вы должны быть уверены, что они также получат или будут иметь возможность получить исходные коды. И вы должны сообщить им эти условия, чтобы они знали свои права.

Мы защищаем ваши права в два этапа: (1) распространяем на программы авторское право и (2) передаем вам эту Лицензию, которая дает вам законное право копировать, распространять и/или модифицировать программы. Кроме того, для защиты каждого автора и нашей собственной защиты мы хотим быть уверенными, что все понимают, что на эти свободно используемые программы отсутствует гарантия. Если программа кем-то модифицирована и передана дальше, мы хотим, чтобы ее получатели знали, что то, что они имеют, не является оригиналом, и поэтому все проблемы, порожденные другими лицами, не должны затрагивать репутацию автора оригинала.

И наконец, любой свободно используемой программе постоянно угрожает патентование. Мы хотим избежать опасности, что вторичные распространители свободно используемой программы самостоятельно запатентуют ее, сделав своей собственностью. Чтобы помешать этому, мы хотим со всей определенностью сказать, что любой патент должен либо свободно предоставлять право на использование программы всем, либо не предоставлять его никому. Конкретные положения и условия копирования, распространения и модификации приводятся ниже.

## ПОЛОЖЕНИЯ И УСЛОВИЯ КОПИРОВАНИЯ, РАСПРОСТРАНЕНИЯ И МОДИФИКАЦИИ

0. Настоящая Лицензия применима к любой программе или другой работе, содержащей уведомление, помещенное обладателем авторского права и указывающее, что она может распространяться в соответствии с требованиями Общедоступной лицензии. Употребляемый далее термин "Программа" относится к любой программе или работе, а "работа, основанная на Программе" означает программу или любую производную от нее работу, находящуюся под защитой законов об авторском праве; иными словами, работу, содержащую Программу или ее часть либо дословно, либо с модификациями и/или в переводе на другой язык. (Далее перевод включается без дополнительных оговорок в термин "модификация"). Далее под обращением "вы" понимается обладатель лицензии. Другая деятельность, отличная от копирования, распространения и модификации, не подпадает под действие данной Лицензии и находится за пределами ее влияния. Собственно эксплуатация Программы ничем не ограничивается и выходные данные, выдаваемые Программой, подпадают под действие Лицензии, только если их содержимое составляет работу, основанную на Программе (независимо от цели, для которой выполняется Программа). Действительно ли это так, зависит от того, что делает Программа.

1. Вы можете копировать и распространять точные копии исходного кода Программы, как вы их получили, на любом носителе, при условии, что вы поместите на каждой копии заметное и подобающим образом составленное уведомление об авторском праве и об отказе в гарантии, сохраните без изменений все записи, относящиеся к данной Лицензии и к отсутствию гарантии, и передадите каждому получателю Программы копию данной Лицензии вместе с копией Программы. Вы можете получать плату за саму физическую процедуру передачи копии и по вашему усмотрению можете за плату предложить гарантийную защиту.

2. Вы можете модифицировать свою копию или копии Программы или любую ее часть, реализуя таким образом работу, основанную на Программе, и копировать и распространять эти модификации и другие работы согласно положениям Раздела 1, обеспечив также соблюдение всех следующих условий:

а). Вы должны снабдить модифицированные файлы заметным уведомлением о том, что вы изменили файлы, и датой изменения.

**b).** Вы должны предоставить для любой работы, которую вы распространяете или публикуете и которая полностью или частично содержит или основана на Программе или любой ее части, разрешение на ее использование в полном объеме и бесплатно третьими лицами согласно положениям Лицензии.

с). Если модифицированная программа обычно должна при своей работе воспринимать команды в интерактивном режиме, вы должны обеспечить, чтобы при запуске для такого интерактивного использования наиболее обычным способом она выводила или отображала объявление, включающее соответствующее сообщение об авторском праве и уведомление об отсутствии гарантии (или ее наличии, если вы предоставляете гарантию) и о том, что пользователь может заниматься дальнейшим распространением программы при соблюдении этих условий, а также информацию о том, как пользователь может просмотреть копию данной Лицензии. (Исключение: Если Программа сама по себе является интерактивной, но в нормальном режиме не выдает такого объявления, ваша работа, основанная на Программе, не обязана выводить это объявление).

Эти требования применимы к модифицированной работе как к целому. Если поддающиеся идентификации части этой работы не являются производными от Программы и могут обоснованно рассматриваться как независимая и отдельная работа, то данная Лицензия и ее положения не применимы к этим частям в случае, если они распространяются как отдельные работы. Но когда вы распространяете эти части в составе работы, основанной на Программе, распространение работы в целом должно подпадать под положения настоящей Лицензии, разрешения которой для других обладателей Лицензии распространяются на работу в полном объеме, то есть на все ее части, независимо от того, кто их написал.

Таким образом, целью данного раздела является не заявление прав или оспаривание ваших прав на работу, написанную исключительно вами, а осуществление права на контроль над распространением производных или коллективных работ , основанных на Программе. Кроме того, простое объединение Программы (или работы, основанной на Программе) с другой работой, не основанной на Программе, на одном носителе для хранения или распространения не переносит действие Лицензии на эту другую работу.

3. Вы можете копировать и распространять Программу (или работу, основанную на Программе, согласно определениям Раздела 2) в объектном коде или исполняемой форме согласно положениям Разделов 1 и 2 при условии, что вы также выполните одно из следующих требований.

а). Сопроводите ее полным машиночитаемым исходным кодом, который должен распространяться согласно положениям Разделов 1 и 2, на носителе, обычно используемом для обмена программами.

**b).** Сопроводите ее письменным предложением, действительным в течение не менее трех лет, предоставить любой третьей стороне за плату, не большую ваших расходов на физическое распространение исходного кода, полную машиночитаемую копию соответствующего исходного кода для распространения согласно положениям Разделов 1 и 2, на носителе, обычно используемом для обмена программами.

с). Сопроводите ее информацией, которую вы получили в качестве предложения по распространению соответствующего исходного кода. (Этот вариант допускается только для некоммерческого распространения и только ес-

ли вы получили программу в объектном коде или исполняемой форме вместе с таким предложением, в соответствии с Подразделом b).

Под исходным кодом работы понимается форма работы, предпочтительная для выполнения ее Модификации. Для работы в исполняемой форме полным исходным кодом считается весь исходный код всех модулей, которые она включает, все связанные с ней файлы определения интерфейсов и скрипты, используемые для управления компиляцией и установкой исполняемых файлов. Однако, в качестве особого исключения распространяемый исходный код не обязательно должен содержать то, что обычно распространяется (в исходном коде или двоичной форме) вместе с основными компонентами (компилятором, ядром и т.д.) операционной системы, на которой работает исполняемая программа, за исключением ситуации, когда компонент сам по себе сопровождает исполняемую программу. Если распространение исполняемого или объектного кодов осуществляется предоставлением доступа для копирования из указанного места, то предоставление эквивалентного доступа для копирования исходного кода из того же места считается распространением исходного кода, даже если третьи стороны не принуждаются копировать исходный код вместе с объектным кодом.

4. Вы не имеете права копировать, модифицировать, выдавать сублицензии или распространять Программу иначе, чем согласно положениям настоящей Лицензии. Любая другая попытка копировать, модифицировать, выдавать сублицензии или распространять Программу не имеет силы и автоматически аннулирует ваши права согласно настоящей Лицензии. Однако стороны, получившие от вас копии или права согласно данной Лицензии, не теряют своих разрешений, пока полностью соблюдают настоящую Лицензию.

5. Вы не обязаны принимать настоящую Лицензию, коль скоро вы ее не подписывали. Однако ничто другое не дает вам права модифицировать или распространять Программу или производные из нее работы. Эти действия запрещены законом и в том случае, если вы не принимаете данную Лицензию. Поэтому ваши действия по модификации или распространению Программы (или любой работы, основанной на Программе) демонстрируют, что вы принимаете настоящую Лицензию в отношении этих действий, а также все ее положения и условия по копированию, распространению или модификации Программы или работ, основанных на Программе.

6. Каждый раз, когда вы передаете Программу (или работу, основанную на Программе), получатель автоматически получает от предыдущего обладателя разрешение на копирование, распространение или модификацию Программы с учетом данных положений и условий. Вы не имеете права накладывать какие бы то ни было дополнительные ограничения на осуществление получателем предоставленных ему прав. Вы не несете ответственности по претензиям, предъявляемым третьими сторонами к данной Лицензии.

7. Если по решению суда или по заявлению о нарушении патента, или по какой-либо другой причине (не обязательно связанной с патентными проблемами) перед вами ставятся условия (решением суда, договором или чем-то иным), которые противоречат условиям Лицензии, это не освобождает вас от соблюдения условий Лицензии. Если вы не можете выполнять свои обязательства согласно этой Лицензии и любые другие подобные обязательства одновременно, вы вообще не имеете права распространять Программу. Например, если патентная лицензия не разрешает безвозмездное распространение Программы всеми теми, кто прямо или опосредованно получил от вас ее копии, то единственный вариант, позволяющий не нарушать как эти, так и другие требования, заключается в

полном отказе от распространения Программы. Если какая-то часть этого раздела не применима к конкретной ситуации или невыполнима при некоторых конкретных обстоятельствах, то считается применимой остальная часть этого раздела, а в других обстоятельствах считается применимым раздел в полном объеме. Целью данного раздела не является попытка подтолкнуть вас к нарушению каких-либо патентных прав или прав собственности, или оспаривать справедливость таких прав; этот раздел имеет единственную цель — защиту целостности системы распространения свободно используемых программ, которая реализуется практикой общедоступных лицензий. Многие люди внесли шедрый вклад в обширную , коллекцию свободно используемых программ, полагаясь на последовательное применение этой системы. Автор/даритель сам имеет право решать, будут ли его программы распространяться через какие-то другие системы, и мы не можем навязывать ему выбор. Назначение настоящего раздела — разъяснить последствия остальных положений данной Лицензии.

8. Если распространение и/или использование Программы ограничено в некоторых странах либо патентами, либо запатентованными интерфейсами, первоначальный обладатель авторского права, который поместил Программу под действие настоящей Лицензии, может явно указать географию распространения, исключающую эти страны, так что распространение будет разрешено только в странах, не попавших в эти исключения. В данном случае настоящая Лицензия включает ограничение как неотъемлемую часть своего текста.
9. Free Software Foundation может время от времени публиковать пересмотренные и/или новые версии Общедоступной лицензии. *Эти* новые версии будут аналогичны по духу настоящей версии, но могут отличаться в деталях, связанных с новыми проблемами или соображениями. Каждая версия имеет уникальный номер. Если в Программе указан номер версии этой Лицензии,, который относится и к "любой более поздней версии", вы можете следовать положениям и условиям либо данной версии, либо любой более поздней версии, опубликованной Free Software Foundation. Если в Программе не указан номер версии Лицензии, вы можете выбрать любую версию, когда-либо опубликованную Free Software Foundation.
10. Если вы хотите вставить части Программы в другие свободно используемые программы, условия распространения которых отличаются, напишите автору и попросите его разрешения. В случае программ, защищенных авторским правом Free Software Foundation, можете писать в Free Software Foundation; мы иногда делаем для этого исключения. Наше решение будет руководствоваться двумя задачами: сохранением свободного статуса для всех производных наших свободно используемых программ и общим поощрением распространения программ.

## ОТКАЗ В ГАРАНТИИ

И. ПОСКОЛЬКУ ПРОГРАММА ИМЕЕТ БЕСПЛАТНУЮ ЛИЦЕНЗИЮ, НА ПРОГРАММУ ОТСУТСТВУЕТ ГАРАНТИЯ (В ПРЕДЕЛАХ, ДОПУСКАЕМЫХ СООТВЕТСТВУЮЩИМ ЗАКОНОМ). ЗА ИСКЛЮЧЕНИЕМ ТЕХ СЛУЧАЕВ, КОГДА ПИСЬМЕННО УСТАНОВЛЕНО ОБРАТНОЕ, ОБЛАДАТЕЛИ АВТОРСКОГО ПРАВА И/ИЛИ ДРУГИЕ СТОЮНЫ ПОСТАВЛЯЮТ ПРОГРАММУ "КАК ЕСТЬ", БЕЗ КАКОЙ БЫ ТО НИ БЫЛО ГАРАНТИИ, ЯВНОЙ ИЛИ ПОДРАЗУМЕВАЕМОЙ, ВКЛЮЧАЯ ПОДРАЗУМЕВАЕМЫЕ ГАРАНТИИ ТОВАРНОСТИ ПРОГРАММЫ И ЕЕ ПРИГОДНОСТИ ДЛЯ КОНКРЕТНЫХ ЦЕЛЕЙ, НО НЕ ТОЛЬКО ЭТО. ВЫ ПРИНИМАЕТЕ НА СЕБЯ ВЕСЬ РИСК В ОТНОШЕНИИ КАЧЕСТВА И ПРОИЗВОДИТЕЛЬНОСТИ ПРОГРАММЫ. ЕСЛИ ПРОГРАММА ОКАЖЕТСЯ ДЕФЕКТНОЙ, ВЫ НЕСЕТЕ ВСЕ РАСХОДЫ ПО НЕОБХОДИМОМУ ОБСЛУЖИВАНИЮ, ВОССТАНОВЛЕНИЮ ИЛИ КОРРЕКЦИИ.

12. НИ В КАКИХ СЛУЧАЯХ, КРОМЕ СЛЕДУЮЩИХ ИЗ СООТВЕТСТВУЮЩИХ ЗАКОНОВ ИЛИ СОГЛАСИЯ В ПИСЬМЕННОЙ ФОРМЕ, НИ ОДИН ОБЛАДАТЕЛЬ АВТОРСКОГО ПРАВА ИЛИ ДРУГОЕ ЛИЦО, КОТОРОЕ МОГЛО МОДИФИЦИРОВАТЬ ПРОГРАММУ И/ИЛИ ДАЛЕЕ РАСПРОСТРАНИТЬ ЕЕ В СООТВЕТСТВИИ С УКАЗАННЫМИ ВЫШЕ РАЗРЕШЕНИЯМИ, НЕ БУДЕТ НЕСТИ ОТВЕТСТВЕННОСТИ ПЕРЕД ВАМИ ЗА УЩЕРБ, ВКЛЮЧАЯ ЛЮБОЙ УЩЕРБ ОБЩЕГО ИЛИ ЧАСТНОГО ХАРАКТЕРА, СЛУЧАЙНЫЙ ИЛИ ЗАКОНОМЕРНЫЙ, ЯВИВШИЙСЯ РЕЗУЛЬТАТОМ ИСПОЛЬЗОВАНИЯ ИЛИ НЕВОЗМОЖНОСТИ ИСПОЛЬЗОВАНИЯ ПРОГРАММЫ (ВКЛЮЧАЯ ПОТЕРЮ ИЛИ НЕПРАВИЛЬНОЕ ОТОБРАЖЕНИЕ ДАННЫХ ИЛИ ПОТЕРИ, КОТОРЫЕ ПОНЕСЛИ ВЫ ИЛИ ТРЕТЬЯ СТОРОНА, ИЛИ НЕСПОСОБНОСТЬ ПРОГРАММЫ ВЗАИМОДЕЙСТВОВАТЬ С ЛЮБЫМИ ДРУГИМИ ПРОГРАММАМИ, НО НЕ ТОЛЬКО ЭТО), ДАЖЕ ЕСЛИ ОТ ВЛАДЕЛЬЦЕВ ИЛИ ТРЕТЬЕЙ СТОРОНЫ БЫЛА ПОЛУЧЕНА КОНСУЛЬТАЦИЯ О ВОЗМОЖНОСТИ ТАКОГО УЩЕРБА.

**КОНЕЦ ПОЛОЖЕНИЙ И УСЛОВИЙ**

## Как применить эти положения к вашим новым программам

Если вы разработали новую программу и хотите, чтобы она принесла максимально возможную пользу обществу, лучший способ достичь этого — сделать ее свободно используемой, что позволит всем и каждому распространять и изменять ее при выполнении этих положений. Для этого вы должны добавить к своей программе следующее уведомление. Самое надежное — поместить его в начало каждого файла исходного кода, для максимального эффекта сопроводив его уведомлением об отсутствии гарантии; кроме того, каждый файл должен как минимум содержать строку авторского права ("copyright") и указание на то, где находится полный текст уведомления.

одна строка, в которой приводится название программы и ее назначение

Copyright (C) год имя автора

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

Добавьте также информацию, как с вами связаться по электронной и обычной почте. Если программа интерактивная, сделайте так, чтобы при запуске в интерактивном режиме она выводила краткое примечание наподобие следующего:

Gnomovision version 69, Copyright (C) год имя автора

Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type "show w". This is free software, and you are welcome to redistribute it under certain conditions; type "show c" for details.

Гипотетические команды "show w" и "show c" должны показывать соответствующие части Общедоступной лицензии. Разумеется, команды, которые вы будете использовать, могут называться как-то иначе, чем "show w" и "show c"; они даже могут вызываться щелчком мыши или через меню — как лучше подойдет для вашей программы. Кроме того, ваш работодатель (если вы работаете программистом) или руко-

водство вашего учебного заведения должны подписать "отказ от авторских прав" ("copyright disclaimer") на программу, если это необходимо. Вот образец; имена, конечно, надо изменить:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program "Gnomovision" (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, I April 1989

Ty Coon, President of Vice

Эта Общедоступная лицензия не разрешает включать вашу программу в запатентованные программы. Если ваша программа является библиотекой подпрограмм, может быть более целесообразно разрешить компоновать с ней запатентованные приложения. В данном случае следует использовать вместо настоящей Лицензии Общедоступную лицензию GNU для библиотек (GNU Library General Public License).

Приложение Д

# Творческая лицензия

## "Творческая лицензия"

## Преамбула

Предназначение этого документа заключается в том, чтобы установить условия, согласно которым Пакет может копироваться, так, чтобы Обладатель авторского права сохранял некоторое подобие авторского контроля над разработкой пакета, и в то же время пользователям пакета предоставлялось право использовать и распространять Пакет более или менее общепринятым способом, а также право вносить приемлемые модификации.

## Определения

"Пакетом" называется совокупность файлов, распространяемая Обладателем авторского права, и производные от этой совокупности файлов, созданные путем модификации текста.

"Стандартной версией" называется такой Пакет, если он не был изменен или был изменен в соответствии с пожеланиями Обладателя авторского права, как указано ниже.

"Обладателем авторского права" считается сторона, названная в авторском праве или авторских правах на пакет.

"Вы" — это вы, если вы предполагаете копировать или распространять этот Пакет.

"Разумная оплата копирования" — это любая сумма, которую вы можете обосновать исходя из стоимости носителей, расходов на копирование, затраченного времени исполнителей и т.д. (Вы не обязаны обосновывать ее перед Обладателем авторского права, а должны сделать это только по отношению к компьютерному сообществу в целом как к рынку, который должен вносить эту плату.)

"Свободно доступный" означает, что непосредственно за объект не взимается никакая плата, хотя связанные с предоставлением этого объекта услуги могут быть платными. Это также означает, что получатели объекта могут в свою очередь распространять его на тех же условиях, на которых они его получили.

1. Вы можете создавать и раздавать точные копии исходной формы Стандартной версии этого Пакета без ограничений, при условии, что вы продублируете все первоначальные объявления об авторском праве и сопутствующие оговорки.
2. Вы можете применять исправления ошибок, усовершенствования для повышения мобильности и другие модификации, являющиеся всеобщим достоянием или полученные от Обладателя авторского права. Пакет, модифицированный таким способом, продолжает считаться Стандартной версией.
3. Вы можете вносить любые другие изменения в вашу копию данного Пакета при условии, что вы поместите в каждый измененный файл заметное уведомление о том, как и когда вы изменили данный файл, и выполните по крайней мере **ОДНО из** следующих действий:

   **a.** Сделаете ваши модификации всеобщим достоянием (Public Domain) или иным способом сделаете их свободно доступными, **например,** передав эти модификации в Usenet или подобное окружение или поместив эти модификации на крупном архивном сайте, таком как `uunet.uu.net`, или же позволив Обладателю авторского права включить ваши модификации в Стандартную версию Пакета.

   **b.** Будете использовать модифицированный Пакет **только** в пределах вашей корпорации или организации.

   **c.** Переименуете все нестандартные исполняемые файлы таким образом, чтобы их имена не смешивались со стандартными исполняемыми файлами, которые также следует представить, и предоставите для каждого нестандартного исполняемого файла отдельную страницу руководства, в которой должно быть четко указано, в чем он отличается от Стандартной версии.

   d. Иным путем договоритесь с Обладателем авторского права по вопросу распространения.
4. Вы можете распространять программы этого Пакета в объектном коде или исполняемой форме при условии, что выполните по крайней мере ОДНО из следующих действий:

   **a.** Будете распространять Стандартную версию исполняемых файлов и файлов библиотек вместе с указаниями (в электронной или иной форме), где можно получить Стандартную версию.

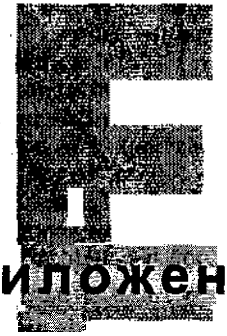
   **b.** Сопроводите распространение исходным кодом Пакета с вашими модификациями в машиночитаемой форме.

   **c.** Дадите нестандартным исполняемым файлам нестандартные имена и четко укажете (в электронной или иной форме) различия, а также предоставите указания, где можно получить Стандартную версию.

   d. Иным путем договоритесь с Обладателем авторского права по вопросу распространения.
5. Вы можете взимать разумную оплату копирования за распространение этого Пакета. Вы можете взимать любую плату, которую вы решите установить, за поддержку этого Пакета. Вы не можете взимать плату непосредственно за этот Пакет. Однако Вы можете распространять этот Пакет в совокупности с другими (возможно, коммерческими) программами как часть большого (возможно, коммерческого) программного дистрибутива при условии, что вы не объявляете этот Пакет вашим собственным изделием. Вы можете внедрять интерпретатор этого Пакета в состав вашего исполняемого файла (путем компоновки), это должно рассматриваться как простая форма объединения, при условии, что таким образом внедряется полная Стандартная версия интерпретатора.

6. Скрипты и файлы библиотек, передаваемые на вход или получаемые на выходе программ этого Пакета, не подпадают автоматически под авторское право этого Пакета, но принадлежат тому, кто их произвел, и могут быть проданы на коммерческой основе и могут быть объединены с этим Пакетом. Если такие скрипты или файлы библиотек объединены с этим Пакетом при помощи так называемых методов "undump" или "unexec" создания двоичного исполняемого образа, то распространение такого образа не должно ни рассматриваться как распространение этого Пакета, ни подпадать под ограничения параграфов 3 и 4 при условии, что вы не объявляете такой исполняемый образ Стандартной версией этого Пакета.
7. Подпрограммы на языке C (или скомпилированные в сравнимую форму подпрограммы на других языках), представленные вами и скомпонованные с этим Пакетом с целью эмуляции подпрограмм и переменных языка, определенного этим Пакетом, должны рассматриваться не как часть этого Пакета, но как эквивалент входных данных согласно параграфу 6 при условии, если эти подпрограммы каким-либо образом не изменяют этот язык так, что это может вызвать негативный результат возвратного тестирования (regression test) языка.
8. Объединение этого пакета с коммерческим дистрибутивом разрешается всегда при условии, что использование этого Пакета является внутренним, то есть не делается явных попыток сделать интерфейсы этого Пакета видимыми для конечного пользователя коммерческого дистрибутива. Такое использование не должно расцениваться как распространение этого Пакета.
9. Имя Обладателя авторского права не может использоваться для рекомендации или продвижения продуктов, полученных из этих программ, без определенного предварительного письменного разрешения.
10. ЭТОТ ПАКЕТ ПРЕДОСТАВЛЯЕТСЯ "КАК ЕСТЬ", БЕЗ КАКОЙ БЫ ТО НИ БЫЛО ЯВНОЙ ИЛИ ПОДРАЗУМЕВАЕМОЙ ГАРАНТИИ, ВКЛЮЧАЯ ПОДРАЗУМЕВАЕМЫЕ ГАРАНТИИ ТОВАРНОСТИ ПРОГРАММЫ И ЕЕ ПРИГОДНОСТИ ДЛЯ КОНКРЕТНЫХ ЦЕЛЕЙ, НО НЕ ТОЛЬКО ЭТО.

Конец



Приложение

# Документация к Perl

Эти документы входят в состав Perl 5.6. Некоторые из них не включены в более ранние версии Perl.

| Документ | Пояснение |
|---|---|
| perlapio | Интерфейс абстракции ввода-вывода для Perl |
| perlbot | Коллекция хитростей для объектов (Bag o' Object Tricks) |
| perlcall | Соглашение о вызовах стандарта С для Perl |
| perldata | Типы данных Perl |
| perldebug | Отладка в Perl |
| perldelta | Что нового в Perl 5.x |
| perldiag | Различные средства диагностики в Perl |
| perldsc | "Кухня" структур данных Perl |
| perlembed | Как внедрять Perl в программы на С |
| perlfaq | Часто задаваемые вопросы о Perl и списки FAQ |
| perlfaq1 | Общие вопросы о Perl |
| perlfaq2 | Получение Perl и его изучение |
| perlfaq3 | Программные средства |
| perlfaq4 | Работа с данными |

| Документ | Пояснение |
|---|---|
| perlfaq 5 | Файлы и форматы |
| perlfaq6 | Регулярные выражения |
| perlfaq7 | Общие проблемы языка Peri |
| perlfaq8 | Взаимодействие с системой |
| perlfaq9 | Сети |
| perlform | Форматы Perl |
| perlfunc | Встроенные функции Perl |
| perlguts | Внутренние функции Perl |
| perlhist | История Perl |
| perlipc | Межпроцессное взаимодействие в Perl |
| perlocale | Национальные настройки в Perl |
| perllol | Работа со списками списков в Perl |
| perlmod | Модули Perl |
| perlmodinstall | Установка модулей CPAN |
| perlmodlib | Формирование новых модулей Perl и редактирование существующих |
| perlobj | Объекты Perl |
| perlop | Операторы и приоритеты Perl |
| perlopentut | Начальный учебник по Perl |
| perlpod | Простая старая документация (Plain Old Documentation, POD) |
| perlport | Написание переносимого кода в Perl |
| perlre | Регулярные выражения Perl |
| perlref | Ссылки и вложенные структуры данных в Perl |
| perlreftut | Очень краткий учебник Марка по ссылкам |
| perlrun | Как работать с интерпретатором Perl |
| perlsec | Безопасность в Perl |
| peristyle | Руководство по стилю Perl |
| perlsub | Подпрограммы Perl |
| perlsyn | Синтаксис Perl |
| perlthrtut | Учебник по потокам в Perl |
| perltie | Использование связывания в Perl |
| perltoc | Оглавление документации к Perl |

| Документ | Пояснение |
|---|---|
| perltoot | Учебник Тома по объектному программированию в Perl |
| **perltraps** | **Perl** — ловушки для неосторожных |
| **perlvar** | Стандартные переменные **Perl** |
| perlxs | Описание языка XS |
| **perlxstut** | Учебник по элементам XSUB |


# Коды ASCII

| Десятичное значение | Шестнадцатеричное значение | ASCII | Описание |
|---|---|---|---|
| 0 | 00 | NUL | CTRL/1 |
| 1 | 01 | SOH | CTRL/A |
| 2 | 02 | STX | CTRL/B |
| 3 | 03 | ETX | CTRL/C |
| 4 | 04 | EOT | CTRL/D |
| 5 | 05 | ENQ | CTRL/E |
| 6 | 06 | ACK | CTRL/F |
| 7 | 07 | BEL | CTRL/G |
| 8 | 08 | BS | CTRL/H, BACKSPACE |
| 9 | 09 | HT | CTRL/I. TAB |
| 10 | OA | LF | CTRL/J, ENTER |
| 11 | 0B | VT | CTRL/K |
| 12 | 0C | FF | CTRL/L |
| 13 | 0D | CR | CTRL/M, RETURN |
| 14 | OE | SO | CTRL/N |

| Десятичное значение | Шестнадцатеричное значение | ASCII | Описание |
|---|---|---|---|
| 15 | OF | SI | CTRL/O |
| 16 | 10 | DLE | CTRL/P |
| 17 | 11 | DC1 | CTRL/Q |
| 18 | 12 | DC2 | CTRL/R |
| 19 | 13 | DC3 | CTRL/S |
| 20 | 14 | DC4 | CTRL/T |
| 21 | 15 | NAK | CTRL/U |
| 22 | 16 | SYN | CTRL/V |
| 23 | 17 | ETB | CTRL/W |
| 24 | 18 | CAN | CTRL/X |
| 25 | 19 | EM | CTRL/Y |
| 26 | 1A | SUB | CTRL/Z |
| 27 | 1B | ESC | ESC, ESCAPE |
| 28 | 1C | FS | CTRL< |
| 29 | 1D | GS | CTR/ |
| 30 | 1E | RS | CTRL/= |
| 31 | 1F | US | CTRL/- |
| 32 | 20 | SP | ПРОБЕЛ |
| 33 | 21 | ! | ! |
| 34 | 22 | " | " |
| 35 | 23 | # | # |
| 36 | 24 | $ | $ |
| 37 | 25 | % | % |
| 38 | 26 | & | & |
| 39 | 27 | ' | ' |
| 40 | 28 | ( | ( |
| 41 | 29 | ) | ) |
| 42 | 2A | * | * |
| 43 | 2B | + | + |
| 44 | 2C | , | , |

| Десятичное значение | Шестнадцатеричное значение | ASCII | Описание |
|---|---|---|---|
| 45 | 2D | - | - |
| 46 | 2E | . | . |
| 47 | 2F | / | / |
| 48 | 30 | 0 | 0 |
| 49 | 31 | 1 | 1 |
| 50 | 32 | 2 | 2 |
| 51 | 33 | 3 | 3 |
| 52 | 34 | 4 | 4 |
| 53 | 35 | 5 | 5 |
| 54 | 36 | 6 | 6 |
| 55 | 37 | 7 | 7 |
| 56 | 38 | 8 | 8 |
| 57 | 39 | 9 | 9 |
| 58 | 3A | : | : |
| 59 | 3B | ; | ; |
| 60 | 3C | < | < |
| 61 | 3D | = | = |
| 62 | 3E | > | > |
| 63 | 3F | ? | ? |
| 64 | 40 | @ | @ |
| 65 | 41 | A | A |
| 66 | 42 | B | B |
| 67 | 43 | C | C |
| 68 | 44 | D | D |
| 69 | 45 | E | E |
| 70 | 46 | F | F |
| 71 | 47 | G | G |
| 72 | 48 | H | H |
| 73 | 49 | I | I |
| 74 | 4A | J | J |

| Десятичное значение | Шестнадцатеричное значение | ASCII | Описание |
|---|---|---|---|
| 75 | 4B | K | K |
| 76 | 4C | L | L |
| 77 | 4D | M | M |
| 78 | 4E | N | N |
| 79 | 4F | O | O |
| 80 | 50 | P | P |
| 81 | 51 | Q | Q |
| 82 | 52 | R | R |
| 83 | 53 | S | S |
| 84 | 54 | T | T |
| 85 | 55 | U | U |
| 86 | 56 | V | V |
| 87 | 57 | W | W |
| 88 | 58 | X | X |
| 89 | 59 | Y | Y |
| 90 | 5A | Z | Z |
| 91 | 5B | [ | [ |
| 92 | 5C | \ | \ |
| 93 | 5D | ] | ] |
| 94 | 5E | ^ | ^ |
| 95 | 5F | _ | _ |
| 96 | 60 | ` | ` |
| 97 | 61 | a | a |
| 98 | 62 | b | b |
| 99 | 63 | c | c |
| 100 | 64 | d | d |
| 101 | 65 | e | e |
| 102 | 66 | f | f |
| 103 | 67 | g | g |
| 104 | 68 | h | h |
| 105 | 69 | i | i |

| Десятичное значение | Шестнадцатеричное значение | ASCII | Описание |
|---|---|---|---|
| 106 | 6A | j | j |
| 107 | 6B | k | k |
| 108 | 6C | l | l |
| 109 | 6D | m | m |
| 110 | 6E | n | n |
| 111 | 6F | o | o |
| 112 | 70 | P | p |
| 113 | 71 | q | q |
| 114 | 72 | r | r |
| 115 | 73 | S | S |
| 116 | 74 | t | t |
| 117 | 75 | U | u |
| 118 | 76 | V | V |
| 119 | 77 | w | W |
| 120 | 78 | X | X |
| 121 | 79 | y | y |
| 122 | 7A | z | Z |
| 123 | 7B | { | { |
| 124 | 7C | \| | \| |
| 125 | 7D | } | } |
| 126 | 7E | ~ | ~ |
| 127 | 7F | DEL | DELETE |

Приложение

# Специальные символы HTML

Эти символы имеют мало общего с Perl или CGI, но так как большая часть приложений CGI генерирует HTML, этот список может пригодиться. Он охватывает набор символов ISO8859-1.

| Символ | Численный код | Описательный код |
|---|---|---|
| " | " | " |
| & | & | & |
| < | < | < |
| > | > | > |
| |   |   |
| ¡ | ¡ | ¡ |
| ¢ | ¢ | ¢ |
| £ | £ | £ |
| ¤ | ¤ | ¤ |
| ¥ | ¥ | ¥ |
| ¦ | ¦ | ¦ |
| § | § | § |

| Символ | Численный код | Описательный код |
|---|---|---|
| ¨ | ¨ | ¨ |
| © | © | © |
| ª | ª | ª |
| « | « | « |
| ¬ | ¬ | ¬ |
| - |  |  |
| ® | ® | ® |
| ¯ | ¯ | ¯ |
| ° | ° | ° |
| ± | ± | ± |
| ² | ² | ² |
| ³ | ³ | ³ |
| ´ | ´ | ´ |
| µ | «181; | µ |
| ¶ | ¶ | ¶ |
| · | · | · |
| ¸ | ¸ | ¸ |
| ¹ | ¹ | ¹ |
| º | º | º |
| » | » | » |
| ¼ | ¼ | ¼ |
| ½ | ½ | ½ |
| ¾ | ¾ | ¾ |
| ¿ | «191; | ¿ |
| À | À | À |
| Á | Á | Á |
| Â | Â | Â |
| Ã | Ã | Ã |
| Ä | Ä | Ä |
| Å | Å | Å |
| Æ | Æ | Æ |

| Символ | Численный код | Описательный код |
|---|---|---|
| Ç | Ç | Ç |
| È | È | &Egave; |
| É | É | É |
| Ê | Ê | Ê |
| Ë | Ë | Ë |
| Ì | Ì | Ì |
| Í | Í | Í |
| Î | Î | Î |
| Ï | Ï | Ï |
| D | Ð | Ð |
| Ñ | Ñ | Ñ |
| 6 | Ò | Ò |
| 6 | Ó | Ó |
| B | Ô | Ô |
| Õ | Õ | Õ |
| Ö | Ö | Ö |
| X | × | × |
| 0 | Ø | Ø |
| O | Ù | Ù |
| 0 | Ú | Ú |
| 0 | Û | Û |
| Ü | Ü | Ü |
| Ý | Ý | Ý |
| P | Þ | Þ |
| ß | ß | ß |
| a | à | à |
| 6 | á | á |
| a | â | &adrc; |
| a | ã | ã |
| ä | ä | ä |
| å | å | å |

| Символ | Численный код | Описательный код |
|---|---|---|
| æ | æ | æ |
| ç | ç | ç |
| è | è | è |
| é | é | é |
| ê | ê | ê |
| ë | ë | ë |
| ì | ì | ì |
| í | í | í |
| î | î | î |
| ï | ï | ï |
| ð | ð | ð |
| ñ | ñ | ñ |
| ò | ò | ò |
| ó | ó | ó |
| ô | ô | ô |
| õ | õ | õ |
| ö | ö | ö |
| ÷ | ÷ | ÷ |
| ø | ø | ø |
| ù | ù | ù |
| ú | ú | ú |
| û | û | û |
| ü | ü | ü |
| ý | ý | ý |
| þ | þ | Whom; |
| ÿ | ÿ | ÿ |


# ИСТОЧНИКИ

## Рекомендуемая литература

Christiansen T. et.al. *Perl Cookbook*. O'Reilly & Associates. ISBN 1565922433 (1998) (Есть издание на русском языке: Кристиансен Т. *Perl: Библиотека программиста*. — СПб. и др.: Питер Ком, 2000. ISBN 5-8046-0094-х.)

Conway D. *Object Oriented Peri* Manning Publications. ISBN 1884777791 (1999)

Hall J. N. *Effective Perl Programming*. Addison-Wesley. ISBN 0201419750 (1997)

Schwartz R. et al. *Learning Perl (2nd Edition)*. O'Reilly & Associates. ISBN 1565922840 (1997) (На русском языке: Шварц Р.Л. *Изучаем Perl. Программирование в среде Unix*. — Киев: BHV, 1999. ISBN 5-7315-0031-2.)

Stein L. et. al. *Writing Apache Modules with. Perl and C: The Apache API and mod_perl*. O'Reilly & Associates. ISBN 156592567X (1999)

Stein L. *Official Guide to Programming with CGI.pm*. John Wiley & Sons. ISBN 0471247448 (1998)

Stein L. *Web Security: A Step-by-Step Guide*. Addison-Wesley. ISBN 0201634899 (1997)

*The Perl Journal*, ежеквартальный журнал, посвященный Perl. http://www.tpj.com

Wall L. et. al. *Programming Perl (3rd Edition)*. O'Reilly & Associates. ISBN 0596000278 (2000)

## Web-сайты

- Perl Home Page
  `http://www.perl.com`
- Perl Paraphernalia
  `http://www.plover.com/~mjd/perl`

- PerlFaq
  http://www.perlfaq.com
- Perldoc.com
  http://www.perldoc.com
- Perl Monks
  http://www.perlmonks.org
- use Perl;
  http://use.perl.org
- Dr. Dobb's Journal (online)
  http://www.ddj.com/topics/perl/
- Randal Schwartz's Web Techniques Columns
  http://www.stonehenge.com/merlyn/WebTechniques/
- Apache Project
  http://www.apache.org
- ApacheWeek
  http://www.apacheweek.com
- Perl в России
  http://www.perl.org.ru/
- PERL.RU
  http://www.perl.ru/
- PerLеводы
  http://perldoc.narod.ru/
- RuPerl — Все о perl на русском
  http://perl.lgg.ru/
- Почти все о языке Perl
  http://www.iif.dn.ua

# Предметный указатель

## Т



## У



## Ф



## Э

*Научно-популярное издание*

**Кевин Мельтцер, Брент** Михальски

# Разработка **CGI-приложений** на Perl

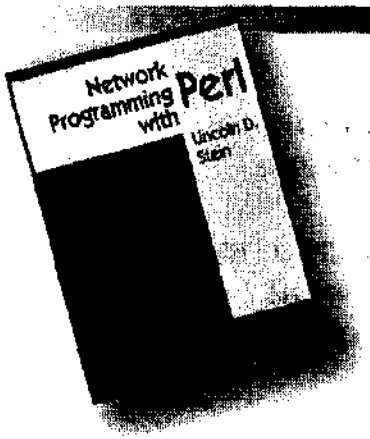
Литературный редактор *Т. Т. Шматко*
Верстка *К. В. Самоцветов*
Художественный редактор *С.А. Чернокозинский*
Технический редактор *Г.Н. Горобец*
Корректоры *ТА. Корзун, Л.В. Коровкина*

Издательский дом **"Вильямс".**
**101509,** Москва, ул. Лесная, д. 43, стр. 1.
Изд. лиц. ЛР JS6 090230 от 23.06.99
Госкомитета РФ по печати.

Подписано в печать 09.10.2001. Формат **70×100/16.**
Гарнитура Times. Печать офсетная.
Усл. печ. л. 32,25. Уч.-изд. л. **22,7.**
Тираж **5000** экз. Заказ **№** 1362.

Отпечатано с диапозитивов в ФГУП "Печатный **двор"**
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

Линкольн Д. Штайн




# Разработка сетевых программ на Perl

**Плановая дата выхода**
**3 кв. 2001 г.**

В этой книге рассматривается разработка и реализация применимых на практике сетевых приложений с использованием объектно-ориентированных средств языка программирования Perl. Приведены все сведения, необходимые для первого знакомства с объектами Perl.

Книга состоит из четырех частей.

В части I представлены основные сведения по сетевым средствам связи протокола TCP/IP. В частности описаны функции и переменные, применяемые для ввода - вывода, рассмотрены исключительные ситуации, которые возникают во время выполнения операций ввода - вывода, и приведено вводное описание сокетов на примере дескрипторов файлов, применяемых для ввода-вывода по каналу.

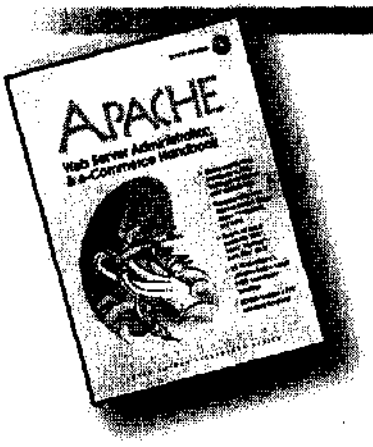
В части II рассматривается коллекция лучших модулей независимых разработчиков, внесенных в Полный сетевой архив Perl (CPAN). В ней описаны модули, обеспечивающие применение службы совместного доступа к файлам по протоколу FTP, а также представлен гибкий и удобный модуль Net::Telnet, который позволяет создавать клиенты для работы с самыми различными сетевыми службами. Показано, как динамически создавать сообщения электронной почты, в том числе двоичные файловые вложения, и отправлять их по месту назначения.

В части III описаны возможности проектирования систем типа клиент/сервер на основе протокола TCP. Рассмотрен TCP-сервер обычного типа, который порождает собственную копию для обработки каждого входящего соединения, описаны демоны inetd UNIX и Windows, позволяющие использовать в качестве сетевых серверов программы, которые специально не предназначались для этой цели.

В части IV рассматриваются методы создания специализированных приложений. Отдельная глава посвящена срочным или "внеочередным" данным TCP. Этот метод передачи данных часто применяется в интерактивных приложениях для отправки сигналов управления на удаленный сервер. Кроме того, в этой части представлен протокол пользовательских дейтаграмм UDP, который позволяет легко создать службу связи, ориентированную на передачу сообщений. В качестве примера рассматривается система интерактивной переписки и передачи сообщений, написанная полностью на языке Perl.

Книга предназначена для программистов Perl начального и среднего уровня.

Скотт Хокинг




# Администрирование Web-сервера Apache и руководство по электронной коммерции

**Плановая дата выхода**
**4-й кв. 2001 г.**

Данная книга в первую очередь ориентирована на разработчиков и администраторов Internet-серверов, функционирующих в самых различных системах. Написанная профессиональным разработчиком, книга содержит всю информацию, позволяющую справиться практически с любой реальной задачей, возникающей при разработке, сопровождении и обслуживании Web-серверов Apache. Подробно рассматриваются основы работы применяемого Web-сервера Apache, получившего широкое распространение прежде всего благодаря использовании идеологии открытого исходного кода. В книге приведены описания процедур инсталляции, конфигурирования, обслуживания как для тех, кто лишь только начинает работать с этим Web-сервером, так и для тех, кто уже обладает определенным опытом и нуждается в конкретном совете. В приложение включен справочник по директивам конфигурационных файлов Web-сервера Apache; кроме того, рассматриваются освещение концепции TCP/IP и разрешение проблем, иногда возникающих при эксплуатации Web-сервера Apache. Книга изобилует множеством примеров, помогающих глубоко усвоить излагаемый в ней материал.
Другими словами, книга предназначена для системных администраторов независимо от того, работают они в больших или маленьких компаниях, провайдерах услуг Internet. Прилагаемый к книге CD-ROM содержит дистрибутивы Web-серверов для работы под управлением различных операционных систем и исходные тексты. Это существенно ускоряет процесс изучения этой книги.

Книга предназначена для профессиональных разработчиков Internet-серверов.

Дж. Глени Брукшир


# Введение в компьютерные науки 6-е издание

**В продаже**

Эта книга представляет собой вводный или базовый курс по компьютерным наукам, который уже много лет читается в университетах США и других стран. Испытание временем, которое она успешно выдержала, свидетельствует о широте охвата и качестве изложения представленного в ней мате-риала. Несмотря на бурные темпы развития этой области знаний, автор данного курса постоянно поддерживает его актуальность, с каждым новым изданием обновляя излагаемый материал и попол-няя перечень освещаемых в нем аспектов компью-терных наук.

Содержание книги охватывает все основные суще-ствующие на сегодняшний день направления в об-ласти компьютерных наук. Каждая из глав посвя-щена обсуждению одного из таких направлений, причем изложение материала ведется в расчете на читателя, не имеющего подготовки в соответсвую-щей области. Однако уровень изложения и глубину охвата темы никак нельзя считать дилетантскими. Изучение всего предлагаемого материала позволит приобрести фундаментальные знания по всем ас-пектам компьютерных наук. Для будущих специа-листов в области компьютерных технологий они образуют надежную основу для дальнейшего обуче-ния, а у студентов иных профилей полученные зна-ния составят ядро компьютерной грамотности, так необходимой в настоящее время каждому ученому и инженеру. Это ядро впоследствии легко можно бу-дет дополнить сведениями о конкретных приложе-ниях и технологиях, используемых в той или иной области знаний

Благодаря полноте и доходчивости изложения ма-териала, не требующего никакой специальной под-готовки, эта книга может быть полезна всем — как будущим профессионалам в области вычислитель-ной техники, так и самому широкому кругу иных специалистов, нуждающихся в приобретении основ компьютерной грамотности.


Альфред В. Ахо, Рави Сети,
Джеффри Д. Ульман

# Компиляторы: принципы, технологии и инструментарий

**В продаже**

Каждый, кто когда-либо интересовался разработкой компиляторов, несомненно, сталкивался с знаменитой "Книгой Дракона" - "Dragon Book", классическим трудом Ахо и Ульмана "Принципы разработки компиляторов". Бурное развитие технологий компиляции привело к рождению нового дракона — книги "Компиляторы: Принципы, технологии, инструментарий" Альфреда Ахо, Рави Сети и Джеффри Ульмана,

Новая книга начинается с изложения принципов со-здания компиляторов, проиллюстрированных раз-работкой простейшего однопроходного компилятора. Оставшаяся часть книги посвящена развитию базовых идей и более прогрессивным и современ-ным технологиям, затрагивая, в частности, такие вопросы как синтаксический анализ, проверку ти-пов, генерацию и оптимизацию кода.

Строгость изложения материала смягчается боль-шим количеством практических примеров. Принци-пы и технологии написания компиляторов столь распространены, что идеи, которые вы найдете в этой книге, часто используются в области информа-ционных технологий. Написание компиляторов ох-ватывает языки программирования, архитектуру вычислительных систем, теорию языков, алгоритмы и технологию создания программного обеспечения. На заре компьютерной эры о компиляторах ходила слава как о программах, крайне сложных в написа-нии. С тех пор открыты и разработаны разнообраз-ные технологические приемы для решения многих важных задач, возникающих при компиляции. Кроме того, разработаны программные среды, про-граммный инструментарий и хорошая реализация многих языков программирования. Благодаря всем этим достижениям солидные компиляторы могут быть реализованы даже в качестве студенческой курсовой работы по проектированию компилято-ров. Помочь в освоении этих технологий и инстру-ментария и призвана данная книга. Однако, несмо-тря на свою "учебную" ориентацию, книга будет полезна всем, кому приходится работать над созда-нием компиляторов или просто интересуется дан-ной тематикой — от начинающих программистов до профессионалов