Perl for System Administration

David N. Blank-Edelman

O'REILLY®

Perl для системного администрирования

Дэвид Н. Бланк-Эдельман

Санкт-Петербург
2001

Дэвид Н. Бланк-Эдельман

Perl для системного администрирования

Перевод Т. Морозовой

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*В. Коновалов*
Редакторы	*Л. Ага, В. Овчинников*
Корректура	*С. Беляева, С. Журавина*
Верстка	*Н. Гриценко*

Дэвид Н. Бланк-Эдельман
Perl для системного администрирования. – Пер. с англ. – СПб: Символ-Плюс, 2001. – 496 с., ил.
ISBN 5-93286-024-3

Эта книга будет полезна администраторам с различным уровнем опыта – от обычных пользователей Linux до администраторов крупных систем. Автор рассматривает основные платформы, включая Unix, Windows NT/2000 и MacOS. При наличии некоторого опыта программирования на Perl вы узнаете, как при помощи этого языка повысить производительность во многих областях, включая: управление учетными записями пользователей; наблюдение за файловой системой и отслеживание процессов; работу с сетевыми службами имен NIS и DNS; администрирование баз данных при помощи DBI и ODBC; работу со службами каталогов LDAP и ADSI; обработку и анализ файлов журналов регистрации; поддержку защищенной сети; наблюдение за удаленными устройствами средствами SNMP.

Автор – опытный системный администратор, работающий в многоплатформенном окружении, что предоставляет вам хорошую возможность поучиться на чужом опыте. Вы узнаете о возможных ловушках и способах их обойти при помощи Perl. Включенные в книгу примеры и сценарии можно использовать для решения рутинных повседневных задач.

ISBN 5-93286-024-3
ISBN 0-56592-609-9 (англ)

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

Подписано в печать 06.09.2001. Формат 70x100 1/16 . Бумага офсетная.
Печать офсетная. Объем 31 печ. л. Тираж 3000 экз. Заказ N
Отпечатано с диапозитивов в ФГУП «Печатный Двор» им. А. М. Горького
Министерства РФ по делам печати, телерадиовещания
и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

Оглавление

Об авторе

Дэвид Н. Бланк-Эдельман – директор по технологиям факультета вычислительной техники Северо-восточного Университета (Northeastern University College of Computer Science). Последние 14 лет он работал системным и сетевым администратором на больших многоплатформенных системах в Brandies University, в технологической группе Кэмбриджа (Cambridge Technology Group) и лаборатории MIT. Кроме того, он является главным техническим редактором журнала «The Perl Journal» и написал немало статей о музыке разных стран. В свободное время он учится играть на мбире – традиционном музыкальном инструменте Зимбабве.

Предисловие

Perl – это мощный язык программирования, уходящий корнями в задачи традиционного системного администрирования. В течение многих лет он адаптировался и расширялся для работы с новыми операционными системами и новыми задачами. До сих пор, однако, не было ни одной книги, посвященной использованию Perl исключительно для системного администрирования, тем самым признавая факт глубокой исторической связи.

Если вы уже немного знакомы с Perl и вам необходимо выполнять задачи системного администрирования, то эта книга для вас. Читатели с различным уровнем опыта как в Perl, так и в системном администрировании, найдут в этой книге что-то для себя полезное.

Структура книги

Каждая глава этой книги посвящена определенному кругу задач системного администрирования. В конце каждой главы приведен список используемых в ней модулей и ссылки на более подробную информацию по рассмотренной теме. В книгу включены следующие главы:

Глава 1 «Введение»

Во введении описан материал, детально рассмотренный далее в книге, рассказано, как он может послужить и что нужно для того, чтобы получить от него максимальную пользу. В этой книге рассмотрен серьезный материал и считается, что он будет использован «влиятельными» пользователями (например, привилегированными пользователями в Unix или администраторами Windows NT/2000). Кроме того, введение содержит важные советы, помогающие писать более безопасные программы на Perl.

Глава 2 «Файловые системы»

Эта глава о том, как содержать в порядке и правильно использовать различные многоплатформенные файловые системы. Мы начнем с рассмотрения принципиальных различий между файловыми системами каждой из операционных систем. Затем мы рассмотрим процесс исследования файловой системы средствами Perl и расскажем, когда это может оказаться полезным. И, наконец, мы узнаем, как работать с дисковыми квотами из Perl.

Глава 3 «Учетные записи пользователей»

В этой главе рассказано о том, что собой представляют учетные записи на двух различных операционных системах. Основная часть главы – простейшая система ведения учетных записей, написанная на Perl. В процессе построения этой системы мы рассмотрим механизмы, необходимые для записи учетных записей в простую базу данных, основанную на XML, а также механизмы для создания учетных записей и их удаления.

Глава 4 «Действия пользователей»

В главе 4 рассмотрены различные механизмы управления процессами для всех трех операционных систем от самых простых (например, процессов в MacOS) до более сложных (в WinNT/2000). Мы организуем работу этих механизмов с помощью вспомогательных сценариев. И, наконец, мы узнаем, как средствами Perl проследить за операциями с файлами и за сетью.

Глава 5 «Службы имен TCP/IP»

Службы имен позволяют узлам в сети TCP/IP общаться друг с другом. В этой главе отражена вся история развития, начиная с файлов узлов, продолжая сетевой информационной службой (NIS) и заканчивая связующим звеном в Интернете – доменной службой имен (DNS). Для каждого шага такого пути мы покажем, как можно упростить профессиональную работу с этими службами при помощи Perl.

Глава 6 «Службы каталогов»

По мере роста сложности рассматриваемой информации растет и важность служб каталогов, которые мы используем для доступа к этой информации. Хорошо, если системные администраторы будут не просто использовать эти службы, но и создавать собственные инструменты для работы с ними. В этой главе рассказано о некоторых из наиболее популярных служб, таких как LDAP и ADSI, а также показано, как с ними работать при помощи Perl.

Глава 7 «Администрирование баз данных SQL»

Очень часто в сферу действий системного администратора попадает и работа с реляционными базами данных. А значит, системные администраторы должны быть знакомы с администрированием баз данных SQL. В этой главе рассмотрены два механизма для работы с базами данных – DBI и ODBC, а также приведены примеры их использования.

Глава 8 «Электронная почта»

В этой главе показано, как с помощью Perl лучше использовать электронную почту в качестве инструмента системного администрирования. После обсуждения основ отправки и разбора электронной почты с помощью Perl мы рассмотрим несколько интересных

приложений, включая анализатор спама и средство обработки электронной почты в службу технической поддержки.

Глава 9 «Журналы»

Системные администраторы зачастую просто утопают в море файлов журналов. Каждая машина, операционная система и программа может вести (и часто это делает) журналы. В этой главе рассмотрены системы ведения журналов в Unix и в WinNT/2000. Мы обсудим подходы к анализу такой информации с целью заставить ее работать на вас.

Глава 10 «Безопасность и наблюдение за сетью»

В последней главе мы обсудим вопросы, связанные с безопасностью. Мы покажем, как Perl может помочь сделать сеть и отдельные узлы в ней более защищенными. Кроме того, мы расскажем о некоторых технологиях контроля, включая использование протокола SNMP (простой протокол управления сетью) и «прослушивание» сети.

Приложения

В некоторых главах предполагается, что у вас уже есть знания по определенным темам, в то время как этого может и не быть. Для тех, кто не знаком с отдельными темами этой книги, есть несколько мини-руководств, которые помогут быстро разобраться в их основах. В число этих руководств входят введение в систему контроля версий (RCS), введение в протокол LDAP (облегченный протокол доступа к каталогам), введение в SQL, XML и протокол SNMP.

Условные обозначения

Курсив

Предназначен для выделения имен файлов, пользователей, каталогов, команд, узлов и адресов (URL), а также терминов, встречающихся в первый раз.

`Моноширинный шрифт`

Используется для выделения имен модулей и функций Perl, а также примеров исходного кода и результатов выполнения команды.

`Моноширинный полужирный шрифт`

Применяется для выделения пользовательского ввода в примерах.

`Моноширинный полужирный курсив`

Служит для выделения части командной строки, которая может быть изменена пользователем, а также для примечаний к программному коду.

Как с нами связаться

Мы тестировали и проверяли всю информацию, приведенную в книге, но вы можете обнаружить, что некоторые возможности изменились (или мы допустили кое-какие ошибки). Пожалуйста, сообщайте нам об ошибках, которые вы найдете, а также присылайте предложения относительно будущих изданий по следующему адресу:

O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472
(800) 998-9938 (в США и Канаде)
(707) 829-0515 (международный/местный)
(707) 829-0104 (факс)

Кроме того, вы можете прислать нам письмо электронной почтой. Чтобы подписаться на наш список рассылки или запросить каталог, пошлите письмо по адресу:

info@oreilly.com

Чтобы задать технические вопросы или прислать комментарий к книге, пишите по адресу:

bookquestions@oreilly.com

Кроме того, у нас есть веб-сайт, посвященный специально этой книге. Там приведены примеры, список найденных ошибок и рассказано о плане следующих изданий. Эта информация доступна по адресу:

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

Подробную информацию об этой и других книгах вы найдете на вебсайте издательства O'Reilly:

http://www.oreilly.com

Благодарности

Работа над книгой оказалась жребием, очень напоминающим строительство классической арки. Она также начиналась с двух колонн в моей жизни, которые сошлись вместе, – технической и личной.

С технической «точки зрения» я глубоко признателен Ларри Уоллу, который не только создал Perl, но и вдохнул в него (да и во все сообщество Perl-программистов) особый дух. Я благодарен великим учителям – Тому Кристиансену и Рэндаллу Шварцу, которые помогли мне и бесчисленному количеству людей разобраться со всеми тонкостями языка. Выше по этой колонне стоят программисты, которые потратили уйму времени и энергии на работу с языком, а затем решили поделиться со мной и со всем остальным сообществом Perl-программистов результатами своей работы. При любой возможности я стараюсь упомя-

нуть этих людей в книге, но моя благодарность относится ко всем названным и не названным, кто обогатил культуру Perl своими усилиями.

Поднимаясь вверх по технической колонне, за разделом о Perl мы обнаруживаем раздел системного администрирования. Здесь мы находим еще одно сообщество людей, которые помогли мне, этой книге и всей компьютерной области в целом. Члены конференций Usenix, SAGE и LISA заслужили благодарность за развитие лучшего, что может предложить область системного администрирования. В особенности мне хочется сказать спасибо Реми Эварду за то влияние, которое он оказал на мое профессиональное и личное понимание этой области, являясь моим другом, учителем и примером для подражания. Именно он – тот системный администратор, каким я хотел бы стать.

Ближе к вершине профессиональной колонны находятся те, кто ответствен за создание этой книги. Для начала я хочу поблагодарить моих редакторов и других комментаторов, потративших бездну часов на изменение этого текста (перечисляю всех в алфавитном порядке): Джо Джонстона, Джерри Картера, Тома Лимонселли, Джона Монтгомери, Хриса Нандора, Майкла Пепплера, Майкла Стока, Натана Торкингтона, Элин Фриш и Тоби Эверетта. Вплоть до самого конца этой работы они продолжали учить меня тонкостям Perl. Я благодарен Рону Портеру за его иллюстрации, Ханне Дайер и Лори Леджун за прелестное животное на обложке и всем сотрудникам издательства O'Reilly, принявшим участие в подготовке книги. И, наконец, я едва достоин права благодарить Линду Муи – моего редактора, чье удивительное мастерство, точность и забота позволили мне «родить» эту книгу и вырастить ее в хорошем доме. Линда Муи просто восхитительна.

Так же как и арка не строится из одной колонны, так и эта книга появилась не без участия другой, более личной основы. Я должен поблагодарить всех людей из моего духовного сообщества «Хавурат Шалом» из Соммервилля, Массачусетс, за их постоянную поддержку в течение всего процесса. Они научили меня смыслу сообщества. Спасибо вам, *Мекор ХаШаим*, за эту книгу и благословение в жизни.

С духовной точки зрения я в долгу перед народом Шона из Зимбабве за их удивительную музыку *мбира*, которая позволяла мне сохранять здравомыслие все это время. В особенности спасибо тем, кто познакомил меня с этой музыкой, – либо учителей, либо учеников, игравших рядом со мной. Эрика Эйзим, Стюарт Кардунер, Тьют Чигамба, Вири Чигонга, Мусекива Чингодза, Форвард Квенда, Космас Магая, Наоми Моланд, Соломон Мурунгу, Пол Новитски и Нина Рубин сыграли свою роль в этом процессе.

Я благодарен моим друзьям Эвнеру, Элен и Филу Шапиро и Алексу Сковронеку за их одобрение. Особое спасибо Джону Орванту и Джоелу Сегелу, двум друзьям, чьи мудрые советы и поддержка предоставили мне возможность и смелость для борьбы. Большое спасибо факультету и сотрудникам Северо-восточного университета компьютерных наук.

Я особенно благодарен членам группы CCS Systems, которые предоставили мне пространство, время и терпение, необходимые для работы с этой книгой. Ларри Финкельштейн – декан колледжа компьютерных наук – также заслуживает особого упоминания. Я никогда не встречал человека вне системного администрирования, который лучше бы понимал системных администраторов, их нужды и всю эту область в целом. Он продолжает учить меня, особенно на примерах, что означает быть настоящим лидером.

Давайте вернемся к метафоре арки, поскольку мы уже почти на вершине. Здесь мы найдем всю мою семью. Я благодарен всем им. Основа моей семьи – Майра, Джейсон и Стивен Бланк – это те люди, чей характер и воспитание (и любовь) в течение многих лет позволили мне сейчас быть с вами. Мои благодарности – Шиммеру и Бендиру, нередко составлявшим мне приятную компанию в написании этой книги, засиживаясь за компьютером до раннего утра. Большое спасибо Кристену Портеру и Тому Доновану из TCM.

Если вы знаете, что такое арки, то, вероятно, заметили, что я упустил самое главное – ключевой камень. Это камень, который находится на самой вершине арки и удерживает всю конструкцию вместе. Синди Бланк-Эдельман была моим ключевым камнем во время работы над этой книгой. Если и есть кто-то, кто пожертвовал ради этой книги большим, чем я сам, то это именно она. Без ее любви, поддержки, заботы, юмора, руководства и вдохновения я не остался бы самим собой, не говоря уж о писании каких-либо книг.

Эта книга посвящается Синди – любви всей моей жизни.

- *Системное администрирование – это ремесло*
- *Чем может помочь Perl*
- *Эта книга покажет вам, как*
- *Что вам нужно*
- *Поиск и установка модулей*
- *Нелегко быть всемогущим*
- *Ссылки на подробную информацию*

1

Введение

Системное администрирование – это ремесло

В моем городе есть несколько троллейбусных линий. Однажды, когда я поехал по незнакомому маршруту, я попросил водителя предупредить меня, когда мы будем подъезжать к нужной мне улице. Но он ответил: «Извините, я не смогу. Я ведь еду только по проводам».

Вы никогда не услышите, чтобы хороший системный администратор описывал бы свою работу подобными словами. В системном и сетевом администрировании зачастую приходится решать, какие провода прокладывать, где и как их соединять и как за ними следить. А затем, в итоге, отказываться от старых решений и начинать все заново. Хороший системный администратор вряд ли делает что-либо по привычке, не задумываясь, особенно в многоплатформенном окружении, где вопросы категорично требуют быстрых ответов. Как и в любом другом ремесле, в системном администрировании существуют способы отвечать на такие вопросы лучше или хуже. Эта книга для всех, кто сталкивается с подобными проблемами, будь то профессиональный системный администратор или новичок. Я попытаюсь показать, какой помощи можно ждать от Perl.

Какой помощи ждать от Perl

В работе системного администратора используется любой и всякий язык программирования, если его применение приносит пользу. Так почему же в этой книге выбран именно Perl?

Ответ на этот вопрос слышится в самой природе системного администрирования. Реми Эвард, мой друг и коллега, однажды описал работу системного администратора такими словами: «С одной стороны, у тебя есть набор ресурсов: компьютеры, сети, программное обеспечение и т. д. А с другой стороны, есть пользователи со своими нуждами и проектами – люди, которые хотят, чтобы их работа выполнялась. Наша задача заключается в том, чтобы состыковать эти два множества оптимальным способом, являясь при необходимости посредником между кругом расплывчатых нужд людей и техническим миром».

Системное администрирование – это зачастую *склеивание*; Perl – один из наиболее подходящих для этого языков. Perl использовался для системного администрирования задолго до того, как появился WWW со всеми своими требованиями к механизмам склеивания.

В Perl есть несколько других особенностей, свойственных принципам системного администрирования:

- Это очевидный потомок различных командных интерпретаторов Unix и языка C – то, чем многие системные администраторы уверенно владеют.
- Он доступен практически на всех современных операционных системах. И на каждой из них интерфейс остается практически одинаковым. Это очень важно для системных администраторов, работающих с несколькими платформами.
- В нем есть отличные инструменты для работы с текстом, по доступу к базам данных и программированию для сети – трем основам профессии.
- Основа языка может быть легко расширена благодаря тщательно продуманному механизму модулей.
- Многочисленное и преданное сообщество пользователей потратило несметное количество часов на создание модулей практически для каждой задачи. Как правило, эти модули тщательно организованы (мы вскоре к этому вернемся). Такая поддержка от сообщества может быть очень значительной.
- На Perl просто интересно программировать.

Для полноты картины следует заметить, что Perl не позволяет решить все мировые проблемы. Иногда он даже не подходит для программирования в области системного администрирования, потому что:

- Механизм объектно-ориентированного программирования в Perl несколько странен. В этом отношении Python гораздо лучше.
- Perl доступен не везде. На только что установленной системе вы скорее найдете командный интерпретатор Борна, а не Perl.
- Perl не всегда прост и последователен и он довольно запутан. В Tcl гораздо меньше сюрпризов.

- Perl достаточно непрост в использовании, чтобы заставить вас не раз наступить на грабли.

Мораль такова – *всегда выбирайте подходящий инструмент*. Таким инструментом для меня чаще всего бывал Perl, поэтому и появилась эта книга.

Эта книга покажет вам, как

В телевизионном шоу «Бэтмэн», популярном в 1966–1968 годах, у энергичной парочки были чудо-пояса с инструментами. Если Бэтмэну и Робину нужно было взобраться на здание, Бэтмэн говорил: «Быстрый Робин, абордажный крюк!». Или же Бэтмэн говорил: «Быстрый Робин, нокаутирующий газ!», и тут же нужный инструмент для борьбы с плохими парнями оказывался в руках у каждого. Цель этой книги – снабдить читателя таким поясом, необходимым для того, чтобы хорошо выполнять работу системного администратора.

В каждой главе вы найдете:

Понятную и четкую информацию о сфере действий системного администратора

В каждой главе мы подробно рассказываем об одной из сфер действий системного администрирования. Число возможных сфер действий при многоплатформенном системном администрировании слишком велико, чтобы рассказать обо всем в одной книге. Лучшие книги по системному администрированию в Unix – «Essential System Administration» (Суть системного администрирования) Элен Фриш (Æleen Frisch) (O'Reilly & Associates) и «Unix System Administration Handbook» (UNIX: руководство системного администратора) Эви Немет (Evi Nemeth), Гарта Снайдера (Garth Snyder) и Трента Р. Хейна (Trent R. Hein) (Prentice-Hall) – одна в два, а другая в три раза больше этой работы. Мы же рассмотрим вопросы, относящиеся к трем различным операционным системам: Unix, Windows NT/2000 и MacOS.

В результате, приходилось выбирать, что включить в книгу, а что отложить. Были исключены темы, которые, на мой взгляд, приобретут существенно большее значение только в последующие пять лет. А такие важные технологии, как XML, были рассмотрены, потому что в ближайшее время они, наверняка, окажут заметное влияние на всю эту область. К сожалению, этот подход привел к тому, что такие вечные вопросы системного администрирования, как резервное копирование и печать, были вытеснены более новыми темами, подобными LDAP и SNMP. Навыки и инструменты, представленные в этой книге, могут помочь в тех областях, которые я пропустил, но детальные описания нужно искать где-то в другом месте.

Я попытался собрать вместе достаточно информации о системном и сетевом администрировании для людей с различным уровнем опыта. Опытные профессионалы и новички могут почерпнуть из этой книги совершенно разную информацию, но каждый из них найдет для себя что-то интересное. В конце каждой главы приведен список источников информации, которые могут помочь глубже разобраться с выбранной темой.

Для каждой темы или области, особенно, если ее изучение требует известного времени, я включил приложения с информацией, необходимой для того, чтобы быстро во всем разобраться. Даже если тема вам знакома, такие приложения могут помочь дополнить свои знания (например, выяснить, как нечто реализовано на другой операционной системе).

Технологии и подходы Perl, которые можно использовать в системном администрировании

Чтобы извлечь из книги все возможное, необходимо владеть некоторыми основами Perl. В каждой главе достаточно много разных по сложности программ, иногда простых и доступных новичкам, а порой требующих довольно серьезного знания Perl. Если будет встречаться технология, структура данных или идиома среднего или повышенного уровня сложности, я потрачу время, чтобы аккуратно рассмотреть ее шаг за шагом. Рассмотрев некоторые интересные технологии программирования на Perl, вы сможете включить их в свою практику. Я надеюсь, что программисты на Perl любого уровня смогут чему-нибудь научиться на приведенных примерах. И, по мере роста вашего уровня, вы сможете вернуться к этой книге и научиться чему-нибудь еще.

Чтобы еще больше расширить такой опыт обучения, я буду часто приводить несколько способов решения одной и той же задачи при помощи Perl, а не единственный из возможных. Запомните девиз Perl: «Существует более одного способа сделать это». Все примеры, отражающие различные подходы, придуманы для того, чтобы лучше оснастить ваш пояс с инструментами: чем больше их будет у вас в руках, тем более правильный выбор вы сможете сделать, когда столкнетесь с новой проблемой.

Иногда кажется очевидным, что одна технология имеет преимущество перед остальными. Но в этой книге описаны лишь некоторые ситуации, с которыми вы можете столкнуться, а решения, достаточно неуклюжие для одной проблемы, могут оказаться разгадкой для другой. Так что наберитесь терпения. Для каждого примера я попытаюсь показать вам как преимущества, так и недостатки разных подходов (и часто буду говорить, какой метод предпочитаю я).

Принципы и лучшие приемы системного администрирования

Как я уже говорил в начале этой главы, существуют лучшие и худшие способы выполнять задачи системного администрирования. В течение последних 15 лет я в качестве системного и сетевого администратора управляю весьма требовательным многоплатформенным окружением. В каждой главе я попытаюсь передать свой опыт, рассказывая о лучших приемах, которым я научился, и глубоких принципах, лежащих в их основе. Иногда я буду ссылаться на «военные рассказы прямо с передовой» из собственного опыта, используя их в качестве отправной точки для обсуждения. Будем надеяться, по мере чтения вся глубина искусства системного администрирования станет для вас очевидной.

Что вам нужно

Чтобы понять большую часть книги, вам следует знать основы языка и иметь кое-какие ресурсы под рукой. Давайте начнем с перечисления тех знаний, которыми вы должны обладать:

Вы должны немного знать Perl

В этой книге недостаточно места, чтобы приводить основы языка Perl, поэтому перед дальнейшим чтением вы должны изучить их самостоятельно. Познакомившись с содержанием таких книг, как «Learning Perl»[1] (Изучаем Perl) Рэндала Шварца (Randal L. Schwartz) и Тома Кристиансена (Tom Christiansen) (O'Reilly) или «Learning Perl on Win32 Systems» (Изучаем Perl для Win32) Рэндала Шварца (Randal L. Schwartz), Эрика Олсона (Erik Olson) и Тома Кристиансена (Tom Christiansen) (O'Reilly), вы будете в хорошей форме, чтобы приступить к кодам из этой книги.

Вы должны знать основы вашей операционной системы (систем)

В этой книге предполагается, что у вас есть некоторый опыт работы с операционной системой (системами), которую вы собираетесь администрировать. Вы должны знать, как работать с этой операционной системой, как выполнять команды, искать документацию и т. д. Вы должны знать и основы более сложных технологий, существующих в операционной системе (например, WMI для Windows 2000 или SNMP).

Вам может понадобиться знание особенностей вашей операционной системы (систем)

Я предпринял попытку описать различия между основными операционными системами, но я не смог охватить все внутренние разли-

[1] Рэндал Л. Шварц, Том Кристиансен «Изучаем Perl», издательская группа BHV, Киев, 2000 г.

чия. В частности, каждый вариант Unix немного отличается от всех остальных. Таким образом, вам может понадобиться найти информацию об особенностях вашей операционной системы и разобраться, будет ли эта информация отличаться от описанной здесь.

Из технических ресурсов вам понадобятся только две вещи:

Perl

Вам нужна копия Perl, установленная или доступная для каждой системы, которую вы хотите администрировать. С веб-сайта *http://www.perl.com* вы можете загрузить дистрибутив либо с исходными кодами, либо в скомпилированном виде для конкретной операционной системы. В примерах этой книги применяется Perl 5.005 (в момент написания книги – это последняя стабильная версия[1]). В Unix мы используем дистрибутив Perl, скомпилированный из исходных кодов, на платформе Win32 – версию, распространяемую ActiveState (build 522), а на MacOS – дистрибутив MacPerl (5.2.0r4).

Возможность найти и установить модули Perl

Следующий раздел этой главы посвящен информации о местонахождении и установке модулей, поскольку эти данные чрезвычайно важны. Мы предполагаем, что у вас есть знания и необходимые права для установки всех нужных модулей.

В конце каждой главы приведен список номеров версий всех модулей, используемых в примерах этой главы. Информация о номерах версий приводится потому, что модули постоянно обновляются. Обновление не всегда сохраняет совместимость с предыдущими версиями, поэтому при возникновении трудностей эта информация поможет вам определить, изменялся ли модуль со времени издания книги.

Поиск и установка модулей

Основное преимущество использования Perl для системного администрирования заключается в доступности свободного исходного кода модулей. Рассмотренные в этой книге модули можно найти в одном из трех источников:

Полная сеть Perl-архивов (Comprehensive Perl Archive Network, CPAN)

CPAN – это огромный архив исходного кода на Perl, документации, сценариев и модулей, продублированных на сотне сайтов по всему миру. Эту информацию можно найти по адресу *http://www.cpan.org*. Самый простой способ найти модули на CPAN – воспользоваться поисковой системой, разработанной и поддерживаемой Элэни Аштоном (Elaine Ashton), Грэхемом Баром (Graham Barr) и Клифтоном

[1] На данный момент вышла версия 5.6.1. – *Примеч. науч. ред.*

Поси (Clifton Posey) на *http://search.cpan.org*. Поле «CPAN Search:» облегчает задачу поиска нужных модулей.

Репозитории для скомпилированных пакетов

Скоро мы познакомимся с менеджером пакетов Perl (Perl Package Manager, PPM), очень важным инструментом для пользователей Perl на Win32. Этот инструмент соединяется с репозиториями (самый известный из которых расположен на ActiveState), чтобы получить собранные пакеты модулей. Адреса репозиториев можно найти в списке часто задаваемых вопросов о PPM на *http://www.activestate.com/Products/ActivePerl/docs/faq/ActivePerl-faq2.html*. Если пакет для Win32 получен не с ActiveState, я обязательно укажу на это. Пакеты для MacOS лучше всего искать на сайте MacPerl Module Porters по адресу *http://pudge.net/mmp/*.

Отдельные веб-сайты

Некоторые модули не попадают на CPAN или в репозитории PPM. Я всегда буду говорить, где можно найти модуль, если он находится в необычном месте.

Как установить модуль после того, как вы его найдете? Ответ зависит от того, какую операционную систему вы используете. В дистрибутив Perl входит документация по установке, которую можно найти в файле *perlmodinstall.pod* (наберите *perldoc perlmodinstall*, если хотите ее прочитать). В следующем разделе я приведу краткое описание действий, которые необходимо выполнить в каждой из операционных систем, рассмотренных в этой книге.

Установка модулей в Unix

В большинстве случаев этот процесс выглядит так:

1. Скачайте модуль и распакуйте его.
2. Запустите *perl Makefile.PL*, чтобы создать необходимый *Makefile*.
3. Запустите *make*, чтобы собрать пакет.
4. Запустите *make test*, чтобы выполнить все тестовые действия, включенные в модуль автором.
5. Запустите *make install*, чтобы установить модуль в отведенное на вашей системе место.

Если вы не хотите возиться с установкой вручную, то можете использовать написанный Андреасом Кенигом (Andreas J. König) модуль `CPAN` (модуль входит в состав Perl). Этот модуль позволит выполнить все эти действия, набрав:

```
% perl -MCPAN -e shell
cpan> install modulename
```

Модуль CPAN достаточно «умен», чтобы обрабатывать зависимости (т. е. если один модуль требует запуска другого модуля, то CPAN установит оба модуля автоматически). В CPAN, кроме того, есть функция для поиска связанных модулей и пакетов. Я рекомендую набрать *perldoc CPAN*, чтобы ознакомиться со всеми полезными возможностями этого модуля.

Установка модулей на Win32

Процесс установки модулей на платформе Win32 совпадает с процессом установки модулей на Unix, но требует одного дополнительного шага – *ppm*. Если вы собираетесь устанавливать модули вручную, следуя инструкциям по Unix, то можете использовать программы, подобные WinZip (*http://www.winzip.com*), чтобы распаковать дистрибутив, и *nmake* (*ftp://ftp.microsoft.com/Softlib/MSLFILES/nmake15.exe*) вместо *make* – для сборки и установки модуля.

Некоторые модули в процессе сборки требуют компиляции исходных файлов на C. У многих пользователей Perl на Win32 нет нужного программного обеспечения для компиляции, поэтому в ActiveState создали менеджер пакетов Perl для работы с дистрибутивами скомпилированных модулей.

Система PPM похожа на модуль CPAN. Для загрузки и установки специальных файлов архивов из репозиториев PPM используется программа *ppm.pl*, написанная на Perl. Вы можете запустить ее, либо набрав *ppm*, либо выполнив команду *perl ppm.pl* из каталога *bin*:

```
C:\Perl\bin>perl ppm.pl
PPM interactive shell (1.1.1) - type 'help' for available commands.
PPM> install module-name
```

ppm, как и CPAN, может искать для вас список доступных и установленных модулей. Наберите *help* после приглашения *ppm*, чтобы получить информацию о том, как использовать эти команды.

Установка модулей на MacOS

Установка модулей на MacOS – это странный гибрид уже рассмотренных методов. Крис Нандор (Chris Nandor) собрал дистрибутив *cpan-mac* (его можно найти либо на CPAN, либо на *http://pudge.net/macperl*), в состав которого входит перенесенная на MacOS версия CPAN, а также невероятное количество других модулей.

После установки дистрибутива *cpan-mac* можно при помощи CPAN загружать и устанавливать большинство модулей, реализованных исключительно на языке Perl. Нандор упростил эту задачу, написав небольшое приложение *installme*. Архивные файлы (т. е. файлы *.tar.gz*), переданные *installme*, будут разархивированы и установлены в стиле CPAN.

Подробности об установке модулей на MacOS можно найти в расширенной версии документа *perlmodinstall.pod*, упомянутого ранее как *macperl modinstall.pod*. Его можно найти также и на *http://pudge.net/macperl*.

Нелегко быть всемогущим

Перед тем как продолжить, давайте отвлечемся на несколько минут и скажем пару предостерегающих слов. У программ, написанных для системного администрирования, есть одна характерная черта, которая отличает их от всех остальных программ. В Unix и Windows NT/2000 они зачастую выполняются с повышенными привилегиями, т. е. с правами пользователей *root* или *Administrator*. А такая сила подразумевает ответственность. На нас, как на программистов, ложится дополнительная ответственность писать безопасные программы. Мы пишем программы, которые могут обходить (и обходят) ограничения безопасности, накладываемые на «простых смертных». Если мы не будем осторожными, менее «нравственные» личности смогут использовать «дыры» нашего кода в низких целях. Вот несколько моментов, о которых вы всегда должны помнить, применяя Perl в этих обстоятельствах.

Не делайте этого

Конечно же, используйте Perl. Но, по возможности, старайтесь избегать выполнения программ в привилегированном окружении. Большинство задач не требует привилегий пользователя *root* или *Administrator*. Например, программа анализа журналов, вероятно, не должна выполняться с правами суперпользователя. Для выполнения этих автоматизированных действий создайте другого пользователя, наделенного меньшими привилегиями. Пусть у вас будет маленькая программа, наделенная привилегиями, которая будет передавать при необходимости данные этому пользователю, и затем этот пользователь будет выполнять анализ.

Избавьтесь от своих привилегий как можно быстрее

Иногда невозможно избежать необходимости запускать сценарий с правами привилегированного пользователя. Например, созданная вами программа доставки почты может потребовать возможности записывать в файл от имени любого пользователя в системе. Программы, подобные этой, должны отказываться от своего «всемогущества» как можно раньше во время своего выполнения.

В программах на Perl, выполняющихся в Unix или Linux, можно установить переменные $< и $>:

```
# Навсегда избавиться от привилегий
```

```
($<,$>) = (getpwnam('nobody'),getpwnam('nobody'));
```

В результате, реальный и эффективный идентификаторы пользователя будут установлены равными идентификатору пользователя *nobody*, не являющемуся привилегированным пользователем. Если вы хотите подойти к проблеме более основательно, то можете также использовать переменные `$(` и `$)` для смены реального и эффективного идентификатора группы.

В Windows NT и Windows 2000 вообще нет идентификаторов пользователей, но для избавления от привилегий существует схожий процесс. В Windows 2000 есть возможность, называемая «RunAs», которую можно использовать для запуска процесса от имени другого пользователя. В Windows NT и Windows 2000 пользователи с правами `Act as part of the operating system` могут выдавать себя за других пользователей. Эти права можно установить с помощью программы *User Manager* или *User Manager for Domains*:

1. В меню Policies выберите пункт User Rights.
2. Отметьте пункт Show Advanced User Rights.
3. Выберите Act as part of the operating system из выпадающего списка.
4. Выберите пункт Add... и определите пользователей или группы, которых вы хотите наделить этими правами. Если вы хотите предоставить такое право определенному пользователю, выберите пункт Show Users.
5. Вероятно, данному пользователю придется повторно зарегистрироваться в системе, чтобы эти изменения вступили в силу.

Вам также понадобится добавить права `Replace a process level token` и, в некоторых случаях, `Bypass traverse checking` (см. документацию по `Win32::AdminMisc`). Как только вы присвоили эти права пользователю, он сможет запускать сценарии на Perl с функцией `LogonAsUser()` из модуля Дэвида Рота (David Roth) `Win32::AdminMisc`, который можно найти на *http://www.roth.net*:

```
use Win32::AdminMisc;
die "Невозможно персонализировать $user\n"
if (!Win32::AdminMisc::LogonAsUser('',$user,$userpw);
```

Замечание: здесь существует некоторая опасность, поскольку в отличие от предыдущего примера, вы должны передать пароль пользователя в функцию `LogonAsUser()`.

Будьте осторожны при чтении данных

При чтении важных данных, скажем, конфигурационных файлов, сначала протестируйте возможность небезопасных состояний. Напри-

мер, стоит проверить, запрещена ли запись в файл и каталоги, в которых он находится (иначе, кто угодно может их испортить). Хороший способ подобной проверки можно найти в главе 8 книги «Perl Cookbook»[1] («Perl: библиотека программиста») Тома Кристиансена (Tom Christiansen) и Натана Торкингтона (Nathan Torkington) (O'Reilly).

Другая забота – ввод пользователей. Никогда не считайте, что данным, поступающим от пользователей, можно доверять. Даже если вы явно просите пользователя: `Пожалуйста, ответьте Да(Y) или Нет(N):`, ничто не помешает ему набрать 2049 случайных символов (либо от вредности со злым умыслом, либо потому, что его двухлетний ребенок занял освободившееся на минуту место за клавиатурой).

Ввод пользователей может быть причиной еще более серьезных проблем. Мой любимый пример – это использование нулевого байта «Poison NULL Byte», о котором сообщалось в статье о проблемах Perl в CGI. Обязательно прочитайте всю статью (ссылка на нее есть в конце этой главы). Неприятности возникают из-за отличий в обработке нулевого байта (`\000`) в Perl и в системных библиотеках C. Для Perl этот символ ничем не примечателен. Однако в библиотеках этот символ используется для обозначения конца строки.

На практике это означает, что у пользователя существует возможность обойти различные проверки. Один пример, приведенный в этой статье, – это программа, меняющая пароль пользователя:

```
if ($user ne "root"){ <вызов соответствующей функции C>}
```

Если переменная `$user` установлена в значение `root\000` (т. е. если за словом `root` следует нулевой байт), то приведенная выше проверка окажется удачной. Когда эта строка будет передана библиотеке, она будет воспринята просто как *root*, и пользователю удастся обойти проверку. Если эту ситуацию не отследить, то подобная «дыра» позволит получить доступ к произвольным файлам и другим ресурсам. Самый простой способ не пострадать от этой проблемы – подправить код, добавив что-то похожее на следующую строку:

```
$input =~ tr/\000//d;
```

Это всего лишь один пример того, как ввод пользователя может вызвать проблемы. Именно поэтому в Perl существует специальное средство предосторожности – режим пометки (taint mode). Изучите страницу руководства *perlsec*, входящую в состав дистрибутива Perl, чтобы ознакомиться с отличным объяснением того, что такое отмеченные данные, а также с другими мерами предосторожности.

---

[1] Том Кристиансен, Натан Торкингтон «Perl: библиотека программиста», издательство «Питер», 2000 г. – *Примеч. ред.*

Будьте осторожны при записи данных

Если ваша программа может записывать или дописывать данные в любой файл локальной файловой системы, вы должны особенно заботиться о том, как, куда и когда записываются данные. В системах Unix это особенно важно, поскольку символические ссылки очень сильно упрощают подмену файлов и перенаправление. Если ваша программа написана не очень аккуратно, может оказаться, что она пишет не в тот файл или устройство. Существует два класса программ, в которых это соображение особенно важно.

В первый класс попадают программы, дописывающие данные в существующие файлы. *Перед* дописыванием в файл в вашей программе должна быть выполнена следующая последовательность шагов:

1. Используйте функцию `stat()` и обычные операторы проверки файлов для проверки атрибутов файлов. Убедитесь, что файл не является ни жесткой, ни символической ссылкой, что у него установлены нужные права и владельцы и т. д.
2. Откройте файл для дописывания.
3. Передайте файловый дескриптор функции `stat()`.
4. Сравните значения, полученные на шагах 1 и 3, чтобы убедиться, что открытый файловый дескриптор соответствует нужному вам файлу.

Смотрите программу *bigbuffy* из главы 9 «Журналы», которая соблюдает эту последовательность шагов.

Во второй класс попадают программы, использующие временные файлы или каталоги. Вы часто видели подобный код:

```
open(TEMPFILE,">/tmp/temp.$$") or die "невозможно записать в /tmp/
temp.$$:$!\n";
```

К сожалению, это недостаточно безопасно для многопользовательских систем. Последовательность идентификаторов процессов (`$$`) на большинстве машин легко предсказуема, а это означает, что также предсказуемо имя следующего временного файла, который будет использовать ваш сценарий. Если кто-то сможет предсказать это имя, он сможет оказаться там раньше вас. А это уже, как правило, плохие новости.

В некоторых операционных системах есть библиотечные вызовы, которые генерируют имена временных файлов, используя современный алгоритм случайных значений. Чтобы проверить вашу операционную систему, вы можете запустить следующий код. Если получаемые имена кажутся вам достаточно случайными, вы можете полагаться на `POSIX::tmpnam()`. Если нет, вы можете написать собственную функцию генерации случайных имен файлов:

```
use POSIX qw(tmpnam);
```

```
for (1..20){ print POSIX::tmpnam(),"\n"; }
```

Как только у вас будет имя файла, которое нельзя отгадать, вам нужно будет открыть его безопасным образом:

```
sysopen(TEMPFILE,$tmpname,O_RDWR|O_CREAT|O_EXCL,0666);
```

Существует другой, более простой способ выполнить эти же два шага (получить имя и открыть временный файл). Метод `IO::File->new_tmpfile()` из модуля `IO::File` не только подберет хорошее имя (если системные библиотеки это поддерживают), но и откроет файл для чтения и записи.

Примеры использования `POSIX::tmpnam()` и `IO::File->new_tmpfile()`, а также другую информацию по этой теме вы можете найти в главе 7 книги рецептов «Perl Cookbook» («Perl: библиотека программиста»). В модуле `File::Temp` Тима Дженнеса (Tim Jenness) также предпринимаются попытки обеспечить безопасные операции работы с временными файлами.

Избегайте состояний перехвата

По мере возможности, старайтесь не писать программ, допускающих состояния перехвата. Обычное состояние перехвата начинается с предположения, что следующая последовательность допустима:

1. Ваша программа будет накапливать некоторые данные.
2. Ваша программа затем будет работать с этими данными.

Если пользователи могут проникнуть в эту последовательность на шаге, скажем 1,5, и выполнить некоторую замену данных, это может привести к неприятностям. Если им удается контролировать вашу программу на шаге 2, чтобы обработать данные, отличающиеся от тех, которые были на шаге 1, значит, им удалось использовать состояние перехвата (т. е., их программа выиграла состязание, чтобы получить данные). В другом случае состояние перехвата может произойти, если вы неверно работаете с блокировкой файлов.

Состояния перехвата часто возникают в программах системного администрирования, которые на первом шаге сканируют файловую систему, а на втором шаге изменяют данные. Бесчестные пользователи могут внести изменения в файловую систему сразу после сканирования, чтобы изменения были внесены в неверный файл. Убедитесь, что в вашем коде нет подобных «дыр».

Наслаждайтесь

Очень важно помнить, что системное администрирование интересно. Не всегда и не тогда, когда вам надо решать самые досаждающие проблемы, но определенное наслаждение в этом можно найти. Есть настоя-

щее удовольствие в том, чтобы поддерживать других людей и создавать инфраструктуру, которая улучшает жизнь всем. Когда созданные вами программы объединяют людей – это прекрасно.

Теперь, когда вы готовы, давайте поработаем над «теми самыми проводами».

Ссылки на подробную информацию

http://dwheeler.com/secure-programs/Secure-Programs-HOWTO.html – документ HOWTO (соображения «как сделать») о безопасном программировании в Linux, но рассмотренные в нем концепции и технологии применимы и в других ситуациях.

http://www.cs.ucdavis.edu/~bishop/secprog.html содержит лучшие способы безопасного программирования от эксперта по безопасности Мэтта Бишопа (Matt Bishop).

http://www.homeport.org/~adam/review.html – список указаний по написанию безопасного кода от Адама Шостака (Adam Shostack).

http://www.dnaco.net/~kragen/security-holes.html – хороший документ о том, как искать дыры в защите (особенно в собственном коде) от Крегена Ситэйкера (Kragen Sitaker).

http://www.shmoo.com/securecode/ предлагает отличную коллекцию статей о том, как писать безопасные программы.

«Perl CGI Problems», Rain Forest Puppy (Phrack Magazine, 1999). Электронный вариант можно найти на *http://www.insecure.org/news/P55-07.txt* или в архивах Phrack на *http://www.phrack.com/archive.html*.

«Perl Cookbook», Tom Christiansen, Nathan Torkington (O'Reilly, 1998) – эта книга рецептов содержит много хороших советов по созданию безопасных программ.

- *Perl приходит на помощь*
- *Различия файловых систем*
- *Прогулка по файловой системе*
- *Обход файловой системы при помощи модуля File::Find*
- *Работа с дисковыми квотами*
- *Получение сведений об использовании файловой системы*
- *Информация о модулях из этой главы*
- *Источники подробной информации*

2

Файловые системы

Perl приходит на помощь

Лэптопы падают медленно. По крайней мере, это выглядело именно так, когда компьютер, с помощью которого я писал эту книгу, упал со стола на твердый деревянный пол. Когда я его поднял, он по-прежнему был цел и работал. Но в процессе проверки лэптопа на предмет наличия повреждений я заметил, что он начал работать все медленнее и медленнее. Мало того, время от времени он стал устрашающе гудеть и жужжать во время обращения к диску. Решив, что замедление было вызвано программной ошибкой, я выключил компьютер. Но выключаться корректно он отказывался. Это был плохой знак.

Еще хуже было нежелание компьютера загружаться. Он начинал процесс загрузки Windows NT, но затем выдавал ошибку «файл не найден» и замирал. Теперь стало ясно, что падение вызвало серьезное физическое повреждение жесткого диска. Вероятно, головки проскользнули по поверхности диска, уничтожив файлы и каталоги, находившиеся в открытом состоянии. Передо мной встал вопрос: «Остались ли неповрежденными хоть какие-то файлы? Выжили ли файлы *этой книги*?»

Первым делом я попытался загрузить Linux, другую операционную систему, установленную на портативном компьютере. Linux загрузился нормально; это меня подбодрило. Однако файлы этой книги находились в разделе NTFS, который не загружался. Используя драйвер NTFS для Linux, написанный Мартином фон Левисом (Martin von Low-

is) и доступный на *http://www.informatik.hu-berlin.de/~loewis/ntfs/* (теперь он поставляется вместе с ядрами Linux 2.2), я смонтировал раздел и обрадовался, увидев, что все мои файлы *казались* неповрежденными.

Мои попытки скопировать файлы из этого раздела проходили успешно, но лишь до тех пор, пока я не достиг некоторого файла. Диск опять издал этот зловещий звук, и скопировать файл не удалось. Было ясно, что если я хочу спасти свои данные, мне нужно пропустить все поврежденные файлы на диске. Программа, которой я пользовался (*gnutar*), могла пропустить список файлов. Вопрос был только в том, какие файлы следовало пропускать? Когда возникла эта неприятность, в файловой системе было больше *шестнадцати тысяч* файлов. Как я мог разобраться, какие файлы были повреждены, а какие нет? Запускать *gnutar* снова и снова было отнюдь не лучшей стратегией. С такой задачей мог справиться Perl!

Позже в этой главе я покажу исходный код, которым пользовался для решения возникшей проблемы. Чтобы понять, как работает этот код, нам следует сначала познакомиться с файловыми системами вообще и, в частности, с тем, как мы работаем с ними в Perl.

Различия файловых систем

Начнем с краткого обзора файловых систем, свойственных каждой из рассматриваемых операционных систем. Возможно, это не представляет для вас ничего нового, особенно, если у вас есть значительный опыт работы с какой-либо операционной системой. Но все же стоит обратить внимание на различия между файловыми системами (особенно на те, которые вам не знакомы), если вы собираетесь писать программы на Perl, работающие на разных платформах.

Unix

Все современные разновидности Unix поставляются с файловыми системами, семантика которых напоминает семантику их общего предка – файловой системы Berkeley Fast File System. Различные производители по-разному расширяли свои файловые системы (так, в Solaris добавили списки контроля доступа (Access Control Lists) для большей безопасности, в Digital Unix стали применять файловую систему *advfs*, основанную на транзакциях, и т. д.). Мы будем писать код, «приведенный к общему знаменателю», что позволит ему работать на различных Unix-платформах.

Вершина, или корень файловой системы, в Unix обозначается символом прямого слэша (/). Для того чтобы уникальным образом идентифицировать файл или каталог в файловой системе, мы строим путь, начинающийся со слэша, и добавляем в него каталоги, разделяя их

слэшами, по мере прохода «вглубь» файловой системы. Последний компонент пути – нужный каталог или файл. В современных вариантах Unix имена каталогов и файлов чувствительны к регистру символов. При известном навыке в именах можно использовать практически все ASCII-символы, но если ограничиться буквенно-цифровыми символами и некоторыми знаками пунктуации, то в дальнейшем можно избежать лишних сложностей.

Microsoft Windows NT/2000

Windows NT (в данной книге речь идет о версии 4.0) поставляется с двумя поддерживаемыми файловыми системами: файловой системой FAT (таблица размещения файлов) и NTFS (файловая система NT). В Windows 2000 добавлена файловая система FAT32 – улучшенная версия FAT, позволяющая иметь разделы бо́льших размеров и кластеры меньших размеров.

В Windows NT используется расширенная версия файловой системы FAT из DOS. Перед тем как рассматривать расширенную версию, очень важно разобраться в недостатках исходной. В обычной файловой системе (FAT реального режима) имена файлов должны соответствовать формату 8.3. Это означает, что имена файлов или каталогов могут содержать не более восьми символов, за которыми должна следовать *точка,* а затем суффикс длиной не более трех символов. В отличие от Unix, где точка в имени файла не имеет специального назначения, в FAT можно использовать только одну точку в качестве разделителя между именем файла и его расширением (суффиксом).

Позднее файловая система FAT была расширена до VFAT, или «FAT защищенного режима». Эта версия поддерживается в Windows NT и Windows 2000. VFAT скрывает от пользователей все ограничения, накладываемые на имена. Более длинные имена файлов без разделителей поддерживаются благодаря хитрому трюку. В VFAT используется цепь из нескольких стандартных слотов для имен файлов/каталогов, чтобы прозрачно встроить поддержку расширенных имен файлов в структуру обычной файловой системы FAT. В целях совместимости к каждому файлу и каталогу по-прежнему можно обратиться, используя псевдонимы в формате 8.3. Например, к каталогу с именем *Downloaded Program Files* можно обратиться, используя имя *DOWNLO~1*.

Между VFAT и файловыми системами Unix существуют четыре основные различия:

1. Файловые системы FAT не чувствительны к регистру. В Unix попытка открыть файл, используя неверный регистр символов (например *MYFAVORITEFILE* вместо *myfavoritefile*), окончится неудачей. В FAT или VFAT это можно сделать без труда.

2. Второе различие – символы, выбранные для разделения компонент пути. В FAT в качестве разделителя вместо прямого слэша исполь-

зуется обратный слэш. При программировании в Perl это следует учитывать. В Perl обратный слэш является специальным символом. Пути с одинарными разделителями, заключенные в одинарные кавычки (т. е. `$path='\dir\dir\filename'`), не вызывают затруднений. Однако необходимость поставить несколько обратных слэшей рядом (например *\\server\dir\file*) может оказаться потенциально опасной. В таких случаях будьте бдительны и не забывайте удваивать идущие подряд обратные слэши. Некоторые функции и модули могут принимать пути с прямыми слэшами, но при программировании на это лучше не полагаться. Лучше перестраховаться и написать `\\\\winnt\\temp\\...`, чем выяснить, что у вас ничего не получилось из-за того, что преобразование не было выполнено.

3. В FAT с файлами и каталогами связаны специальные флаги, называемые *атрибутами*. Примеры атрибутов – «Read-only» (только для чтения) и «System» (системный).

4. Наконец, последнее отличие – указание корневого каталога. Корень в FAT начинается с указания буквы диска, на котором располагается эта файловая система. Например, абсолютный путь к файлу может быть указан так *c:\home\cindy\docs\resume\current.doc*.

Файловые системы FAT32 и NTFS имеют ту же семантику, что и VFAT. В них одинаково реализована поддержка длинных имен файлов и используется один и тот же способ обозначения корневого каталога. Поддержка длинных имен в NTFS несколько сложнее, т. к. там разрешено использование Unicode в именах файлов. Unicode – это многобайтовая кодировка, которую можно применять для представления всех символов всех языков планеты.

В NTFS также есть несколько функциональных особенностей, отличающих ее от других файловых систем Windows NT/2000 и основных файловых систем Unix. NTFS поддерживает понятие списков контроля доступа (ACL). ACL предоставляет хорошо разграниченный механизм прав доступа к файлам и каталогам. Позже в этой главе мы приведем пример кода, использующего преимущества некоторых из этих отличий.

Перед тем как перейти к другой операционной системе, очень важно хотя бы упомянуть UNC – универсальное соглашение об именовании. UNC – это соглашение о расположении объектов (в нашем случае файлов и каталогов) в сетевом окружении. Вместо имени диска и двоеточия, с которых начинается абсолютный путь, часть *имя диска:* заменяется на *\\сервер\имя_ресурса*. Это соглашение подвержено той же синтаксической двусмысленности обратного слэша в Perl, о которой мы уже говорили. В результате нередко можно увидеть целый ряд косых черточек:

```
$path = "\\\\server\\sharename\\directory\\file"
```

MacOS

Несмотря на GUI-ориентированный подход, иерархическая файловая система MacOS (HFS, Hierarchical File System) также позволяет указывать текстовые имена файлов, хотя для этого нужно немного изловчиться. Абсолютные пути задаются в следующем виде: *Диск/Имя_тома:Папка:Папка:Папка:Имя_файла*. Отсутствие двоеточий указывает на то, что файл находится в текущем каталоге.

В отличие от двух предыдущих операционных систем, пути в HFS считаются абсолютными, если *не* начинаются с разделителя пути (:). Путь HFS, начинающийся с двоеточия, является относительным. Есть небольшое отличие записи пути в MacOS по сравнению с другими файловыми системами – это количество разделителей, которое необходимо указывать при ссылке на объект, стоящий выше в иерархии каталогов. Например, в Unix используется *../../../FileName* для обращения к файлу, находящемуся тремя уровнями выше текущего каталога. В MacOS понадобилось бы использовать четыре разделителя (т. е. *::::FileName*), поскольку необходимо включить ссылку на текущий каталог помимо трех предыдущих уровней.

В HFS длина имен файлов и каталогов ограничена 31 символом. В MacOS версии 8.1 был введен альтернативный многосимвольный формат, названный расширенным форматом MacOS, или HFS+, для поддержки Unicode в именах файлов длиной до 255 символов. И хотя файловая система HFS+ позволяет использовать такие длинные имена, на момент написания этой книги они еще не поддерживаются в MacOS.

Еще более заметным отличием от предыдущих двух файловых систем (по крайней мере, с точки зрения программирования на Perl) является использование в MacOS понятия «fork» (ветвление) при хранении файлов. У каждого файла есть *поток данных* (*data fork*) и *поток ресурсов* (*resource fork*). В первом хранятся данные, а во втором содержатся различные *ресурсы*. В эти ресурсы могут входить исполняемый код (в случае, если это программа), определения пользовательского интерфейса (диалоговые окна, шрифты и т. д.) или любые другие компоненты, определяемые программистом. И хотя в этой главе мы не будем рассматривать ветвления, в MacPerl есть возможность чтения и записи в оба потока.

В MacPerl основные операторы и функции работают только с потоком данных. Например, оператор *–s* возвращает только размер потока данных файла. Если вы хотите обратиться к потоку ресурсов файла, вам придется использовать дополнительные модули, входящие в состав Macintosh Toolbox.

Каждый файл в файловой системе HFS также имеет два специальных тега: *creator* (*создатель*) и *type* (*тип*), позволяющие операционной

системе идентифицировать, каким приложением был создан файл и какого он типа. Эти теги играют ту же роль, что и расширения, используемые в файловых системах FAT (например *.doc* или *.exe*). Позже в этой главе мы увидим, как применять теги тип/создатель в собственных целях.

Сводка различий файловых систем

Ниже представлены те различия, о которых мы только что говорили, и некоторые другие интересные факты (табл. 2.1) .

Таблица 2.1. Сравнение файловых систем

OS и файловая система	Разделитель пути	Чувствительность к регистру	Длина имени файла	Формат абсолютного пути	Формат относительного пути	Уникальные возможности
Unix (файловая система Berkeley Fast File System и другие)	/	Да	В зависимости от операционной системы	*/dir/file*	*dir/file*	Дополнения в зависимости от операционной системы
MacOS (HFS)	:	Да	31 символ (или 255 при использовании HFS+)	*volume:dir:file*	*:dir:file*	Потоки данных/ресурсов, атрибуты создатель/тип
WinNT/2000 (NTFS)	\	Нет	255 символов	*Drive:\dir\file*	*dir\file*	ACL, атрибуты, Unicode в именах файлов
DOS (BASIC FAT)	\	Нет	8.3	*Drive:\dir\file*	*dir\file*	Атрибуты

Учет различий файловых систем в Perl

Perl может помочь создавать программы, в которых учитывается большинство особенностей файловых систем. В его состав входит модуль `File::Spec`, позволяющий нивелировать некоторые различия между файловыми системами. Например, если мы передаем компоненты пути методу `catfile` таким образом:

```
use File::Spec;

$path = File::Spec->catfile("home","cindy","docs","resume.doc");
```

то в Windows NT/2000 переменная `$path` будет иметь значение `home\cindy\docs\resume.doc`, тогда как в Unix она будет иметь значение `home/cindy/docs/resume.doc` и т. д. В модуле `File::Spec` также есть методы, например `curdir` и `updir`, возвращающие обозначения для текущего и родительского каталогов (например «`.`» и «`..`»). Методы этого модуля предоставляют абстрактный способ построения и манипулирования именами путей. Если вы предпочитаете не использовать объектно-ориентированный синтаксис, то модуль `File::Spec::Functions` предоставляет более короткий путь к методам из `File::Spec`.

Прогулка по файловой системе

Наверняка вам уже не терпится посмотреть на какие-нибудь приложения, написанные на Perl. Начнем мы с «прогулки по файловой системе» – одной из наиболее часто встречающихся задач системного администрирования, связанных с файловыми системами. Обычно этот процесс состоит в поиске по всем деревьям каталогов и выполнении некоторого действия в зависимости от результатов поиска. Для этой задачи в каждой операционной системе есть свое средство. В Unix это команда *find*, в Windows NT/2000 – *Find Files or Folders* или *Search For Files or Folders*, а в MacOS – *Find File* или *Sherlock*. Все эти команды полезны для поиска, но выполнять произвольные сложные действия, по мере нахождения требуемых файлов, они не способны. Мы увидим, каким образом Perl позволяет писать более изысканные обзорные программы, начав с простейших и наращивая сложность по мере продвижения.

Для начала давайте рассмотрим простой сценарий, четко ставящий перед нами задачу, которую необходимо решить. В этом сценарии мы предстанем системным администратором Unix с переполненной файловой системой пользователей и исчерпанным дисковым пространством. (Мы начали с Unix, но вскоре рассмотрим и другие операционные системы.)

Мы не можем добавить новое дисковое пространство, не имея на это денег, поэтому попытаемся рациональнее использовать существующие ресурсы. Первый наш шаг – удалить из файловой системы все файлы, которые можно удалить. В Unix первые кандидаты на удаление – это core-файлы, которые остаются после аварийных завершений программ. Большинство пользователей либо не замечают, что эти файлы создаются, либо просто игнорируют их, оставляя напрасно занятыми большие дисковые пространства. Нам нужно средство для поиска и удаления таких «пустышек» из файловой системы.

Обзор файловой системы мы начинаем с чтения содержимого какого-либо каталога и затем продолжаем обход с этой точки. Давайте немного упростим задачу и начнем с программы, которая изучает содержимое текущего каталога и сообщает, были ли найдены в нем core-файлы и другие каталоги-кандидаты для поиска.

Начнем мы с того, что откроем каталог, используя примерно тот же синтаксис, что и для открытия файла. Если попытка была неудачной, мы выходим из программы и выводим сообщение об ошибке (`$!`), установленное вызовом `opendir()`:

```
opendir(DIR,".") or die "Не могу открыть текущий каталог: $!\n";
```

Мы получаем дескриптор каталога, в нашем случае `DIR`, который можно передать функции `readdir()`, чтобы получить список файлов и каталогов в текущем каталоге. Если `readdir()` не может прочитать содержимое каталога, выводится сообщение об ошибке (которое объясняет, почему попытка была неудачной) и программа завершает работу:

```
# читаем имена файлов/каталогов данного каталога в @names
@names = readdir(DIR) or die "Невозможно прочитать текущий каталог:$!\n";
```

Затем закрываем открытый дескриптор каталога:

```
closedir(DIR);
```

Теперь можно работать с полученными именами:

```
foreach $name (@names) {
   next if ($name eq ".");              # пропускаем текущий каталог
   next if ($name eq "..");             # пропускаем родительский каталог

   if (-d $name){                       # является ли каталогом?
      print "найден каталог: $name\n";
      next;                             # можно перейти к следующему имени
                                        # из цикла for
   }
   if ($name eq "core") {               # это файл с именем core?
     print "найден!\n";
  }
}
```

Теперь у нас есть очень простая программа, которая проверяет содержимое одного каталога. Она не только не обходит файловую систему, но даже не «проползает» по ней. Чтобы пройти по файловой системе, мы должны зайти во все каталоги, которые нам встретятся, и посмотреть на их содержимое. Если в этих подкаталогах есть еще подкаталоги, в них мы тоже должны заглянуть.

Когда у вас есть иерархия контейнеров и операция, которая одинаково выполняется на каждом внешнем контейнере и каждом внутреннем контейнере из этой иерархии, это называется рекурсией (по крайней мере в компьютерных науках). До тех пор пока иерархия не очень глубока и не зацикливается (т. е. все контейнеры содержат только непосредственно дочерние объекты и в них отсутствуют ссылки на другие части иерархии), рекурсивные решения имеют наибольший смысл.

Именно с этим мы сталкиваемся и в нашем примере: мы собираемся просмотреть каталог, все его подкаталоги, все их подкаталоги и т. д.

Если вы никогда раньше не видели рекурсивного кода (т. е. кода, который вызывает себя), на первый взгляд он вам может показаться несколько странным. Рекурсивный код чем-то напоминает процесс раскраски матрешек. Там тоже внутри одной куклы находится другая кукла такой же формы, но меньшего размера, внутри которой есть еще одна кукла, и т. д. до самой маленькой куклы в центре.

Инструкция по раскраске матрешек могла бы выглядеть так:

1. Изучите куклу, которая находится перед вами. Внутри у нее есть кукла меньшего размера? Если да, то вытащите ее оттуда.
2. Повторяйте шаг 1 с вытащенными куклами до тех пор, пока не дойдете до центра.
3. Раскрасьте центральную куклу. Когда она высохнет, поместите ее во внешнюю по отношению к ней куклу и повторите шаг 3 со следующим контейнером.

Этот процесс одинаков на каждом шаге. Если у предмета, который вы держите в руках, есть «подпредметы», отложите предмет в сторону и займитесь сначала «подпредметом». Если же подпредметов у него нет, произведите с ним необходимые действия и займитесь последним отложенным предметом.

В терминах программирования это обычно подпрограмма, которая работает с контейнерами. Сначала в ней проверяется наличие подконтейнеров у текущего контейнера. Если они есть, подпрограмма вызывает *себя*, чтобы выполнить какие-либо действия с подконтейнером. Если же их нет, выполняется некое действие и выполняется возврат туда, откуда подпрограмма вызывалась. Если вы не видели код, который вызывает сам себя, я советую посидеть с карандашом и бумагой и проследить выполнение программы до тех пор, пока вы не убедитесь, что она действительно работает.

Давайте рассмотрим примеры кода с рекурсией. Чтобы добавить рекурсию в наш код, мы сначала перенесем операцию сканирования каталога и действия над его содержимым в отдельную подпрограмму `ScanDirectory()`. Подпрограмма `ScanDirectory()` принимает единственный аргумент – каталог, который надо просканировать. Она просматривает текущий каталог, входит в нужный подкаталог и сканирует его. По завершении сканирования подпрограмма возвращается в каталог, из которого она была вызвана. Вот новый вариант программы:

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

# обратите внимание на использование ключа -s для обработки
# параметров. В NT/2000 вам придется вызывать этот сценарий
# ключом -s (т.е. perl -s script), если файловая ассоциация
# в системе не настроена должным образом для файлов perl.
```

```
#
# -s также считается устаревшим, многие программисты
# предпочитают использовать отдельный модуль
# (из семейства модулейGetopt::) для разбора (анализа) параметров.

use Cwd; # модуль для определения текущего рабочего каталога

# Эта подпрограмма принимает имя каталога и рекурсивно
# сканирует файловую систему, начиная с этого места, ищет
# файлы с именем "core"
sub ScanDirectory{
    my ($workdir) = shift;

    my ($startdir) = &cwd; # запомнить, откуда мы начали

    chdir($workdir) or die "Невозможно войти в каталог $workdir:$!\n";
    opendir(DIR, ".") or die "Невозможно открыть $workdir:$!\n";
    my @names = readdir(DIR) or die "Невозможно прочитать $workdir:$!\n";
    closedir(DIR);

    foreach my $name (@names){
        next if ($name eq ".");
        next if ($name eq "..");

if (-l $name){    # пропускаем ссылки
        next;
}
        if (-d $name){                  # это каталог?
            &ScanDirectory($name);
            next;
        }
        if ($name eq "core") {          # имя файла "core"?
            # если в командной строке указан ключ -r, на самом
            # деле удаляем этот файл
            if (defined $r){
                unlink($name) or die "Невозможно удалить $name:$!\n";
            }
            else {
                print "найден в $workdir!\n";
            }
        }
    }
    chdir($startdir) or
          die "Невозможно перейти к каталогу $startdir:$!\n";
}

&ScanDirectory(".");
```

Самое важное отличие от предыдущего примера заключается в изменении поведения программы в случае, если найден подкаталог внутри требуемого каталога. Теперь, если был найден каталог, вместо того

чтобы сообщать об этом, как было в предыдущем примере, наша программа рекурсивно вызывает себя, чтобы изучить сначала содержимое этого каталога. По окончании сканирования всего подкаталога (т. е. вызов `ScanDirectory()` возвращает значение) программа возвращается к просмотру остального содержимого текущего каталога.

Для того чтобы сделать нашу программу полнофункциональным ликвидатором core-файлов, мы добавили в нее функцию удаления файлов. Обратите внимание на то, как это реализовано: файлы будут удалены, только если сценарий вызывается с определенным ключом *–r* (от «remove» – удаление) в командной строке.

В Perl мы указываем встроенный ключ *–s* в строке вызова (`#!/usr/bin/perl -s`) для автоматического разбора параметров. Это самый простой способ разбора параметров, переданных в командной строке. Искушения ради, мы могли бы использовать какой-либо модуль из семейства `Getopt`. Если в командной строке присутствует ключ (например *–r*), то при запуске сценария устанавливается глобальная скалярная переменная с тем же именем (например `$r`). Если Perl вызывается без ключа *–r*, мы вернемся к старому поведению подпрограммы – она будет лишь сообщать, что найдены core-файлы.

Когда вы пишете автоматические утилиты, постарайтесь сделать так, чтобы разрушительные действия были затруднены. Учтите: Perl, как и большинство серьезных языков программирования, позволяет уничтожить файловую систему без особых усилий.

Теперь, чтобы ориентированные на NT/2000 читатели не подумали, что предыдущие примеры к ним не относятся, покажем, что эта программа может пригодиться и для них. Единственное изменение строки:

```
if ($name eq "core") {
```

на:

```
if ($name eq "MSCREATE.DIR") {
```

позволяет создать программу, которая удалит все раздражающие скрытые файлы нулевой длины, забытые инсталляторами некоторых программ Microsoft.

Имея в запасе этот код, давайте вернемся к проблеме, с которой начиналась эта глава. После того как мой портативный компьютер приземлился на пол, средство, которое позволило бы определить, какие файлы можно прочитать с диска, а какие нет, стало необходимо мне, как воздух.

Вот какую программу я написал для этого:

```
use Cwd; # модуль для определения текущего рабочего каталога
$|=1;    # отключаем буферизацию ввода/вывода

sub ScanDirectory {
    my ($workdir) = shift;

    my($startdir) = &cwd; # запоминаем, откуда мы начали

    chdir($workdir) or die "Невозможно зайти в каталог $workdir:$!\n";

    opendir(DIR, ".") or die "Невозможно открыть каталог $workdir:$!\n";
    my @names = readdir(DIR);
    closedir(DIR);

    foreach my $name (@names){
        next if ($name eq ".");
        next if ($name eq "..");

        if (-d $name){                      # это каталог?
            &ScanDirectory($name);
            next;
        }
        unless (&CheckFile($name)){
            print &cwd."/".$name."\n"; # выводим имя
                                       # поврежденного файла
        }
    }
    chdir($startdir) or die "Невозможно перейти к каталогу $startdir:$!\n";
}

sub CheckFile{
    my($name) = shift;

    print STDERR "Проверяется ". &cwd."/".$name."\n";
    # пытаемся получить состояние файла
    my @stat = stat($name);
    if (!$stat[4] && !$stat[5] && !$stat[6] && !$stat[7] && !$stat[8]){
        return 0;
    }
    # пытаемся открыть этот файл
    unless (open(T,"$name")){
        return 0;
    }
    # читаем файл по байту за один раз
    for (my $i=0;$i< $stat[7];$i++){
        my $r=sysread(T,$i,1);
        if ($r != 1) {
            close(T);
            return 0;
```

```
            }
        }
        close(T);
        return 1;
    }

    &ScanDirectory(".");
```

Различия между этой программой и последним примером заключаются в наличии дополнительной подпрограммы для проверки каждого найденного файла. Для каждого файла мы вызываем функцию `stat`, чтобы проверить, можно ли прочитать информацию о файле (например, его размер). Если мы сделать этого не можем, значит, файл поврежден. Если же прочитать эту информацию можно, мы предпринимаем попытку открыть файл. А в качестве последней проверки пытаемся прочитать каждый байт из файла. Это не гарантирует, что файл не поврежден (его содержимое могло измениться), но это говорит о том, что файл можно прочитать.

Вы можете удивиться, зачем применять такую странную функцию, как `sysread()`, для чтения файла, если можно применить `<>` или `read()`, обычно используемые для этого в Perl. Дело в том, что `sysread()` позволяет читать файл побайтно, не применяя обычную буферизацию. Если файл поврежден в позиции `X`, нет смысла ждать, пока будут прочитаны байты в позициях `X+1`, `X+2`, `X+3` и т. д., как это бывает при обращении к обычным библиотечным функциям. Мы хотим, чтобы попытки читать файл в таком случае прекратились немедленно. Обычно файл читается по кускам в целях повышения производительности, но это нежелательно, т. к. в нашем случае компьютер будет издавать ужасающие звуки в течение длительного времени, когда наткнется на поврежденный файл.

Теперь, после рассмотрения использованной мною программы, я расскажу, чем закончилась эта история. После того как рассмотренный сценарий проработал всю ночь (без преувеличений), он нашел 95 поврежденных файлов из 16000. К счастью, ни один из них не был файлом из книги, которую вы сейчас читаете; я снял копии со всех хороших файлов и перенес их в другое место. Perl просто спас положение.

Обход файловой системы при помощи модуля File::Find

Теперь, когда вы познакомились с основами просмотра файловой системы, покажем более быстрый и изящный способ сделать то же самое. В Perl есть модуль `File::Find`, позволяющий эмулировать команду *find* из Unix. Самый простой способ начать работать с этим модулем – воспользоваться командой *find2perl* для генерирования кода-прототипа Perl.

С *find2perl* не всегда легко работать в Perl в отличных от Unix-системах. Для ее выполнения пользователям MacOS, например, понадобится Macintosh Programmer's Workshop (MPW) либо им придется изменить исходный код, чтобы получать `@ARGV` в диалоговом окне. Вот пример кода, позаимствованный у Криса Нандора (Chris Nandor), соавтора «MacPerl: Power and Ease» (MacPerl: Сила и легкость):

```
@ARGV = @ARGV ? @ARGV :
split "\s", MacPerl::Ask("Arguments?");
```

Во всех перенесенных версиях есть модуль `File::Find`, который используют команды *find2perl* и *find.pl*, так что это не должно стать проблемой. Позже в этой главе мы покажем, как вызвать его напрямую.

Предположим, что вам нужна программа для поиска файлов *beesknees* в каталоге */home*. Вот командная строка, использующая команду Unix *find*:

```
% find /home -name beesknees -print
```

Передадим те же параметры команде *find2perl*:

```
% find2perl /home -name beesknees -print
```

и получим:

```
#!/usr/bin/perl
    eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
        if $running_under_some_shell;

require "find.pl";

# Обходим нужные файловые системы

&find('/home');

exit;

sub wanted {
    /^beesknees$/ && print("$name\n");
}
```

Код, сгенерированный командой *find2perl*, довольно прямолинеен. Он загружает необходимую библиотеку *find.pl* при помощи оператора `require`, затем вызывает подпрограмму `&find()` с именем начального каталога. Вскоре мы обсудим назначение подпрограммы `&wanted()`, поскольку именно здесь будут находиться все интересные изменения, которые мы хотим изучить.

Перед тем как вносить изменения в этот код, важно обратить внимание на те немногие моменты, которые могут не показаться очевидными при рассмотрении приведенного фрагмента:

- Те, кто работал над модулем `File::Find`, столкнулись с проблемой переносимости этого модуля на другие платформы. Внутренние подпрограммы модуля `File::Find` действуют так, что один и тот же код работает и в Unix, и в MacOS, и в NT, и в VMS и т. д.
- Хотя код, сгенерированный *find2perl,* на первый взгляд похож на код Perl 4 (например, тут используется `require` для загрузки файла *.pl*), *find.pl* в действительности устанавливает несколько псевдонимов из Perl 5. Обычно бывает полезно заглянуть «под завесу», прежде чем использовать модуль в собственной программе. Если вам нужен исходный код модуля, уже установленного в системе, то, выполнив команду *perl –V* или следующую команду, вы получите список каталогов стандартных библиотек:

```
% perl-e 'print join("\n",@INC,"")'
```

Давайте поговорим о подпрограмме `&wanted()`, которую мы изменим для своих нужд. Подпрограмма `&wanted()` вызывается для каждого найденного `&find()` (или `&File::Find::find()`, чтобы быть точным) файла или каталога при обходе файловой системы. Именно код из `&wanted()` должен выбирать «интересные» файлы или каталоги и работать с ними. В примере, приведенном выше, сначала проверяется соответствие имени файла или каталога строке `beesknees`. Если они совпадают, оператор `&&` заставляет Perl выполнить оператор `print`, чтобы вывести имя найденного файла.

При создании собственных подпрограмм `&wanted()` нам придется учитывать два практических момента. Поскольку `&wanted()` вызывается по одному разу для имени каждого файла или каталога, важно сделать этот код коротким и аккуратным. Чем быстрее мы сможем выйти из подпрограммы `&wanted()`, тем быстрее `find` сможет перейти к новому файлу или каталогу и тем быстрее будет выполняться вся программа. Также важно иметь в виду «закадровые» соображения совместимости, о которых мы недавно упоминали. Было бы позором одновременно иметь переносимый вызов `&find()` и системно-зависимую подпрограмму `&wanted()` (кроме случаев, где этого невозможно избежать). Несколько подсказок, помогающих избежать такой ситуации, можно получить, посмотрев на текст модуля `File::Find`.

Для первого использования модуля `File::Find` давайте перепишем пример, созданный для удаления core-файлов, и затем немного его расширим. Для начала наберем:

```
% find2perl -name core -print
```

что даст нам следующее:

```
require "find.pl";

# Обходим нужные файловые системы

&find('.');

exit;

sub wanted {
    /^core$/ && print("$name\n");
}
```

Затем мы добавим *–s* к строке вызова Perl и изменим подпрограмму `&wanted()`:

```
sub wanted {
   /^core$/ && print("$name\n") && defined $r && unlink($name);
}
```

При вызове программы с ключом *–r* мы получаем необходимую нам функцию (удаление core-файлов). Внесем небольшое изменение, добавляющее некоторую степень защиты нашему потенциально разрушительному коду:

```
sub wanted {
   /^core$/ && -s $name && print("$name\n") &&
               defined $r && unlink($name);
}
```

Теперь, перед тем как выводить имя файла или удалять его, мы проверяем, не является ли размер файла *core* нулевым. Некоторые пользователи иногда создают ссылку на */dev/null* с именем *core* в своем домашнем каталоге, чтобы core-файлы в нем не сохранялись. Параметр *–s* указывается для того, чтобы убедиться, что по ошибке не будут удалены ссылки или файлы нулевой длины. Если мы хотим действовать осторожнее, нам следует выполнить еще две дополнительные проверки:

1. Открыть файл и убедиться, что этот файл действительно является core-файлом. Сделать это можно как из Perl, так и вызвав команду Unix *file*. Определить, что файл действительно является core-файлом, может оказаться достаточно сложно в случае, если файловые системы смонтированы по сети с компьютерами другой архитектуры с иными форматами core-файлов.
2. Посмотреть на время изменения файла. Если кто-то в настоящий момент отлаживает программу, используя файл core, он вряд ли обрадуется, если вы утащите этот файл прямо «из-под нее».

Давайте на время отдохнем от мира Unix и посмотрим на примеры, имеющие отношение к MacOS и Windows NT/2000. Ранее в этой главе я уже говорил, что в MacOS у каждого файла есть два атрибута – *создатель* и *тип*, позволяющие операционной системе определить, какое

приложение создало этот файл и какого он типа. Эти атрибуты хранятся в виде четырехсимвольных строк. Например, для текстового документа, созданного приложением SimpleText, эти атрибуты будут иметь значения `ttxt` (для создателя) и `TEXT` (для типа). Из Perl (только для MacPerl) мы можем получить эту информацию при помощи функции `MacPerl::GetFileInfo()`. Синтаксис ее таков:

```
$type = MacPerl::GetFileInfo(filename);
```

или:

```
($creator,$type) = MacPerl::GetFileInfo(filename);
```

Чтобы найти все текстовые файлы в файловой системе MacOS, мы можем выполнить следующее:

```
use File::Find;

&File::Find::find(\&wanted,"Macintosh HD:");

sub wanted{
  -f $_ && MacPerl::GetFileInfo($_) eq "TEXT" &&
           print "$Find::File::name\n";
}
```

Вы, должно быть, заметили, что это выглядит немного иначе, чем наши предыдущие примеры. Однако действует этот код точно так же. Мы просто вызываем процедуры из `File::Find` напрямую, без *find.pl*. Кроме того, мы используем переменную `$name`, определенную в пространстве имен `File::Find`, чтобы вывести абсолютный путь файла, а не только его имя. Взгляните на полный список переменных, определяемых `File::Find` при обходе файловой системы (табл. 2.2).

Таблица 2.2. Переменные File::Find

Переменная	Смысл
`$_`	Имя текущего файла
`$File::Find::dir`	Имя текущего каталога
`$File::Find::name`	Полный путь для текущего файла (т. е. `$File::Find::dir/$_`)

Вот похожий пример, но для NT/2000:

```
use File::Find;
use Win32::File;

&File::Find::find(\&wanted,"\\");

sub wanted{
  -f $_ &&
```

```
    # значение переменной attr присваивается функцией
    # Win32::File::GetAttributes
    (Win32::File::GetAttributes($_,$attr)) &&
    ($attr & HIDDEN) &&
     print "$File::Find::name\n";
}
```

Этот пример ищет по всей файловой системе на текущем диске все скрытые файлы (т. е. те файлы, у которых установлен атрибут `HIDDEN`). Этот код работает и на NTFS и на FAT.

А вот пример для файловой системы NTFS, который ищет все файлы, если к ним разрешен полный доступ для специальной группы `Everyone`, и выводит их имена:

```
use File::Find;
use Win32::FileSecurity;

# Определяем маску DACL для полного доступа
$fullmask = Win32::FileSecurity::MakeMask(FULL);

&find(\&wanted,"\\");

sub wanted {
    # Win32::FileSecurity::Get не любит файл подкачки
    # pagefile.sys, пропустить его
    next if ($_ eq "pagefile.sys");
    (-f $_) &&
        Win32::FileSecurity::Get($_, \%users) &&
        (defined $users{"Everyone"}) &&
        ($users{"Everyone"} == $fullmask) &&
         print "$File::Find::name\n";
}
```

В вышеприведенном коде мы запрашиваем все файлы у списка контроля доступа ACL (кроме файла подкачки Windows NT). Затем мы проверяем, есть ли в этом списке запись для группы `Everyone`. Если есть, мы сравниваем запись `Everyone` со значением для полного доступа (полученным `MakeMask()`) и выводим абсолютный путь файла, если они совпадают.

А вот еще один пример из реальной жизни, демонстрирующий, насколько полезным может оказаться даже самый простой код. Недавно я пытался дефрагментировать (заново перестроенный) раздел NT на диске своего портативного компьютера, но все закончилось сообщением об ошибке `Metadata Corruption Error` (повреждение метаданных). Внимательно изучая веб-сайт производителя программного обеспечения, я нашел там замечание, что «такая ситуация может быть вызвана наличием файлов, длина имен которых превышает допустимую в Windows NT». Там было предложено найти эти файлы, копируя каждый каталог на новое место и сравнивая количество файлов в оригинале и

копии. Если в копии каталога файлов меньше, необходимо найти те файлы, которые не были скопированы.

С учетом количества каталогов в моем разделе и времени, необходимого для выполнения этой процедуры, такое решение представляется просто нелепым. Вместо этого я написал следующее, используя уже обсужденные методы:

```
require "find.pl";

# Обходим нужные файловые системы

&find(\&wanted,'.')[1];
print "max:$max\n";

exit;

sub wanted {
    return unless -f $_;
    if (length($_) > $maxlength){
        $max = $name;
        $maxlength = length($_);
    }
    if (length($name) > 200) { print $name,"\n";}
}
```

В результате будут выведены имена файлов длиной более 200 символов, а также самое длинное найденное имя. Работа сделана, спасибо Perl.

Давайте снова вернемся к Unix, чтобы закончить этот раздел довольно сложным примером. Идея, которая представляется слишком простой в контексте системного администрирования, но в итоге может принести огромную пользу – это понятие наделения пользователя полномочиями. Если ваши пользователи могут решить свои проблемы самостоятельно при помощи средств, которые вы им предоставляете, от этого все только выиграют.

Большая часть этой главы посвящена решению проблем, возникающих при переполнении файловой системы. Зачастую это происходит из-за того, что пользователи недостаточно осведомлены о своем окружении, либо из-за того, что слишком обременительно выполнять операции по управлению дисковым пространством. Множество писем в службу поддержки начинаются со слов «В моем домашнем каталоге больше нет свободного места, но я не знаю из-за чего». Вот скелет сценария *needspace*, который может помочь пользователям, столкнувшимся с этой проблемой. Пользователь просто набирает *needspace*, и сценарий пытается найти в домашнем каталоге пользователя то, что

[1] `\&wanted` в оригинале пропущено. – *Примеч. науч. ред.*

Когда не надо использовать модуль File::Find

Когда метод `File::Find` *не* подходит? На ум приходят четыре ситуации:

1. Если файловая система, с которой вы работаете, не следует обычной семантике, вы не сможете применять этот модуль. Например, драйвер файловой системы NTFS для Linux, который я использовал при решении проблемы с упавшим компьютером, почему-то не выводил в пустых каталогах «.» или «..». Это очень мешало `File::Find`.
2. Если вам нужно изменять имена каталогов *во время обхода* файловой системы, `File::Find` теряется и начинает вести себя непредсказуемым образом.
3. Если вам нужно разыменовывать символические ссылки на каталоги (для Unix), `File::Find` пропустит их.
4. Если вам нужно обойти файловую систему, смонтированную на вашей машине (например, файловую систему Unix, смонтированную через NFS на машине с Windows), `File::Find` будет использовать семантику, принятую для «родной» файловой системы.

Вряд ли вы столкнетесь с этими ситуациями, но если это произойдет, загляните в раздел этой главы, посвященный обходу файловой системы вручную.

можно удалить. Он ищет файлы двух типов: резервные копии и те файлы, которые можно автоматически создать заново. Давайте внимательно рассмотрим код:

```
use File::Find;
use File::Basename;

# массив расширений файлов и расширений, из которых они могут
# быть получены
% derivations = (".dvi" => ".tex",
                 ".aux" => ".tex",
                 ".toc" => ".tex",
                 ".o"   => ".c",
                 );
```

Мы начнем с того, что загрузим нужные нам библиотеки: знакомый уже модуль `File::Find` и другую полезную библиотеку `File::Basename`. Эта библиотека пригодится при разборе путей файлов. Затем мы инициализируем хэш-таблицу известными расширениями производных файлов; например, мы знаем, что если выполнить команду *TeX* или *LaTeX* для файла *happy.tex*, мы можем получить файл *happy.dvi*, и что

happy.o можно получить, скорее всего, скомпилировав файл *happy.c* компилятором C. Выражение «скорее всего» употреблено потому, что иногда требуется несколько исходных файлов, чтобы сгенерировать один файл. Однако мы можем делать только простые предположения, основываясь на расширениях файлов. Обобщенный анализ зависимостей – сложная задача, и мы даже не будем пытаться решать ее здесь.

Затем мы определяем местонахождение домашнего каталога пользователя, получая идентификатор пользователя, выполняющего сценарий (`$<`), и передаем его функции `getpwuid()`. `getpwuid()` возвращает информацию из файла паролей в виде списка (подробности об этом позже); индекс массива (`[7]`) выбирает из этого списка элемент, соответствующий домашнему каталогу. Существуют способы получить эту информацию при помощи данных из командного интерпретатора (например, обратившись к переменной окружения `$HOME`), но в таком виде код переносится лучше.

Получив домашний каталог, мы переходим в него и начинаем сканирование, используя вызов `&find()` так же, как и в предыдущих примерах:

```
$homedir=(getpwuid($<))[7];         # находим домашний каталог пользователя
chdir($homedir) or
  die "Невозможно войти в ваш домашний каталог $homedir:$!\n";

$|=1; # не буферизованный вывод в STDOUT
print "Поиск";
find(\&wanted, ".");  # проходим по каталогам, &wanted выполняет
                      # всю работу
```

Вот как выглядит вызываемая нами подпрограмма `&wanted()`. Сначала она ищет core-файлы, а также резервные копии и автосохраненные файлы, остающиеся после редактирования в *emacs*. Мы считаем, что эти файлы можно удалить, не проверяя существование исходных файлов (вероятно, это небезопасное предположение). Если такие файлы найдены, их размеры и пути к ним сохраняются в хэше, ключами которого являются пути к файлам, а значениями – размеры этих файлов.

В остальной части подпрограммы подобным образом отыскиваются производные файлы. Мы вызываем подпрограмму `&BaseFileExists()`, для того чтобы убедиться, что эти файлы можно получить из других файлов этого же каталога. Если подпрограмма возвращает значение «истина», мы сохраняем имя файла и его размер для последующего использования:

```
sub wanted {
    # выводим точку для каждого каталога, чтобы пользователь
    # видел, что что-то происходит
    print "." if (-d $_);

    # ищем только файлы
    return unless (-f $_);
```

```
    # ищем core-файлы, сохраняем их в хэше %core и возвращаемся
    $_ eq "core" && ($core{$File::Find::name} = (stat(_))[7])
                 && return;

    # ищем резервные копии и автосохраненные файлы, оставшиеся
    # после редактирования файла в emacs
    (/^#.*#$/ || /~$/) &&
      ($emacs{$File::Find::name}=(stat(_))[7]) &&
      return;

    # ищем производные tex-файлов
     (/\.dvi$/ || /\.aux$/ || /\.toc$/) &&
       &BaseFileExists($File::Find::name) &&
       ($tex{$File::Find::name} = (stat(_))[7]) &&
       return;

    # ищем производные .o файлов
     /\.o$/ &&
       &BaseFileExists($File::Find::name) &&
       ($doto{$File::Find::name} = (stat(_))[7]) &&
       return;
}
```

Вот текст подпрограммы, проверяющей, можно ли получить данный файл из другого файла в этом же каталоге (например, существует ли файл *happy.o*, если мы нашли файл *happy.c*):

```
sub BaseFileExists {
    my($name,$path,$suffix) =
     &File::Basename::fileparse($_[0],'\..*');

    # если мы не знаем, как получить файл этого типа
    return 0 unless (defined $derivations{$suffix});

    # все просто, мы видели исходный файл раньше
    return 1 if (defined
                 $baseseen{$path.$name.$derivations{$suffix}});

    # если файл (или ссылка на файл) существует и
    # имеет ненулевой размер
    return 1 if (-s $name.$derivations{$suffix} &&
                ++$baseseen{$path.$name.$derivations{$suffix}});
}

print "готово.\n";
```

Вот как выполняется эта подпрограмма:

1. `&File::Basename::fileparse()` **используется для выделения из пути имени файла, пути к файлу и его суффикса (например *resume.dvi*, */home/cindy/docs/*, *.dvi*).**

2. Затем суффикс файла проверяется, чтобы определить, считаем мы этот файл производным или нет. Если нет, мы возвращаем значение 0 (т. е. «ложь» в скалярном контексте).
3. Затем мы проверяем, встречался ли нам файл, исходный (base file) по отношению к данному, и если да, то возвращаем значение «истина». В некоторых ситуациях (в частности, в случае с *TeX/LaTeX*), из одного исходного файла можно получить несколько производных. Такая проверка ускоряет выполнение сценария, т. к. мы в этом случае избавлены от обхода файловой системы.
4. Если мы не встречали раньше исходный файл (с тем же именем, но другим расширением), то проверяем, существует ли он и больше ли нуля его размер. Если да, мы сохраняем информацию о файле и возвращаем 1 (т. е. «истина» в скалярном контексте).

Теперь нам остается только вывести информацию, которую мы собрали при обходе файловой системы:

```
foreach my $path (keys %core){
    print "Найден core-файл, занимающий ".&BytesToMeg($core{$path}).
          "MB в ".&File::Basename::dirname($path).".\n";
}

if (keys %emacs){
    print "Следующие файлы, скорее всего, являются резервными копиями,
           созданными emacs:\n";

    foreach my $path (keys %emacs){
        $tempsize += $emacs{$path};
        $path =~ s/^$homedir/~/;      # изменяем путь, чтобы
                                      # вывод был аккуратнее
        print "$path ($emacs{$path} байт)\n";
    }
    print "\nОни занимают ".&BytesToMeg($tempsize)."MB в сумме.\n";
    $tempsize=0;
}

if (keys %tex){
    print "Следующие файлы, скорее всего, можно получить заново, если
           запустить La/TeX:\n";
    foreach my $path (keys %tex){
        $tempsize += $tex{$path};
        $path =~ s/^$homedir/~/;      # изменяем путь, чтобы
                                      # вывод был аккуратнее
        print "$path ($tex{$path} байт)\n";
    }
    print "\nОни занимают ".&BytesToMeg($tempsize)."MB в сумме.\n";
    $tempsize=0;
}

if (keys %doto){
```

```
    print "Следующие файлы, скорее всего, можно получить, если вновь
           скомпилировать исходные файлы:\n";
    foreach my $path (keys %doto){
        $tempsize += $doto{$path};
        $path =~ s/^$homedir/~/;      # изменяем путь, чтобы
                                      # вывод был аккуратнее
        print "$path ($doto{$path} байт)\n";
    }
    print "\nОни занимают ".&BytesToMeg($tempsize)."MB в сумме.\n";
    $tempsize=0;
}

sub BytesToMeg{ # преобразуем размер в байтах в формат X.XXMB
    return sprintf("%.2f",($_[0]/1024000));
}
```

Прежде чем закончить этот разговор, надо заметить, что предыдущий пример можно расширять множеством способов. Пределов для программ такого типа просто не существует. Вот несколько идей:

- Используйте более сложные структуры данных для хранения расширений производных файлов и найденных файлов. Приведенный выше код был написан с расчетом на то, чтобы его было легко читать людям, не очень хорошо разбирающимся со структурами данных в Perl. В нем используются повторяющиеся фрагменты и его довольно тяжело расширить, если в этом появится необходимость. В идеале было бы неплохо, чтобы все расширения производных файлов не были связаны со специальными хэшами (например `%tex`) в коде.
- Ищите каталоги, в которых веб-броузеры кэшируют страницы (это очень распространенный источник потерянного пространства на диске).
- Предложите пользователю удалить найденные файлы. Для удаления файлов используйте оператор `unlink()` и подпрограмму `rmpath` из модуля `File::Path`.
- Больше анализируйте файлы, вместо того чтобы строить предположения по их именам.

Работа с дисковыми квотами

Perl-сценарии, подобные приведенным в предыдущем разделе, предлагают нам способы, позволяющие манипулировать ненужными файлами, обилие которых приводит к переполнению диска. Но даже если такие сценарии запускать регулярно, наши действия все равно будут ответными, т. к. администратор уделяет таким файлам время только тогда, когда они уже появились и захламили файловую систему.

Существует другой, более активный подход: квоты на файловые системы. Квоты, или ограничения операционной системы, позволяют огра-

ничить объем дискового пространства, отведенный определенному пользователю. Квоты существуют в Windows 2000 и во всех современных разновидностях Unix. В NT4 для этого необходимы продукты сторонних разработчиков, а в MacOS для пользователей существует понятие S.O.L. (Simply or Sore Out of Luck – просто не повезло).[1]

И хотя это активный подход, поддерживать его гораздо сложнее, чем «чистящие» сценарии, поскольку он применяется ко всем файлам, а не только к лишним, например, к core-файлам. Большинство системных администраторов считают лучшей стратегией использовать комбинацию автоматических «чистящих» сценариев и дисковых квот. Первое помогает ограничить использование второго.

В этом разделе мы поговорим о работе с дисковыми квотами в Unix средствами Perl. Перед тем как перейти к конкретному разговору, нужно понять, как квоты устанавливаются и как их можно ввести вручную. Чтобы сделать возможным применение квот в файловой системе, системный администратор Unix обычно добавляет запись в таблицу смонтированных файловых систем (например файл */etc/fstab* или */etc/vfstab*) и перезагружает систему либо вручную вызывает команду, разрешающую использование квот (обычно *quotaon*).

Вот пример файла */etc/vfstab* из Solaris:

```
#device           device               mount  FS    fsck   mount     mount
#to mount         to fsck              point  type  pass   at boot   options
/dev/dsk/c0t0d0s7 /dev/dsk/c0d0t0d0s7  /home  ufs   2       yes       rq
```

Параметр *rq* в последнем столбце включает квоты для файловой системы. Хранятся они для каждого пользователя отдельно. Для просмотра информации о квотах пользователя на всех смонтированных файловых системах, на которых квоты применяются, надо вызвать команду *quota*:

```
$ quota -v sabrams
```

чтобы получить данные, подобные этим:

```
Disk quotas for sabrams (uid 670):
Filesystem   usage  quota   limit  timeleft files quota  limit timeleft
/home/users  228731 250000  253000              0     0      0
```

В следующих нескольких примерах нас будут интересовать только первые три колонки этого вывода. Первое число – это объем занятого в настоящий момент дискового пространства пользователем *sabrams* на файловой системе, смонтированной как */home/users*. Второе – это размер «мягкой квоты» пользователя. Мягкая квота – это объем дискового пространства, после превышения которого операционная система в

[1] Здесь имеется в виду, что в MacOS дисковые квоты просто не предусмотрены. – *Примеч. науч. ред.*

течение некоторого времени выдает предупреждения, но не ограничивает выделение дискового пространства. Последнее число – это «жесткая квота», т. е. абсолютный верхний предел для объема пространства, занятого данным пользователем. Если программа попытается использовать еще некоторое дисковое пространство после превышения пользователем квоты, операционная система отвергнет этот запрос и вернет сообщение об ошибке, подобное `disk quota exceeded`.

При желании изменить размеры квот вручную необходимо использовать команду *edquota*, которая загружает небольшой временный файл с информацией о текущих размерах квот в редактор, определяемый переменной окружения `EDITOR` командного интерпретатора. Вот пример такого файла с информацией об ограничениях для четырех файловых систем, на которых применяются квоты. Скорее всего, домашний каталог этого пользователя находится в каталоге */exprt/server2*, т. к. только в этой файловой системе для него отведены квоты:

```
fs /exprt/server1 blocks (soft = 0, hard = 0) inodes (soft = 0, hard = 0)
fs /exprt/server2 blocks (soft = 250000, hard = 253000) inodes (soft = 0,
hard = 0)
fs /exprt/server3 blocks (soft = 0, hard = 0) inodes (soft = 0, hard = 0)
fs /exprt/server4 blocks (soft = 0, hard = 0) inodes (soft = 0, hard = 0)
```

Использование *edquota* вручную может быть удобным способом редактирования ограничений одного пользователя, но это совершенно невозможно в случае десятков, сотен и тысяч учетных записей пользователей. Один из недостатков Unix – нехватка утилит командной строки для редактирования информации о квотах. В большинстве версий Unix есть функции библиотеки C для выполнения этой задачи, но нет утилит командной строки для написания сценариев. И, следуя девизу Perl, что «Существует более одного способа сделать это» («There's More Than One Way To Do It», TMTOWTDI, произносится как «тим-тоади», «tim-toady»), мы рассмотрим два различных способа установки квот из Perl.

Редактирование квот при помощи edquota

Первый способ требует некоторой хитрости с нашей стороны. Совсем недавно мы упоминали процесс ручной установки дисковых квот: *edquota* вызывает редактор, в котором пользователь редактирует небольшой текстовый файл, после чего эти изменения используются для обновления информации о квотах. Не существует никаких указаний на то, какие данные необходимо вводить, чтобы внести изменения. В действительности, не существует даже ограничений на то, какой редактор будет применяться. Все, что нужно команде *edquota*, – это программа, которую можно запустить и которая необходимым образом изменит маленький текстовый файл. Подойдет любой допустимый путь (заданный переменной окружения `EDITOR`) к такой программе. Почему бы не

указать программе *edquota* сценарий на Perl? Давайте и рассмотрим такой сценарий в нашем следующем примере.

Наш сценарий должен будет выполнять две задачи: во-первых, он должен принимать от пользователя аргументы командной строки, устанавливать соответствующим образом переменную окружения `EDITOR` и вызывать программу *edquota*. В свою очередь, *edquota* запустит другую копию нашей программы, которая и займется собственно редактированием этого временного файла. Ниже показана схема действий (рис. 2.1).

Второй копии программы необходимо сообщить, что именно она должна изменить по требованию исходной программы. Как она получает эту информацию из первой копии, вызвавшей *edquota*, менее очевидно, чем этого бы хотелось. В странице руководства по *edquota* сказано: «Вызывается редактор *vi(1)*, если только в переменной EDITOR не указано иное». Идея передать аргументы командной строки через EDITOR или другую переменную окружения довольно опасна хотя бы потому, что мы не знаем, как на это отреагирует утилита *edquota*. Поэтому нам придется полагаться на один из методов межпроцессного взаимодействия, доступных в Perl. Например, два процесса могут:

- Передавать друг другу временный файл
- Создать именованный канал и общаться по нему
- Передавать AppleEvents (в MacOS)
- Использовать объект синхронизации (mutex) или оговоренные ключи реестра (в NT/2000)
- Общаться через сокеты
- Использовать разделяемую память

И так далее. От вас как от программиста зависит, какой метод вы выберете, хотя зачастую определять выбор будут данные. Рассматривая их, вы будете принимать во внимание:

- Направление соединения (одно- или двунаправленное?)
- Частоту соединения (будет передано одно сообщение или несколько кусочков?)
- Размер данных (будет это 10-мегабайтный файл или 20 символов?)
- Формат данных (будет это двоичный файл или просто текст фиксированной ширины, разделенный определенным символом?)

Наконец, учитывайте то, насколько сложным вы хотите сделать свой сценарий.

В нашем случае мы собираемся выбрать простой, но мощный метод обмена информацией. Так как первый процесс должен передать второму только набор инструкций по изменению информации (какие квоты изменять и на какие значения), мы установим между ними стандартный

канал Unix.[1] Первый процесс пошлет запрос на изменение в поток вывода, а копия, запущенная программой *edquota*, прочитает эту информацию со своего потока ввода.

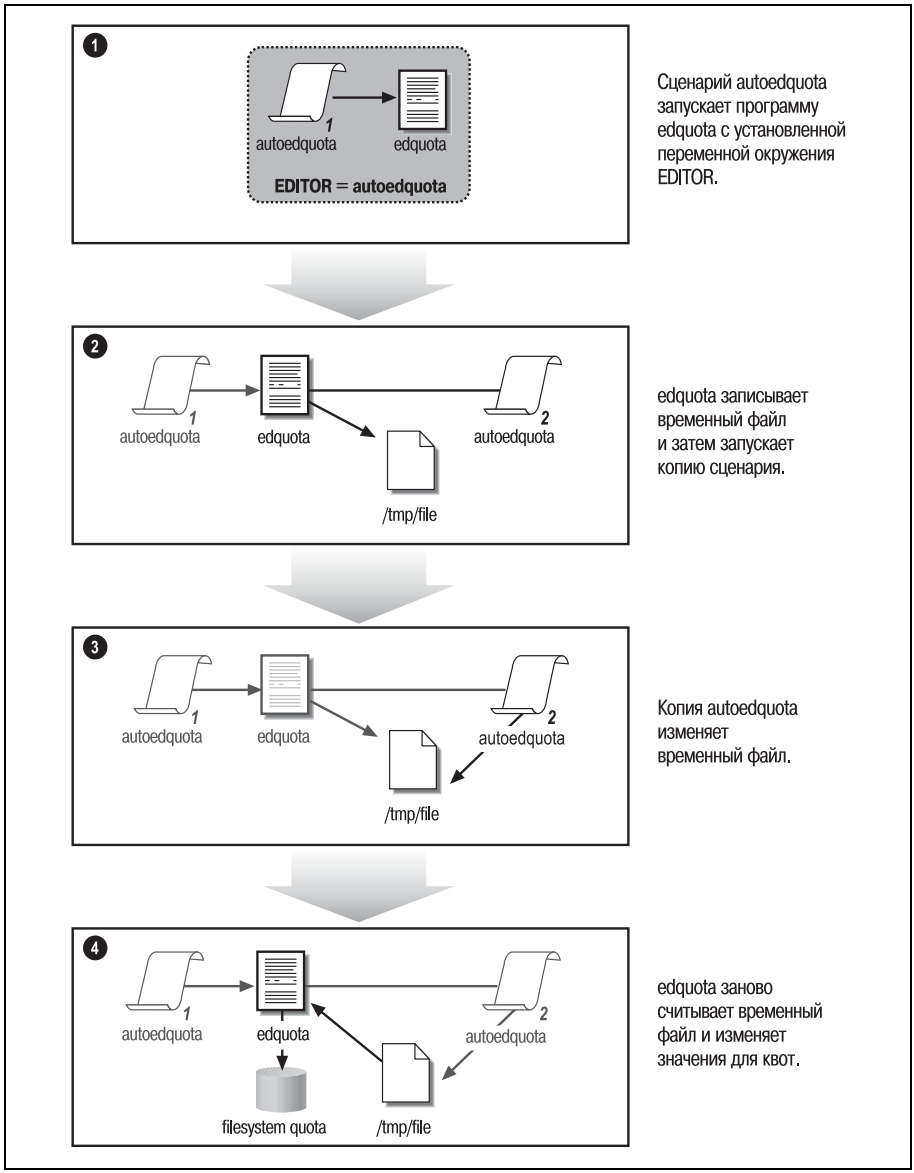

***Рис. 2.1.** Изменение квот при помощи «ловкости рук»*

[1] В действительности, это канал к программе *edquota*, которая передает потоки ввода и вывода запущенному Perl-сценарию.

Давайте напишем программу. Первое, что должна делать программа при запуске, – это решить, какую роль она должна выполнять. Пусть при первом вызове она получает несколько аргументов командной строки (то, что надо изменить), тогда как во второй раз, уже вызванная программой *edquota*, она получает только один аргумент (имя временного файла). Программа требует наличия нескольких ключей командной строки, если вызывается более чем с одним аргументом, поэтому мы можем полагаться на данное допущение при выборе роли для нашего сценария. Вот как выглядит код, определяющий роль сценария:

```
$edquota = "/usr/etc/edquota";      # путь к edquota
$autoedq = "/usr/adm/autoedquota"; # полный путь к этому сценарию

# это первый или второй запуск?

# если присутствует более одного аргумента – это первый запуск
if ($#ARGV > 0) {
    &ParseArgs;
    &CallEdquota;
}
# в противном случае это второй запуск, и мы должны выполнить
# редактирование
else {
    &EdQuota();
}
```

Рассмотрим код, вызываемый при первом запуске и используемый для анализа аргументов и вызова *edquota* через канал:

```
sub ParseArgs{
    use Getopt::Std; # для обработки параметров

    # Устанавливаем переменную $opt_u равной идентификатору
    # пользователя, $opt_f – равной имени файловой системы,
    # $opt_s – в значение для мягкого ограничения и $opt_h –
    # в значение для жесткого ограничения
    getopt("u:f:s:h:"); # двоеточие говорит о том, что у этого
                        # ключа есть аргумент
    die "ИСПОЛЬЗОВАНИЕ: $0 -u uid -f <fsystem> -s <softq> -h <hardq>\n"
      if (!$opt_u || !$opt_f || !$opt_s || !$opt_h);
}

sub CallEdquota{
    $ENV{"EDITOR"} = $autoedq; # записываем в
                               # переменную окружения EDITOR
                               # путь к нашему сценарию

    open(EPROCESS, "|$edquota $opt_u") or
      die "Невозможно запустить edquota:$!\n";
```

```
    # посылаем измененные строки во вторую копию сценария
    print EPROCESS "$opt_f|$opt_s|$opt_h\n";

    close(EPROCESS);
}
```

Вот вторая часть выполняемого действия:

```
sub EdQuota {
    $tfile = $ARGV[0]; # получаем имя временного файла edquota

    open(TEMPFILE, $tfile) or
      die "Невозможно открыть временный файл $tfile:$!\n";

    # открываем файл-черновик, можно было бы
    # использовать и IO::File → new_tmpfile()
    open(NEWTEMP, ">$tfile.$$") or
      die " Невозможно открыть временный файл-черновик $tfile.$$:$!\n";

    # получаем строку ввода из первого вызова и отсекаем символ \n
    chomp($change = <STDIN>);
    my($fs,$soft,$hard) = split(/\|/,$change); # разбираем ответ

    # считываем из временного файла строку. Если она содержит
    # информацию о файловой системе, которую мы хотим
    # изменить, изменяем эти значения. Записываем строку
    # (вероятно, измененную) в черновик.
    while (<TEMPFILE>){
      if (/^fs $fs\s+/){
        s/(soft\s*=\s*)\d+(, hard\s*=\s*)\d+/$1$soft$2$hard/;
      }
        print NEWTEMP;
    }
    close(TEMPFILE);
    close(NEWTEMP);

    # перезаписываем временный файл измененным черновиком,
    # так что изменения передаются edquota
    rename("$tfile.$$",$tfile)
      or die "Невозможно переименовать $tfile.$$ в $tfile:$!\n";
}
```

Приведенная выше программа – это всего лишь скелет, но он все же описывает способ автоматического изменения квот. Если вы когда-то пытались изменять вручную квоты для многих пользователей, приведенный выше пример должен вас порадовать. Перед тем как использовать такой сценарий для внесения реальных изменений, необходимо добавить механизм, защищающий от внесения противоречащих друг другу изменений, а также механизм проверки ошибок. В любом случае, технология такого рода может вам пригодиться и в других ситуациях, помимо работы с квотами.

Редактирование квот при помощи модуля Quota

Когда-то очень давно предыдущий метод (или, если быть честным, предыдущий «хак») был единственным способом автоматизировать изменения квот, если, конечно, вас не радовала перспектива редактирования системных вызовов из библиотеки C, чтобы встроить их в интерпретатор Perl. Теперь, когда механизм расширений Perl существенно упростил встраивание библиотечных вызовов в Perl, создание модуля Quota для Perl стало только делом времени. Благодаря Тому Зорнеру (Tom Zoerner) и другим процесс установки квот средствами Perl теперь намного проще, если этот модуль поддерживает вашу версию Unix. Если нет, предыдущий метод все равно будет работать нормально.

Вот небольшой пример, в котором принимаются те же аргументы, что и в предыдущем:

```
use Getopt::Std;
use Quota:;

getopt("u:f:s:h:");
die "USAGE: $0 -u uid -f <filesystem> -s <softquota> -h <hard quota>\n"
      if (!$opt_u || !$opt_f || !$opt_s || !$opt_h);

$dev = Quota::getcarg($opt_f) or die "Невозможно преобразовать путь
                                      $opt_f:$!\n";

($curblock,$soft,$hard,$btimeout,$curinode,$isoft,$ihard,$itimeout)=
      Quota::query($dev,$uid) or die "Невозможно запросить квоту для
                                      $uid:$!\n";

Quota::setqlim($dev,$opt_u,$opt_s,$opt_h,$isoft,$ihard,1) or
      die " Невозможно установить квоту:$!\n";
```

После анализа аргументов остаются три простых шага: во-первых, мы используем Quota::getcarg() для получения идентификатора устройства, который передается другим подпрограммам. Затем мы передаем этот идентификатор и идентификатор пользователя функции Quota::query(), чтобы получить текущие параметры квот. Нам нужны эти настройки, чтобы не нарушить ограничения, которые мы не будем изменять (например, число файлов). Наконец, мы устанавливаем квоту. Вот и все, всего лишь три строчки кода на Perl.

Помните, что девиз Perl TMTOWTDI означает «существует более одного способа сделать это», но это вовсе не значит «несколько одинаково хороших способов».

Получение сведений об использовании файловой системы

Располагая методами контроля над использованием файловой системы, которые мы только что рассмотрели, вполне естественно проверить, как они работают. Чтобы закончить эту главу, давайте рассмотрим способ получения сведений об использовании файловой системы для каждой из операционных систем.

MacOS – операционная система, для которой эта задача наиболее сложна. В MacOS есть процедура `PBHGetVInfo` из Macintosh Toolbox для получения информации о томах, но в настоящее время не существует модулей MacPerl, которые сделали бы простым вызов этой функции. Вместо этого мы используем обходной путь и поручим *Finder* запросить эту информацию для нас. Благодаря модулю связки это легко осуществить, но из-за необходимости предварительных действий в MacOS эта задача выполняется сложнее всего.

Все используемые далее материалы основаны на работе Криса Нандора (Chris Nandor) и их можно найти на *http://pudge.net* или на CPAN. Наберитесь терпения, пока мы будем рассматривать процесс настройки шаг за шагом:

1. Установите связку *cpan-mac*. В *cpan-mac* входит модуль `CPAN.pm`, написанный Андреасом Кенигом (Andreas J. König), и другие нужные методы, о которых говорилось в главе 1. Даже если вы не хотите получать сведения об использовании файловой системы в MacOS, эту связку лучше установить. После того как вы ее установите, обязательно выполните все инструкции из файла *README*.
2. Установите последнюю версию модуля `Mac::AppleEvents::Simple`, перетащив файл дистрибутива в *installme*.
3. Установите модуль `Mac::Glue`. «Крошка» *installme* распаковывает содержимое дистрибутива `Mac::Glue` в новый каталог в процессе установки. Обязательно запустите сценарии установки *gluedialect* и *gluescriptadds* из подкаталога *scripts* того каталога, в который распакован дистрибутив.
4. Создайте файл связки для *Finder*. Откройте *System Folder* и перетащите файл *Finder* на вершину *gluemac*, чтобы создать необходимый файл (и, что очень приятно, документацию для него).

Этот сложный процесс установки позволяет нам написать такую простенькую на вид программу:

```
use Mac::Glue qw(:all);

$fobj  = new Mac::Glue 'Finder';

$volumename = "Macintosh HD"; # имя одного из смонтированных дисков
```

```
$total = $fobj->get($fobj->prop('capacity',
                    disk => $volumename),
                    as => 'doub');
$free  = $fobj->get($fobj->prop('free_space',
                    disk => $volumename),
                    as => 'doub');

print "свободно $free байт из $total\n";
```

Перейдем к более простым темам. Для того чтобы получить ту же информацию на компьютере с Win32, мы могли бы использовать модуль `Win32::AdminMisc`, написанный Дэйвом Ротом (Dave Roth):

```
use Win32::AdminMisc;

($total,$free) = Win32::AdminMisc::GetDriveSpace("c:\\");

print "свободно $free байт из $total\n";
```

И, наконец, закончим эту главу рассмотрением эквивалентного варианта для Unix. Существует несколько модулей для Unix, включая `Filesys::DiskSpace` Фабьена Тассена (Fabien Tassin), `Filesys::Df` Яна Гафри (Ian Guthrie) и `Filesys::DiskFree` Алана Баркли (Alan R. Barclay). В первых двух из них используется системный вызов `statvfs()`, а последний анализирует вывод Unix-команды *df* на всех поддерживаемых системах. Выбор какого-либо из этих модулей зависит от ваших предпочтений и от того, что поддерживается вашей операционной системой. Я предпочитаю `Filesys::Df`, поскольку он предлагает массу возможностей и не запускает другой процесс (потенциальный риск, как уже говорилось в главе 1) во время запроса. Вот один из способов написания эквивалентного двум предыдущим примерам варианта:

```
use Filesys::Df;

$fobj = df("/");

print $fobj->{su_bavail}*1024." байт в ".
      $fobj->{su_blocks}*1024." байт свободно\n";
```

Нам необходимо выполнить некоторые расчеты (а именно: `*1024`), поскольку `Filesys::Df` возвращает значения в блоках, а каждый блок равен 1024 байтам в нашей системе. Функции `df()` этого модуля может быть передан второй необязательный параметр, определяющий размер блока, если это необходимо. Следует также отметить в этом коде два запрошенных значения хэша. `su_bavail` и `su_blocks` – это возвращенные модулем значения размера диска и объема использованного пространства. В большинстве файловых систем Unix команда *df* выводит информацию об использованном пространстве, скрывая 10% диска, зарезервированных для суперпользователя. При желании узнать размер доступного и свободного в настоящий момент дискового про-

странства с точки зрения обычного пользователя мы должны использовать `user_blocks` и `user_bavail`.

Мы только что видели ключевые фрагменты кода, при помощи которых можно создавать более сложные системы, наблюдающие и управляющие пространством на дисках. Эти наблюдатели за файловыми системами помогут вам решить проблемы с пространством на дисках еще до их появления.

Информация о модулях из этой главы

Имя	Идентификатор на CPAN	Версия
`File::Find` (входит в состав Perl)		
`File::Spec` (входит в состав Perl)		
`Cwd` (входит в состав Perl)		
`Win32::File::GetAttributes` (входит в состав ActiveState Perl)		
`Win32::FileSecurity` (входит в состав ActiveState Perl)		
`File::Basename` (входит в состав Perl)		
`Getopt::Std` (входит в состав Perl)		
`Quota`	TOMZO	1.2.3
cpan-mac (связка)	CNANDOR	0.40
`Mac::AppleEvents::Simple`	CNANDOR	0.81
`Mac::Glue`	CNANDOR	0.58

Источники подробной информации

Страницы руководства по *perlport* – просто неоценимый источник информации для Perl-программистов о различиях платформ. Крис Нандор (Chris Nandor) и Гурасами Сарати (Gurasamy Sarathy) дискутировали по поводу первых версий этого руководства на Perl Conference 2.0; Нандор опубликовал некоторые выдержки из этой беседы на *http://pudge.net/macperl/tpc/98*.

- *Информация о пользователях в Unix*
- *Информация о пользователях в Windows NT/2000*
- *Создание системы учетных записей для работы с пользователями*
- *Информация о модулях из этой главы*
- *Рекомендуемая дополнительная литература*

3

Учетные записи пользователей

Попробуйте выбрать правильный ответ на следующий вопрос. Если бы не было пользователей, то системные администраторы:

a) Были бы добрее.

b) Исчезли бы с лица земли.

Независимо от того, что говорят по этому поводу системные администраторы в «трудную минуту», b) – лучший ответ на этот вопрос. Как я уже говорил в главе 1, в конце концов, системное администрирование нужно для того, чтобы сделать возможным применение доступных технологий.

Откуда тогда все это ворчание? Пользователи вносят в системы и сети, которые мы администрируем, две вещи, значительно их усложняющие: неопределенность и индивидуальность. О неопределенности мы поговорим в следующей главе при обсуждении активности пользователей, а пока обратимся к индивидуальности.

В большинстве случаев пользователи хотят сохранять собственную индивидуальность. И дело даже не в том, что они предпочитают иметь уникальные имена, – они рассчитывают иметь и свое личное «имущество». Они хотят, чтобы у них была возможность сказать: «Это *мои* файлы. Храню я их в *своих* каталогах. Печатаю их на *моей* принтерной квоте. И я могу сделать их доступными на *моей* домашней странице в Сети». Современные операционные системы ведут учет всей этой информации для каждого пользователя.

Но кто следит за всеми учетными записями в системе или в сети? Кто ответственен за их создание, защиту и распределение? Рискну предпо-

ложить и скажу «*вы*, дорогой читатель». Ну а если не вы лично, то утилиты, которые вы создаете, и которые действуют от вашего имени в качестве посредников. Эта глава призвана помочь читателю справиться с такой ответственностью.

Начнем разговор о пользователях с рассмотрения информации, которая формирует их индивидуальность, и способов хранения этой информации в системе. Начнем мы с пользователей операционной системы всех вариантов Unix, а затем рассмотрим те же проблемы в Windows NT/2000. Для текущих версий MacOS такая проблема не актуальна, поэтому в данной главе MacOS упоминаться не будет. Познакомившись с этой информацией для обеих операционных систем, мы постараемся создать систему ведения учетных записей.

Информация о пользователях в Unix

При обсуждении этой темы мы будем иметь дело лишь с несколькими ключевыми файлами, поскольку в них хранится постоянная информация о пользователе. Говоря «постоянная», я имею в виду атрибуты, которые существуют до тех пор, пока существует пользователь, сохраняясь даже тогда, когда пользователь не зарегистрирован в системе. Иначе я буду называть это *учетной записью*. Если в системе у вас есть учетная запись, вы можете зарегистрироваться и стать пользователем данной системы.

Существование пользователя в системе начинается с того момента, когда информация о нем впервые заносится в файл паролей (или же служба каталогов обеспечивает аналогичную информацию). Пользователь «уходит со сцены», когда эта запись удаляется. Рассмотрим подробнее, как хранится эта информация.

Классический файл паролей в Unix

Начнем мы с «классического» формата файла паролей, а затем перейдем к более сложным вопросам. Я называю данный формат классическим, потому что на нем основаны все существующие в настоящее время форматы файлов паролей в Unix. Более того, он и сейчас встречается во многих вариантах Unix, включая SunOS, Digital Unix и Linux. Обычно это файл */etc/passwd*, содержащий последовательность текстовых ASCII-строк, причем каждая строка соответствует одной учетной записи или является ссылкой на другую службу каталогов. Любая строка файла состоит из нескольких полей, разделенных двоеточиями. Мы внимательно рассмотрим все эти поля, после того как научимся их получать.

Вот пример строки из файла */etc/passwd*:

```
dnb:fMP.olmno4jGA6:6700:520:David N. Blank-Edelman:/home/dnb:/bin/zsh
```

Существует по крайней мере два способа получать подобную информацию средствами Perl:

1. К файлу можно обратиться «вручную», рассматривать его как обычный текстовый и соответствующим образом анализировать:

```
$passwd = "/etc/passwd";
open(PW,$passwd) or die "Невозможно открыть $passwd:$!\n";
while (<PW>){
    ($name,$passwd,$uid,$gid,$gcos,$dir,$shell) = split(/:/);
    <далее следует ваша программа>
}
close(PW);
```

2. Другой способ позволяет «предоставить все полномочия системе». В этом случае нам будут доступны некоторые библиотечные вызовы Unix, которые проанализируют файл за нас. Тогда последний пример можно переписать так:

```
while(($name,$passwd,$uid,$gid,$gcos,$dir,$shell) = getpwent()){
      <далее следует ваша программа>
}
endpwent();
```

Употребление системных вызовов содержит еще одно преимущество: их можно автоматически использовать с любой из служб имен (например, NIS). Вскоре мы рассмотрим и другие системные вызовы (включая более простой способ применения `getpwent()`), а пока разберемся с полями, которые получили в наших примерах:[1]

Имя

В этом поле хранится короткое (обычно не длиннее восьми символов), уникальное в пределах системы регистрационное имя пользователя. Функция `getpwent()`, которую мы уже видели в предыдущем примере в списочном контексте, возвращает значения данного поля, если вызывается в скалярном контексте:

```
$name = getpwent();
```

Идентификатор пользователя (UID)

В Unix-системах идентификатор пользователя (UID) зачастую более важен, чем регистрационное имя. Все файлы в системе принадлежат пользователю с каким-либо идентификатором, а не регистрационным именем. Если в файле */etc/passwd* поменять регистраци-

[1] Значения, возвращаемые функцией `getpwent()`, в Perl 5.004 и 5.005 отличаются. Здесь мы приводим список переменных из 5.004. В версии 5.005 появляются два дополнительных поля `$quota` и `$comment` перед полем `$gcos`. За более подробной информацией о функции `getpwent()` обратитесь к системной документации. – *Примеч. науч. ред.*

онное имя пользователя, обладающего идентификатором 2397, с *danielr* на *drinehart*, то мгновенно владельцем всех его файлов станет пользователь *drinehart*. Для операционной системы идентификатор пользователя – постоянная информация. При выделении ресурсов и выяснении прав ядро и файловые системы следят за идентификаторами, а не регистрационными именами. Регистрационное имя можно считать информацией, внешней для операционной системы; эта информация существует, чтобы упростить жизнь пользователя.

Вот простой пример, позволяющий установить очередной доступный уникальный идентификатор в файле паролей. Достаточно выяснить максимальный идентификатор и использовать его для создания следующего номера:

```
$passwd = "/etc/passwd";
open(PW,$passwd) or die "Невозможно открыть $passwd:$!\n";
while (<PW>){
    @fields = split(/:/);
    $highestuid = ($highestuid < $fields[2]) ? $fields[2] : $highestuid;
}
close(PW);
print "Следующий доступный идентификатор: " . ++$highestuid . "\n";
```

Ниже перечислены другие полезные функции и переменные, имеющие отношение к именам и идентификаторам пользователей (табл. 3.1).

Таблица 3.1. Переменные и функции, имеющие отношение к именам и идентификаторам пользователей

Функция/ Переменная	Использование
`getpwnam($name)`	В скалярном контексте возвращает идентификатор, соответствующий этому регистрационному имени; в списочном контексте возвращает все поля данной записи из файла паролей
`getpwuid($uid)`	В скалярном контексте возвращает регистрационное имя, соответствующее данному идентификатору; в списочном контексте возвращает все поля данной записи из файла паролей
`$>`	Соответствует эффективному идентификатору пользователя текущей выполняющейся программы на Perl
`$<`	Соответствует реальному идентификатору пользователя текущей выполняющейся программы на Perl

Идентификатор первичной группы (GID)

В многопользовательских системах пользователи их группы часто работают с файлами и другими ресурсами совместно. В Unix существует механизм, позволяющий работать с группами пользователей. Учетная запись в системе может входить в несколько групп, но при этом принадлежать она должна только одной главной группе

(primary group). Поле GID в файле паролей соответствует как раз первичной группе для данной учетной записи.

Имена групп, их идентификаторы и члены группы обычно перечислены в файле */etc/group*. Чтобы включить учетную запись в несколько групп, необходимо просто указать ее в нескольких местах данного файла. В некоторых операционных системах существует жесткое ограничение на число групп, которым может принадлежать учетная запись (а значит, и пользователь). Чаще всего ограничение равно 8. Вот пример пары строк из файла */etc/group*:

```
bin::2:root,bin,daemon
sys::3:root,bin,sys,adm
```

Первое поле – это имя группы, второе – пароль (в некоторых системах может употребляться пароль для присоединения к группе), третье – идентификатор группы и последнее поле – список пользователей в группе.

Способы объединения пользователей в группы зависят от конкретного узла, поскольку границы (и административные, и границы проектов) везде разные. Таким образом, группы можно создавать для разделения различных пользователей (студенты, продавцы и т. д.), по выполняемым действиям (операторы резервных копий, сетевые администраторы и т. д.) либо по назначению учетных записей (резервные учетные записи и пр.).

Работа с файлами групп средствами Perl очень похожа на процесс разбора файла *passwd* из предыдущих примеров. Его можно считать обычным текстовым файлом либо применять специальные функции для выполнения подобной задачи. Посмотрите на функции и переменные, имеющие отношение к группам (табл. 3.2).

Таблица 3.2. Переменные и функции, имеющие отношение к именам и идентификаторам групп

Функция/ Переменная	Используется
`getgrent()`	В скалярном контексте возвращает имя группы; в списочном контексте возвращает поля: `$name`, `$passwd`, `$gid`, `$members`
`getgrnam($name)`	В скалярном контексте возвращает идентификатор группы; в списочном контексте возвращает те же поля, что и функция `getgrent()`
`getgrgid($gid)`	В скалярном контексте возвращает имя группы; в списочном контексте возвращает те же поля, что и функция `getgrent()`
`$)`	Соответствует эффективному идентификатору группы текущей выполняемой программы
`$(`	Соответствует реальному идентификатору группы текущей выполняемой программы

«Зашифрованный» пароль

Мы уже рассмотрели три основных поля, в которых содержится информация о пользователе в Unix. Следующее поле не является частью хранимой информации, но оно подтверждает права, обязанности и привилегии, присущие пользователю с конкретным идентификатором. Именно так компьютер узнает, что тому, кто выдает себя за пользователя *mguerre*, позволено присвоить конкретный идентификатор. Существуют и другие, лучшие формы авторизации (например, использование криптографических методов с открытым ключом), но этот способ унаследован от ранних версий Unix.

Очень часто в этом поле в файле паролей можно увидеть лишь звездочку (*). Подобный знак применяется для запрещения регистрации пользователя в системе, без удаления при этом самой учетной записи.

Работа с паролями пользователей – это отдельная тема. Ей будет посвящена глава 10 «Безопасность и наблюдение за сетью».

Поле GCOS

Поле GCOS[1] самое бесполезное (с точки зрения компьютера). Обычно в этом поле записано полное имя пользователя (например «Рой Дж. Бив»). Часто люди добавляют туда должность и/или номер телефона.

Системные администраторы, заботящиеся о приватности пользователей (чему и следует быть), должны проверять содержимое данного поля. Это стандартный путь для определения соответствия между реальным именем пользователя и его регистрационным именем. В большинстве Unix-систем такое поле находится в файле */etc/passwd*, доступном всем для чтения, следовательно, эта информация может попасть в руки кого угодно в системе. Многие программы, почтовые клиенты и демоны finger-запросов обращаются к этому полю при добавлении регистрационного имени пользователя к какой-то информации. Если у вас есть необходимость скрыть реальные имена пользователей от других людей (например, если речь идет о политических диссидентах, федеральных свидетелях или известных персонах), вы обязательно должны следить за этим полем.

В качестве дополнительной информации: если вы поддерживаете сайт с менее развитой пользовательской базой, было бы неплохо запретить пользователям изменять их поле GCOS на случайные строки (по тем же причинам, по которым выбранные пользователями регистрационные имена могут вызвать проблемы). Вряд ли вы придете в восторг, увидев в своем файле паролей бранные выражения или иную непрофессиональную информацию.

---

[1] Занятные подробности о происхождении названия этого поля можно найти в словаре компьютерного жаргона *http://www.jargon.org*.

Домашний каталог

Следующее поле содержит имя *домашнего каталога* пользователя. Это тот каталог, откуда начинается работа с системой. Обычно здесь хранятся файлы, определяющие настройки пользователя.

В целях безопасности очень важно, чтобы запись в домашний каталог была разрешена только его владельцу. Домашние каталоги, доступные для записи всем, открывают возможность хакерских действий. Правда, существуют ситуации, когда домашние каталоги, доступные для записи только самим владельцам, тоже вызывают проблемы. Например, в случае с ограниченными интерпретаторами (если пользователи могут регистрироваться в системе только для выполнения определенных задач без права изменять что-либо в системе) домашние каталоги, доступные для записи пользователю, категорически запрещены.

Вот пример кода на Perl, который позволяет убедиться, что все домашние каталоги пользователей принадлежат своим владельцам и недоступны для записи остальным:

```
use User::pwent;
use File::stat;

# замечание: этот код очень сильно загрузит машину, если
# домашние каталоги монтируются автоматически
while($pwent = getpwent()){
    # убеждаемся, что это действительно каталог, даже если
    # он спрятан за символическими ссылками
    $dirinfo = stat($pwent->dir."/.");
    unless (defined $dirinfo){
        warn "Невозможно получить информацию о ".$pwent->dir.": $!\n";
        next;
    }
    warn «Домашний каталог пользователя ".$pwent->name." не имеет
владельца с корректным uid (". $dirinfo->uid." вместо ".$pwent-
>uid.")!\n"
        if ($dirinfo->uid != $pwent->uid);

    # каталог может быть доступным всем для записи, если
    # у него установлен «бит-липучка" (т. е. 01000),
    # подробности в странице руководства по chmod
    warn $pwent->name."'s homedir is world-writable!\n"
      if ($dirinfo->mode & 022 and (!$stat->mode & 01000));
}
endpwent();
```

Этот пример на вид несколько отличается от предыдущих, поскольку в нем используются два замысловатых модуля Тома Кристиансена (Tom Christiansen): `User::pwent` и `File::stat`. Эти модули изменяют функции `getpwent()` и `stat()`, заставляя их возвращать значения, отличные от ранее упомянутых. Когда загружены модули `User::pwent`

и `File::stat`, эти функции возвращают объекты вместо списков или скалярных значений. У каждого объекта есть метод, названный по имени поля, которое было бы возвращено в списочном контексте. Поэтому такой код:

```
$gid = (stat("filename"))[5];
```

можно переписать гораздо понятнее :

```
use File::stat;
$stat = stat("filename");
$gid = $stat->gid;
```

или даже так:

```
use File::stat;
$gid = stat("filename")->gid;
```

Командный интерпретатор пользователя

Последнее поле классического файла паролей – это поле, соответствующее командному интерпретатору пользователя. Обычно это один из интерпретаторов (*sh*, *csh*, *tcsh*, *ksh*, *zsh*), но это может быть и путь к любой исполняемой программе или сценарию. Время от времени, некоторые ради шутки (но наполовину всерьез) устанавливают в качестве своего командного интерпретатора по умолчанию интерпретатор Perl. По крайней мере, в один интерпретатор (*zsh*) хотят всерьез встроить интерпретатор Perl, но этого пока еще не случилось. Тем не менее, были предприняты серьезные попытки создать командный интерпретатор Perl shell (*http://www.focusresearch.com/gregor/psh/*), а также встроить Perl в редактор Emacs, который легко может заменить целую операционную систему (*http://john-edwin-tobey.org/perlmacs/*).

Бывают ситуации, когда необходимо указать в этом поле нечто отличное от стандартного командного интерпретатора. Например, если вы хотите создать учетную запись, работающую с системой меню, вы можете поместить в данное поле имя такой программы. В этом случае стоит принять некоторые меры предосторожности, чтобы пользователь, применяющий эту учетную запись, не получил бы доступ к командному интерпретатору, иначе не миновать бед. Часто встречаемая ошибка – включение в такое меню почтовой программы, которая позволяет запускать редактор или инструмент постраничного просмотра для чтения или редактирования почты. Оба эти средства могут иметь возможность выхода в интерпретатор.

Список доступных в системе стандартных командных интерпретаторов часто хранится в файле */etc/shells*, видимо, для удобства демона FTP. Большинство FTP-демонов не позволят обычному пользователю подсоединиться к системе, если их командный интерпретатор, заданный в */etc/passwd* (или сетевом файле паролей), не при-

сутствует в */etc/shells*. Вот пример на Perl, который докладывает об учетных записях с неподтвержденными командными интерпретаторами:

```
use User::pwent;

$shells = "/etc/shells";
open (SHELLS,$shells) or die "Невозможно открыть $shells:$!\n";
while(<SHELLS>){
    chomp;
    $okshell{$_}++;
}
close(SHELLS);

while($pwent = getpwent()){
   warn $pwent->name." has a bad shell (".$pwent->shell.")!\n"
     unless (exists $okshell{$pwent->shell});
}
endpwent();
```

Дополнительные поля в файлах паролей в BSD 4.4

При смене BSD (Berkeley Software Distribution) с версии 4.3 на 4.4 к классическому формату файла паролей были добавлены две характерные особенности: дополнительные поля и формат двоичных баз данных, используемых для хранения информации об учетных записях.

В BSD 4.4 в файле паролей между полями GID и GCOS появились новые поля. Первым было добавлено поле *class*. Оно позволяет системному администратору разбить все учетные записи системы на отдельные классы (например, для различных классов учетных записей могут существовать различные ограничения ресурсов, таких как время использования процессора). Кроме того, были добавлены поля *change* и *expire*, в которых хранятся данные о сроке продолжительности пароля и времени действия учетной записи. Подобные поля встретятся также и в формате следующего файла паролей в Unix.

При компиляции в операционной системе, поддерживающей эти дополнительные поля, Perl включает их содержимое в значение, возвращаемое функциями типа `getpwent()`. Это одна из причин, по которой стоит употреблять в программах `getpwent()`, а не разбирать файл паролей вручную при помощи `split()`.

Формат двоичных баз данных в BSD 4.4

Вторая характерная черта BSD – использование баз данных, а не обычного текста для хранения информации о паролях. В BSD файлы паролей хранятся в формате DB – значительно улучшенной версии старых библиотек DBM (Database Management). Это изменение позволяет системе быстро обращаться к информации о паролях.

Программа *pwd_mkdb* в качестве аргумента принимает имя текстового файла паролей, создает и переносит в нужное место два файла баз данных, а затем перемещает исходный текстовый файл в */etc/master.passwd*. Две базы данных позволяют обеспечить механизм теневых паролей – они отличаются правами на чтение и содержимым поля, в котором хранится зашифрованный пароль. Более подробно мы поговорим об этом в следующем разделе.

Perl может напрямую работать с DB-файлами (операции с самим форматом встречаются в главе 9 «Журналы»), но обычно я не рекомендую напрямую редактировать базы данных, пока система используется. Дело тут в блокировке: необходимо убедиться, что другие программы не читают и не записывают данные в файл паролей в тот момент, когда вы собираетесь редактировать его как базу данных. Стандартные программы, подобные *chpasswd*, выполняют необходимую блокировку самостоятельно.[1] Ловкий прием, заключающийся в использовании переменной EDITOR, который мы употребили при работе с квотами в главе 2 «Файловые системы», можно применить и при вызове *chpasswd*.

Теневые пароли

Не следует забывать, насколько важна защита содержимого поля GCOS, т. к. целым рядом различных механизмов эта информация доступна для всех. Другая, менее доступная, но довольно уязвимая информация – это список зашифрованных паролей всех пользователей системы. И хотя эти пароли зашифрованы, одно то, что они хранятся в файле, доступном всем для чтения, вносит изрядную степень риска. Некоторые части файла паролей должны быть доступны всем для чтения (например, связь между регистрационным именем и идентификатором пользователя), но не весь файл. Нет никакой необходимости выставлять на всеобщее обозрение список зашифрованных паролей, т. к. пользователи могут попытаться запустить программы для взлома паролей.

Одна из возможных альтернатив – перенести все пароли в специальный файл, читать который сможет только суперпользователь. Этот второй файл известен как файл «теневых паролей», т. к. в нем хранятся строки, затеняющие записи из обычного файла паролей.

Вот как все это работает: оригинальный файл паролей остается нетронутым, за одним небольшим исключением. Вместо зашифрованного пароля в это поле помещается специальный символ или символы, которые говорят о том, что используется механизм затенения паролей. Обычно это символ `x`, но в BSD используется `*`.

[1] Программа *pwd_mkdb* может выполнить блокировку, а может и не выполнить (это зависит от версии и типа BSD), поэтому будьте осторожны.

Я слышал, что существуют некоторые пакеты (для поддержки теневых паролей), вставляющие в это поле специальную строку символов, которая выглядит обычной. Если ваш файл паролей попадет в руки злоумышленника, то он потратит много времени, взламывая случайные строки, не имеющие ничего общего с настоящими паролями.

Большинство операционных систем используют файл теневых паролей для хранения дополнительной информации об учетной записи. Такой формат включает дополнительные поля, которые мы видели в BSD-файлах, и в них хранится информация об истечении срока действия учетной записи и информация о смене пароля.

В большинстве случаев обычные Perl-функции, подобные `getpwent()`, могут работать с файлами теневых паролей. Если стандартные библиотеки C, входящие в состав операционной системы, делают то, что нужно, то Perl тоже будет делать все верно. Говоря «делать то, что нужно», я подразумеваю, что если ваши сценарии на Perl запускаются с подходящими привилегиями (с привилегиями суперпользователя), то эти функции будут возвращать зашифрованный пароль. В остальных случаях пароль этим функциям не доступен.

Значительно хуже, если вы захотите получить дополнительные поля из файла теневых паролей. Perl может и не вернуть их вам. Эрик Истабрукс (Eric Estabrooks) написал модуль `Passwd::Solaris`, но он будет полезен только при работе в Solaris. Если эти поля имеют для вас принципиальное значение или вы хотите действовать наверняка, то, как ни грустно (это противоречит моим рекомендациям использовать `getpwent()`), но часто проще открыть файл *shadow* и получить нужные значения вручную.

Информация о пользователях в Windows NT/2000

Теперь, когда мы выяснили, как образована информация о пользователях в Unix-системах, мы можем посмотреть, как это делается в NT/2000. Большая часть этой информации сходна с уже рассмотренной, поэтому следует обратить внимание на различия между двумя операционными системами.

Хранение и доступ к информации о пользователях в Windows NT/2000

NT/2000 хранит постоянную информацию о пользователях в базе данных *SAM* (Security Accounts Manager, диспетчер учетных записей в системе защиты). База данных SAM – это часть реестра NT/2000, находящаяся в *%SYSTEMROOT%/system32/config*. Файлы, входящие в состав реестра, хранятся в двоичном формате, следовательно, обыч-

ные функции для работы с текстом в Perl нельзя применять для чтения или внесения изменений в эту базу данных. Теоретически, если NT/2000 не запущена, можно использовать операторы над двоичными данными (`pack()` и `unpack()`) для работы с SAM, но такой способ безумен и мучителен.

К счастью, существуют более удачные методы доступа к этой информации и работы с ней в Perl.

Один из способов – вызвать внешнюю программу, которая обесепечит ваше взаимодействие с операционной системой. На каждой машине с NT/2000 есть команда *net*, она позволяет добавлять, удалять и просматривать данные о пользователях. *net* довольно странная и ограниченная команда и, вероятно, к ее использованию стоит прибегать в крайнем случае.

Вот так, например, команда *net* выполняется на машине с двумя учетными записями:

```
C:\>net users

User accounts for \\HOTDIGGITYDOG
-----------------------------------
Administrator            Guest
The command completed successfully.
```

При необходимости, вывод этой команды было бы просто разобрать из Perl. Помимо *net* существуют и другие коммерческие пакеты, в состав которых входят программы, запускаемые из командной строки и выполняющие те же действия.

Другой подход – использовать модуль `Win32::NetAdmin` (входящий в состав дистрибутива ActiveState Perl) или один из модулей, созданных для расширения функциональности `Win32::NetAdmin`. В их число входят модули `Win32::AdminMisc` Дэвида Рота (David Roth, модуль находится на *http://www.roth.net*) и `Win32::UserAdmin` (описанный Эшли Мэггитом (Ashley Meggitt) и Тимоти Ритчи (Timothy Ritchey) в книге «Windows NT User Administration» (Windows NT: Администрирование пользователей), модуль можно найти на *ftp://ftp.oreilly.com/pub/examples/windows/winuser/*).

Для выполнения большинства операций с пользователями я предпочитаю модуль `Win32::AdminMisc`, поскольку он предлагает множество инструментов системного администрирования, кроме того, Рот активно поддерживает его в нескольких форумах. И хотя доступная в Сети документация по этому модулю очень хороша, лучшая документация – это книга самого автора «Win32 Perl Programming: The Standard Extensions» (Программирование на Perl для Win32: стандартные расширения) (Macmillan Technical Publishing). Такую книгу всегда полезно иметь под рукой, если вы собираетесь писать на Perl программы для Win32.

Приведу пример, перечисляющий пользователей на локальной машине, а также некоторые сведения об этих пользователях. Выводимые в примере строки похожи на строки из файла */etc/passwd* в Unix:

```
use Win32::AdminMisc;

# получаем список всех локальных пользователей
Win32::AdminMisc::GetUsers('','',\@users) or
   die "Невозможно получить список пользователей: $!\n";

# получаем их атрибуты и выводим их
foreach $user (@users){
  Win32::AdminMisc::UserGetMiscAttributes('',$user,\%attribs)
    or warn " Невозможно получить атрибуты: $!\n";
  print join(":",$user,
                 '*',
                 $attribs{USER_USER_ID},
                 $attribs{USER_PRIMARY_GROUP_ID},
                 '',
                 $attribs{USER_COMMENT},
                 $attribs{USER_FULL_NAME},
                 $attribs{USER_HOME_DIR_DRIVE}."\\".
                 $attribs{USER_HOME_DIR},
                 ''),"\n";
}
```

Наконец, вы можете использовать модуль `Win32::OLE` для доступа к интерфейсам активных служб каталогов (ADSI, Active Directory Service Interfaces). Данная служба встроена в Windows 2000 и ее можно установить на Windows NT 4.0. Эта тема и соответствующие примеры будут подробно рассмотрены в главе 6 «Службы каталогов».

Другие примеры программ на Perl для работы с пользователями в NT/2000 встретятся позже, а пока вернемся к обсуждению различий между пользователями в Unix и Windows NT/2000.

Идентификаторы пользователей в NT/2000

Идентификаторы пользователей в NT/2000 создаются не простыми смертными и их нельзя использовать повторно. В отличие от Unix, где мы просто берем следующий свободный идентификатор пользователя, в Windows NT/2000 операционная система уникальным образом генерирует эквивалентный идентификатор каждый раз при создании пользователя. Уникальный идентификатор пользователя (который в NT/2000 называется *относительным идентификатором* или RID, Relative ID) объединяется с идентификатором машины и домена, и вместе они образуют длинный идентификационный номер – *идентификатор безопасности* (SID, Security ID), который используется в качестве идентификатора пользователя (UID). Например, RID равный `500`, является частью длинного идентификатора SID, который выглядит так:

```
S-1-5-21-2046255566-1111630368-2110791508-500
```

RID – это то число, которое мы получаем в результате вызова функции `UserGetMiscAttributes()` в последнем примере. Вот так должен выглядеть код для получения идентификатора RID конкретного пользователя:

```
use Win32::AdminMisc;

Win32::AdminMisc::UserGetMiscAttributes('',$user,\%attribs);
print $attribs{USER_USER_ID},"\n";
```

Вы не сможете (каким бы то ни было нормальным способом) заново создать пользователя после того, как он был удален. И даже если вы создадите пользователя с тем же самым именем, его идентификатор безопасности (SID) все равно будет отличаться. У нового пользователя не будет доступа к файлам и ресурсам его предшественника.

По этой причине в некоторых книгах по NT рекомендуется переименовывать учетные записи, которые наследуются от других людей. Если к новому работнику должны перейти все файлы и привилегии уходящего работника, следует скорее переименовать существующую учетную запись, чтобы сохранить SID, чем создавать новую учетную запись, переписывать все файлы и затем удалять старую. Лично я нахожу такой способ передачи учетных записей несколько грубоватым, поскольку в этом случае новый работник наследует все поврежденные и бесполезные настройки реестра от своего предшественника. Но это самый удобный способ, а иногда это важно.

Частично эта рекомендация связана с мучениями при передаче права собственности на файлы. В Unix привилегированный пользователь может сказать: «Изменить права владения для всех этих файлов так, чтобы они перешли к новому пользователю». В NT, однако, право на владение нельзя дать, его можно только получить. К счастью, существует два способа обойти это ограничение и считать, что мы используем семантику Unix. В Perl мы можем:

- Вызвать исполняемый файл, включая:
  - Программу *chown* либо из пакета Microsoft NT Resource (коммерческий продукт, упомянутый далее), либо из дистрибутива Cygwin с *http://www.cygnus.com* (бесплатный).
  - Программу *setowner*, входящую в число утилит NTSEC, продаваемых Pedestal Software на *http://www.pedestalsoftware.com*. Я предпочитаю ее, т. к. программа отличается гибкостью и при этом требует наименьших затрат.
- Использовать модуль `Win32::Perms`, написанный Дэвидом Ротом (David Roth), который можно найти на *http://www.roth.net/perl/perms*. Вот простой пример, изменяющий владельца каталога и его содержимое, включая подкаталоги:

```
use Win32::Perms;

$acl  = new Win32::Perms();
$acl->Owner($NewAccountName);
$result = $acl->SetRecurse($dir);
$acl->Close();
```

Пароли в NT/2000

Алгоритмы, применяемые для шифрования паролей, ограничивающих доступ к владениям пользователей в NT/2000 и Unix, криптографически несовместимы. Вы не можете передавать зашифрованные пароли из одной операционной системы в другую, что бывает необходимо для смены пароля или создания учетных записей. В результате, два набора паролей приходится использовать и/или хранить синхронно. Это различие – просто проклятье всех системных администраторов, вынужденных управлять смешанным окружением Unix–NT/2000. Некоторые администраторы обходят эти проблемы, используя специальные модули авторизации – как коммерческие, так и прочие.

Если вы не применяете специальные механизмы авторизации, то единственное, что вы можете сделать как программист на Perl, – создать систему, благодаря которой пользователи смогут представлять свои пароли в виде обычного текста. Такие пароли позволяют выполнять связанные с ними операции (изменение пароля и пр.), различные в каждой операционной системе.

Группы в NT

До сих пор при обсуждении идентификации пользователя я не упоминал о различиях между хранением этой информации на локальной машине и средствами какой-либо сетевой службы, например NIS. Для той информации, о которой шла речь, не было существенно, используется ли она на одной системе, на всех системах в сети или в рабочей группе. Чтобы обоснованно говорить о группах пользователей в NT/2000 и их связи с Perl, мы должны, к сожалению, отойти от этого соглашения. Мы остановимся на группах в Windows NT 4.0. В Windows 2000 был добавлен еще один уровень сложности, поэтому информацию о группах в Windows 2000 я вынес во врезку «Изменения групп в Windows 2000», которую вы найдете в этой главе.

В NT информация о пользователях может храниться в одном из двух мест: в SAM на конкретной машине или в SAM на контроллере домена. Именно здесь проявляется различие между *локальным пользователем*, который может входить в систему и работать только на одной машине, и *пользователем домена*, который может регистрироваться на любой из машин в домене.

Группы в NT также бывают двух типов: *глобальные* и *локальные*. Разница между ними заключается не совсем в том, чего можно было бы ожидать из названия. Неверно, что первая состоит из пользователей домена, а вторая из локальных пользователей. Так же неверно и то, что один тип пользователей имеет доступ только на одну машину, в то время как другой пользуется всей сетью, как могли ожидать лица, знакомые с Unix. Частично верно и одно определение, и другое, но давайте рассмотрим все подробно.

Если начать с рассмотрения целей, лежащих за названием, и механизмом их реализации, станет немного яснее. Вот чего мы пытаемся достичь:

- Учетные записи пользователей всего домена должны обслуживаться централизованно. Администраторы должны иметь возможность определить произвольное подмножество прав и привилегий пользователей, которые можно присвоить всей группе одновременно.
- При желании, все машины домена должны иметь возможность использовать преимущества такого централизованного управления. Администратор отдельной машины по-прежнему должен иметь возможность создавать пользователей, «живущих» только на этой машине.
- Администратор каждой машины должен иметь возможность решать, каким пользователям разрешать доступ к этой машине. Администратор должен иметь возможность делать это, используя группы, существующие в домене, а не задавать имена пользователей вручную.
- Члены этих групп и локальные пользователи должны иметь возможность считаться равными с точки зрения администратора (речь идет о правах и прочем).

Глобальные и локальные группы позволяют нам добиться всего вышеперечисленного. В двух предложениях это можно объяснить так: в глобальные группы входят только пользователи домена. В локальные группы входят локальные пользователи, а также в них входят/импортируются пользователи глобальных групп.

Вот простой пример для объяснения, как все это работает. Скажем, у вас есть домен NT для кафедры в университете, в котором уже созданы пользователи домена – студенты, преподаватели и служащие. Когда появляется новый исследовательский проект под названием Omphaloskepsis, системные администраторы создают новую глобальную группу `Global-Omph People`. В данную глобальную группу входят все пользователи домена, занятые в этом проекте. Когда студенты и персонал присоединяются к проекту или выходят из него, они, соответственно, добавляются или удаляются из этой группы.

Для этого проекта используется отдельный компьютерный класс. На машинах такого класса созданы гостевые учетные записи для несколь-

ких представителей факультета с других кафедр (они не являются пользователями домена). Системный администратор этого класса делает следующее (разумеется, на Perl), чтобы запретить использовать компьютеры тем, кто не занят в этом проекте:

1. Создает на каждой машине *локальную* группу `Local-Authorized Omphies`.
2. Добавляет в эту локальную группу локальные гостевые учетные записи.
3. Добавляет *глобальную* группу `Global-Omph People` в указанную локальную группу.
4. Добавляет право (права пользователей мы обсудим в следующем разделе) `Log on Locally` (регистрироваться локально) локальной группе `Local-Authorized Omphies`.
5. Удаляет право `Log on Locally` для всех остальных неавторизованных групп.

В результате только авторизованные локальные пользователи и пользователи из авторизованных глобальных групп могут регистрироваться на машинах этого класса. Как только в группу `Global-Omph People` добавляется новый пользователь, он автоматически получает право регистрироваться на этих машинах без каких-либо изменений учета. Как только вы освоитесь с понятием локальных/глобальных групп, такая схема покажется вам удобной.[1]

Подобная схема была бы совсем удобной, если бы не усложняла программирование на Perl. Во всех упомянутых модулях существуют отдельные функции для локальных и глобальных групп. Например, в `Win32::NetAdmin` имеем:

`GroupCreate()`	`LocalGroupCreate()`
`GroupDelete()`	`LocalGroupDelete()`
`GroupGetAttributes()`	`LocalGroupGetAttributes()`
`GroupSetAttributes()`	`LocalGroupSetAttributes()`
`GroupAddUsers()`	`LocalGroupAddUsers()`
`GroupDeleteUsers()`	`LocalGroupDeleteUsers()`
`GroupIsMember()`	`LocalGroupIsMember()`
`GroupGetMembers()`	`LocalGroupGetMembers()`

[1] Для тех, кто работает в Unix, но все еще читает этот раздел, скажем, что подобного можно достичь, используя сетевые группы NIS и специальные записи в файле */etc/passwd* на каждой машине домена NIS. Подробности можно найти на страницах руководства по *netgroup*.

Отличия групп в Windows 2000

Практически все, что мы говорили о локальных и глобальных группах в NT, также относится и к Windows 2000, но существует несколько характерных особенностей, о которых необходимо упомянуть:

1. В Windows 2000 используются активные каталоги (Active Directory, более подробную информацию о них можно найти в главе 6) для хранения информации о пользователях. Это означает, что информация о глобальных группах теперь хранится в активном каталоге на контроллере домена, а не в его SAM.
2. Локальные группы теперь называются *локальными группами домена*.
3. Была добавлена третья, *пространственная* (*scope*) группа. Помимо глобальных и локальных групп домена в Windows 2000 добавились *универсальные* группы. Универсальные группы, по существу, разрывают границы домена. В них могут входить учетные записи, глобальные группы и универсальные группы из любого места каталога. В локальные группы домена могут входить как глобальные группы, так и универсальные группы.

На момент написания этой книги стандартные модули Perl для администрирования учетных записей еще не учитывали таких изменений. Эти модули можно по-прежнему использовать, поскольку интерфейсы NT4 SAM пока еще действуют, но они не смогут применять новые возможности. Поэтому данная врезка – единственное место, где мы упоминаем об этих различиях в Windows 2000. Подробную информацию вам придется искать в описании интерфейсов служб активных каталогов (Active Directory Service Interfaces, ADSI), о которых речь пойдет в главе 6.

Таким образом, не исключено, что в программах для выполнения одной и той же операции понадобится употребить две функции. Например, если нужно получить список всех групп, в которые может входить пользователь, придется вызвать две функции – одну для локальных, а другую для глобальных групп. Приведенные функции характеризуются своими названиями. Детальное описание можно найти в документации и книге Рота.

Вот короткий совет из книги Рота: чтобы получить список локальных групп, ваша программа должна выполняться с привилегиями администратора, но имена глобальных групп должны быть доступны всем пользователям.

Права пользователей в NT/2000

Последнее различие между информацией о пользователях в Unix и NT/2000, о котором мы поговорим, – это понятие «пользовательских прав». В Unix действия, предпринимаемые пользователем, ограничиваются как правами доступа к файлам, так и различиями между суперпользователем и остальными пользователями. В NT/2000 права реализованы гораздо сложнее. Пользователи (и группы) могут быть наделены особой силой, которая становится частью информации о пользователях. Например, если предоставить обычному пользователю право `Change the System Time` (Изменение системного времени), то он сможет изменять настройки системных часов.

Некоторые считают, что такая концепция прав пользователей сбивает с толку, поскольку они предпринимали попытки прибегнуть к помощи отвратительного диалогового окна User Rights Policy (Политика прав пользователей) из NT 4.0 в приложениях *User Manager* (Диспетчер пользователей) или *User Manager for Domains* (Диспетчер пользователей для доменов). В этом диалоговом окне информация представлена виде, прямо противоположном тому, в котором большинство пользователей ожидают ее там увидеть. Она содержит перечень возможных прав пользователей и предлагает добавить группы или пользователей к списку тех, у кого такие права уже есть. Вот как выглядит это диалоговое окно (рис. 3.1) пользовательских прав в действии.

Было бы лучше, если бы права пользователей разрешалось добавлять и удалять, а не наоборот. В действительности, именно так мы и будем поступать, используя Perl.

Один из возможных подходов – вызвать программу *ntrights.exe* из Microsoft *NT Resource Kit*. Если вы об этом никогда не слышали, обязательно прочитайте следующую врезку.

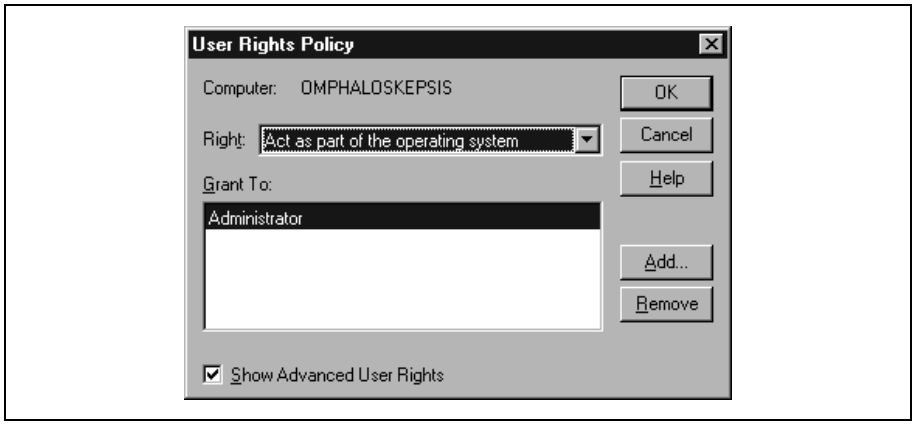

Рис 3.1. *Диалоговое окно User Rights Policy Диспетчера пользователей в NT 4.0*

Работать с *ntrights.exe* очень легко; достаточно вызывать эту программу из Perl, как любую другую (т. е., применяя обратные кавычки или функцию `system()`). В этом случае мы обратимся к *ntrights.exe* при помощи такой командной строки:

```
C:\>ntrights.exe +r <right name> +u <user or group name> [-m \\machinename]
```

чтобы предоставить право пользователю или группе (на машине *machinename*, имя которой указывать не обязательно). Чтобы отнять право, необходимо применить такой синтаксис:

```
C:\>ntrights.exe -r <right name> +u <user or group name> [-m \\machinename]
```

Пользователи Unix знакомы с употреблением символов + и – (как в *chmod*), в данном случае для ключа *–r*, чтобы предоставить или лишить привилегий. Список допустимых имен (например, `SetSystemtimePrivilege` для разрешения устанавливать системное время) можно найти в документации Microsoft *NT Resource Kit* по команде *ntrights*

Второй подход, с использованием только Perl, связан с применением модуля `Win32::Lanman`, написанного Йенсом Хелбергом (Jens Helberg), который можно найти либо на *ftp://ftp.roth.net/pub/ntperl/Others/Lanman/*, либо на *http://jenda.krynicky.cz*. Начнем с того, что рассмотрим процесс получения прав для учетной записи. Этот процесс состоит из нескольких шагов, поэтому рассмотрим его подробно, шаг за шагом.

Сначала необходимо загрузить модуль:

```
use Win32::Lanman;
```

Затем следует получить идентификатор (SID) для учетной записи, с которой надо работать. В следующем примере мы получим SID для учетной записи *Guest*:

```
unless(Win32::Lanman::LsaLookupNames($server, ['Guest'],\@info){
   die "Невозможно найти SID:
    ".Win32::Lanman::GetLastError()."\n";
}
```

`@info` – это массив ссылок на анонимные хэши, каждый элемент которого соответствует отдельной учетной записи (в нашем случае это один-единственный элемент для учетной записи *Guest*). В каждом хэше есть такие ключи: `domain`, `domainsid`, `relativeid`, `sid` и `use`. На следующем шаге нас будет интересовать только ключ `sid`. Теперь мы можем узнать о правах этой учетной записи:

```
unless (Win32::Lanman::LsaEnumerateAccountRights($server,
                  ${$info[0]}{sid}, \@rights)){
   die "Невозможно узнать права:
    ".Win32::Lanman::GetLastError()."\n";
}
```

Microsoft Windows NT/ Windows 2000 Resource Kits

«У вас должен быть установлен *NT 4.0 Server* и/или *Workstation Resource Kit*» – в этом, обычно, единодушны и серьезные администраторы NT, и средства информации. Microsoft Press опубликовал два больших тома, каждый из которых полон жизненно необходимой информации об одной из версий операционной системы NT/2000. Ценность этих книг заключается не столько в сведениях, сколько в компакт-дисках, распространяемых вместе с книгами. На компакт-дисках есть множество важных утилит для администрирования NT/2000. Утилиты, поставляемые с книгой по NT/2000 Server, содержат и утилиты, входящие в компакт-диск для версии NT Workstation/Windows 2000 Professional. Если вам придется выбирать одну из книг, предпочтите издание, посвященное NT/2000 Server.

Многие из этих утилит распространяются группой разработчиков NT/2000, написавших собственные программы, поскольку они нигде не смогли найти нужные им инструменты. Например, в состав этих утилит входят программы для добавления пользователей, изменения информации о безопасности файловой системы, отображения установленных драйверов принтеров, работы с профилями, помощи с отладкой служб доменов и обозревателя сети и т. д.

Инструменты из пакета дополнительных программ поставляются «как есть» (as is), иными словами, они практически не поддерживаются. Такая политика «неподдержки» может показаться грубой, но она преследует важную цель – дать возвожность Microsoft предоставить администраторам множество полезных программ и не заставлять их платить непомерно много за поддержку. В программах из этого пакета есть некоторые мелкие ошибки, но, в целом, они работают замечательно. Обновления, исправляющие ошибки в некоторых утилитах, публикуются на веб-сайте Microsoft.

Массив `@rights` теперь содержит набор строк, описывающих все права учетной записи *Guest*.

Узнать, чему соответствует API-имя (Application Program Interface, интерфейс прикладного программирования) того или иного права пользователя, может оказаться непростой задачей. Самый легкий способ выяснить, каким правам какие имена соответствуют, – ознакомиться с документацией SDK (Software Developement Kit, набор инструментальных средств разработки программного обеспечения), которая находится на *http://msdn.microsoft.com*. Нужную документацию отыскать легко, потому что Хелберг сохранил имена стандартных

функций SDK для функций в Perl. Чтобы найти имена доступных прав, достаточно поискать в MSDN (Microsoft's Developer Network) «LsaEnumerateAccountRights», и мы быстро их отыщем.

Подобная информация полезна и для изменения прав пользователей. Например, если мы хотим разрешить пользователю *Guest* выключать (останавливать) систему, мы можем применить следующее:

```
use Win32::Lanman;

unless (Win32::Lanman::LsaLookupNames($server, ['Guest'],
                                      \@info)) {
  die " Невозможно найти SID: ".Win32::Lanman::GetLastError()."\n";
}


unless (Win32::Lanman::LsaAddAccountRights($server,
            ${$info[0]}{sid}, [&SE_SHUTDOWN_NAME])) {
    die " Невозможно изменить права: ".Win32::Lanman::GetLastError()."\n"
}
```

На этот раз мы нашли право `SE_SHUTDOWN_NAME` в документации по SDK и применили подпрограмму `&SE_SHUTDOWN_NAME` (определенную в `Win32::Lanman`), возвращающую значение этой константы SDK.

`Win32::Lanman::LsaRemoveAccountRights()` – это функция. Она используется для лишения прав и принимает аргументы, схожие с теми, которые применяет функция для добавления прав.

Перед тем как перейти к другим темам, необходимо упомянуть, что в `Win32::Lanman` входит также и функция, действующая аналогично неудачному интерфейсу диспетчера пользователей, о котором мы говорили раньше. Вместо того чтобы сопоставлять пользователей с правами, мы можем сопоставлять права с пользователями. Применяя функцию `Win32::Lanman::LsaEnumerateAccountsWithUserRight()`, мы можем получить список идентификаторов (SID), у которых есть определенные поля. Иногда такое знание может сослужить добрую службу.

Создание системы учетных записей для работы с пользователями

Теперь, в достаточной мере познакомившись с информацией о пользователях, мы можем перейти к вопросу администрирования учетных записей. Вместо того чтобы просто привести список подпрограмм и функций Perl, небходимых для добавления и удаления пользователей, я хочу рассмотреть предмет разговора на другом уровне, рассказывая об этих операциях в широком контексте. В оставшейся части этой главы мы попытаемся написать скелет системы учетных записей, которая работает с пользователями как в NT, так и в Unix.

Наша система учетных записей будет состоять из четырех частей: пользовательского интерфейса, хранилища данных, сценариев обработки (в Microsoft это назвали бы «бизнес-логикой») и низкоуровневых библиотечных вызовов. В организации процесса они работают все вместе (рис. 3.2).

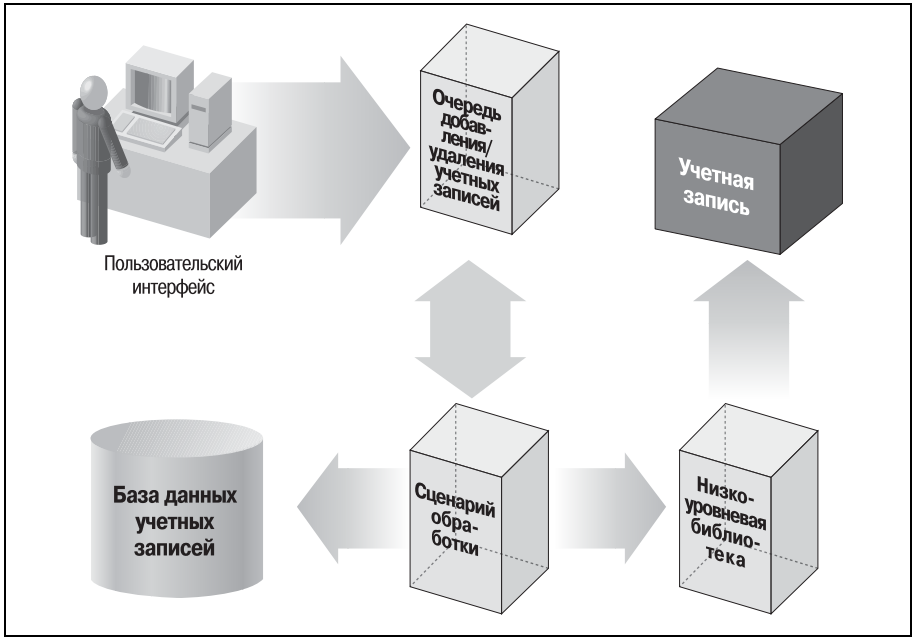

Рис. 3.2. *Структура простой системы учетных записей*

Запросы поступают в систему через пользовательский интерфейс и помещаются в файл «очереди добавления учетных записей» для обработки. Мы будем называть ее просто «очередью добавления». Сценарии обработки читают эту очередь, создают необходимые учетные записи и сохраняют информацию о созданных учетных записях в отдельной базе данных. Этот процесс отвечает за добавление пользователей в систему.

Процесс удаления пользователя подобен только что описанному. Пользовательский интерфейс создает «очереди удаления». Второй сценарий обработки читает эту очередь, удаляет пользователей из системы и обновляет центральную базу данных.

Данные операции разделены по концептуально различным группам, благодаря чему достигается максимальная гибкость при необходимости что-либо изменить. Например, желание сменить систему баз данных, потребует только других вызовов низкоуровневой библиотеки. Точно так же, для включения дополнительных шагов в процесс добавления пользователей (к примеру, провести сравнение с базой данных отдела кадров) придется изменить только сценарий обработки. Нач-

нем с рассмотрения первого компонента – пользовательского интерфейса, применяемого для создания первоначальной очереди учетных записей. Мы будем использовать простой текстовый интерфейс для запроса параметров учетной записи, поскольку строим только костяк системы:

```
sub CollectInformation{
    # список полей приводится только для наглядности. На самом
    # деле его надо хранить в центральном конфигурационном
    # файле
    my @fields = qw{login fullname id type password};
    my %record;

    foreach my $field (@fields){
        print "Please enter $field: ";
        chomp($record{$field} = <STDIN>);
    }
    $record{status}="to_be_created";
    return \%record;
}
```

В этой подпрограмме создается список, состоящий из различных полей учетной записи пользователя. Как уже говорилось в комментариях, этот список упоминается в коде программы только для краткости. Хорошим стилем проектирования программного обеспечения было бы чтение списка имен полей из дополнительного конфигурационного файла.

После того как список создан, подпрограмма рассматривает его в цикле и запрашивает значение для каждого поля. Каждое значение затем сохраняется в хэше. После получения ответов на все вопросы ссылка на этот хэш возвращается для последующей обработки. Наш следующий шаг – записать информацию в очередь добавления. Перед тем как посмотреть на этот код, мы должны рассказать о хранилище данных и форматах данных, используемых в нашей системе учетных записей.

База данных

Центральная часть любой системы учетных записей – это база данных. Некоторые администраторы используют только файл */etc/ passwd* или базу данных SAM для хранения записей о пользователях системы, но такое решение часто оказывается недальновидным. Помимо информации, о которой мы уже говорили, в отдельной базе данных можно хранить метаданные о каждой учетной записи: например, дату создания учетной записи, срок ее действия, номера телефонов пользователей и прочие сведения. Когда появляется такая база данных, ее можно применять не только для работы с учетными записями. Она годится для создания списков рассылки, служб LDAP и индексации вебстраниц пользователей.

Почему настоящие системные администраторы создают системы учетных записей

Системные администраторы делятся на две категории: ремесленники и архитекторы. Ремесленники большую часть своего времени проводят в непосредственном контакте с подробностями внутреннего устройства ОС. Они знают множество тайн об аппаратном и программном обеспечении, которое они администрируют. Если что-то идет не так, как надо, они знают, какую использовать команду, файл, или какой «гаечный ключ» нужно применить. Талантливые ремесленники могут поразить вас способностью определить и исправить неполадки, находясь даже в соседней комнате от «проблемной» машины.

Архитекторы же тратят время, осматривая компьютерные пространства с высоты. Они мыслят более абстрактно, решая, как сформировать более сложные системы из отдельных частей. Архитекторы озабочены вопросами масштабируемости, расширяемости и повторного использования.

Администраторы обоих типов вносят важный вклад в системное администрирование. Я больше всего уважаю системных администраторов, которые могут быть ремесленниками, но при этом предпочитают действовать как архитекторы. Они решают проблему, а потом определяют, какие изменения в системе можно сделать, чтобы избежать повторения ошибки в дальнейшем. Они думают о том, как даже маленькие усилия с их стороны могут послужить для дальнейшего выигрыша.

Отлично действующее компьютерное окружение требует, чтобы архитекторы работали с ремесленниками в тесном взаимодействии. Ремесленники больше всего полезны при работе в рамках, созданных архитекторами. В автомобильном мире ремесленники нужны для сборки и ремонта машин. Но ремесленники расчитывают на то, что проектировщики машин разрабатывают трудно ломаемые и быстро ремонтируемые автомобили. Чтобы хорошо выполнять свою работу, им нужна инфраструктура, напоминающая сборочный цех, инструкция по эксплуатации и канал поставок запасных частей. Если архитектор хорошо выполняет свою работу, работа ремесленника становится проще.

Какое отношение это имеет к предмету нашего обсуждения? Что ж, вероятно, ремесленники будут применять имеющиеся в операционной системе инструменты для работы с пользователями. Они даже могут пойти дальше и написать небольшие сценарии, упрощающие такие задачи, как добавление пользователей. Архитектор, посмотрев на эту проблему, тут же начнет создавать

систему ведения учетных записей. Архитектор задумается над такими вопросами:

- Природа повторяющихся действий при работе с пользователями и способы, позволяющие максимально автоматизировать данный процесс.
- Тип информации, собираемой системой ведения учетных записей, и условия, при которых правильно созданная система может послужить основой для других действий. Например, как службу каталогов LDAP (Lightweight Directory Access Protocol) и инструменты для автоматического создания вебстраниц можно добавить к такой системе.
- Защита данных в системе учетных записей (т. е. безопасность).
- Создание системы, которая масштабируется при увеличении числа пользователей.
- Создание системы, которую можно использовать и на других машинах.
- Как другие системные администраторы решают такие проблемы.

Упоминание о создании отдельной базы данных заставляет некоторых нервничать. Они думают так: «Теперь мне нужно покупать действительно дорогую коммерческую базу данных, отдельный компьютер, на котором она будет работать, и нанимать администратора баз данных». Если у вас в системе тысячи или десятки тысяч учетных записей, с которыми необходимо работать, – да, вам понадобится все это (хотя можно обойтись и некоммерческими базами данных, такими как Postgres и MySQL). В этом случае переходите к главе 7 «Администрирование баз данных SQL», чтобы подробно узнать о работе с подобными базами данных в Perl.

Но когда в этой главе я говорю *база данных*, то употребляю этот термин в самом широком смысле слова. Плоские файлы вполне подойдут в нашем случае. Пользователи Windows даже могут работать с файлами баз данных Access (например *database.mdb*). В целях переносимости в этом разделе для различных создаваемых компонентов мы будем использовать простые текстовые базы данных. Но чтобы это было более интересным, базы данных будут в формате XML. Если вы никогда раньше не имели дела с XML, пожалуйста, потратьте немного времени и ознакомьтесь с приложением C «Восьмиминутное руководство по XML».

Почему XML? У XML есть несколько свойств, которые делают его хорошим выбором для подобных файлов и других конфигурационных файлов системного администрирования:

- XML – это текстовый формат, следовательно, для работы с ним мы можем использовать наши обычные Perl-приемы, чтобы легко с ним работать.
- XML очень понятен и практически самодокументирован. Разбирая файл, разделенный опередеенными символами, такой как */etc/passwd*, не всегда просто определить, какому полю соответствует какая часть строки. В XML этой проблемы нет, поскольку каждое поле можно окружить очевидным тегом.
- Располагая правильным анализатором, XML может являться также и самопроверяющим. Если применять анализатор, проверяющий синтаксис, то будет очень просто найти ошибки в формате, т. к. этот файл не будет верно разобран в соответствии с определением типа документа (DTD). Модули, которые мы будем применять в этой главе, основаны на анализаторе, не проверяющем синтаксис, но сейчас проводится важная работа по добавлению проверки синтаксиса. Один из шагов в этом направлении – модуль `XML::Checker`, являющийся частью *libxml-enno* Энно Дерксена (Enno Derksen). Анализатор, даже не проверяющий синтаксис, все-таки способен найти много ошибок, если он проверяет формат документа.
- XML достаточно гибок для описания практически любой текстовой информации. Эта гибкость означает, что вы можете применять одну библиотеку анализатора для всех данных, а не писать новый анализатор для каждого нового формата.

Мы будем использовать текстовые файлы в XML-формате для основного файла, в котором хранятся учетные записи, и для очереди добавления/удаления.

И в этом случае вы увидите, что правило TMTOWTDI по-прежнему действует. Для каждой операции с XML, которая нам понадобится, мы рассмотрим или, по крайней мере, упомянем несколько способов ее выполнения. Обычно, собирая подобную систему, лучше ограничить число реализованных опций, а действуя таким образом, вы сможете понять, какие возможности программирования существуют при работе с XML из Perl.

Создание XML-файла из Perl

Давайте вернемся к событиям, о которых мы говорили в разделе «Права пользователей в NT/2000». Тогда речь шла о том, что необходимо записать информацию об учетной записи, получаемую посредством функции `CollectInformation()`, в файл очереди. Но мы так и не видели примеров программы, выполняющей эту задачу. Давайте посмотрим, как записывается этот файл в формате XML.

Проще всего создать XML-файл при помощи простых операторов `print`, но мы поступим лучше. Модули `XML::Generator` Бенджамина Холзмана (Benjamin Holzman) и `XML::Writer` Дэвида Меггинсона (David

Megginson) могут упростить этот процесс и сделать его менее подверженным ошибкам. Они могут обработать такие детали, как соответствие открывающих/закрывающих тегов, а также позаботятся об экранировании специальных символов (<, >, & и т. д.). Вот пример программы, применяемой в нашей системе учетных записей для создания кода XML при помощи модуля `XML::Writer`:

```
sub AppendAccountXML {
    # получаем полный путь к файлу
    my $filename = shift;
    # получаем ссылку на анонимный хэш записи
    my $record = shift;

    # XML::Writer использует объекты IO::File для управления
    # выводом
    use IO::File;

    # дописываем в этот файл
    $fh = new IO::File(">>$filename") or
       die "Unable to append to file:$!\n";

    # инициализируем модуль XML::Writer и говорим ему
    # записывать данные в файловый дескриптор $fh
    use XML::Writer;
    my $w = new XML::Writer(OUTPUT => $fh);

    # записываем открывающий тег для каждой записи <account>
    $w->startTag("account");

    # записываем открывающие/закрывающие внутренние теги и
    # данные в <account>
    foreach my $field (keys %{$record}){
       print $fh "\n\t";
       $w->startTag($field);
       $w->characters($$record{$field});
       $w->endTag;
    }
    print $fh "\n";

    # записываем закрывающий тег для каждой записи <account>
    $w->endTag;
    $w->end;
    $fh->close();
}
```

Теперь можно использовать всего лишь одну строчку, чтобы получить данные и записать их в файл очереди:

```
&AppendAccountXML($addqueue,&CollectInformation);
```

Вот что получается в результате работы этой подпрограммы:[1]

```
<account>
    <login>bobf</login>
    <fullname>Bob Fate</fullname>
    <id>24-9057</id>
    <type>staff</type>
    <password>password</password>
    <status>to_be_created</status>
</account>
```

Да, мы храним пароли открытым текстом. Это очень плохая идея, и даже в случае с нашей демонстрационной системой стоит дважды подумать, прежде чем ее использовать. В настоящей системе учетных записей надо либо шифровать пароль перед тем, как помещать его в очередь, либо вообще не хранить его там.

Функция `AppendAccountXML()` будет применяться еще раз, когда мы захотим записать данные в очередь удаления и в нашу базу данных учетных записей.

Использование модуля `XML::Writer` в подпрограмме `AppendAccountXML()` имеет несколько преимуществ:

- Код получается достаточно читаемым, и тот, кто хоть немного разбирается в языках разметки, сразу же сориентируется в именах `startTag()`, `characters()` и `endTag()`.
- И хотя в нашем случае этого не понадобится, функция `characters()` обеспечивает некоторую защиту – она экранирует зарезервированные символы, например символ (`>`).
- Мы не должны запоминать последний открывающий тег, чтобы потом добавить соответствующий закрывающий. `XML::Writer` заботится об этом за нас и позволяет вызвать функцию `endTag()`, не указывая, *какой* закрывающий тег нам нужен. В нашем случае отслеживание парных тегов не так существенно, поскольку отсутствует их глубокая вложенность, но такая возможность становится очень важной в других ситуациях, где используются более сложные элементы.

Чтение кода XML при помощи XML::Parser

Скоро мы рассмотрим еще один способ построения кода XML в Perl, но сначала вернемся к чтению того кода, который только что научились создавать. Нам необходима программа, которая будет анализировать

[1] В качестве небольшого замечания: в спецификации XML рекомендуется начинать каждый файл с объявления XML (т. е., со строки вида `<?xml version="1.0"?>`). Это не обязательно, но если мы хотим подчиняться правилам, то в модуле `XML::Writer` есть метод `xmlDecl ()`, выводящий такую строку.

очереди добавления и удаления учетных записей, а также основную базу данных.

Можно было бы добавить анализатор XML. Но если с нашим ограниченным набором данных без использования регулярных выражений все бы и прошло, то в случае более сложных XML-данных это вряд ли получилось бы просто.[1] Для обычного анализа проще применить модуль `XML::Parser`, первоначально написанный Ларри Уоллом (Larry Wall) (он был значительно расширен и поддерживается Кларком Купером (Clark Cooper)).

`XML::Parser` – это модуль, основанный на событиях. Такие модули работают, как брокеры на бирже. Перед началом торгов вы оставляете им ряд инструкций о том, какие действия необходимо предпринять, если произойдут конкретные события (например, продать тысячу акций, если цена упадет до $3^1/_4$, купить другие акции в начале торгового дня и т. д.). В случае с программами, основанными на событиях, возникающие ситуации называются *событиями* (*events*), а список инструкций о том, что делать в случае конкретного события, – *обработчиками событий* (*event handlers*). Обработчики – это обычно специальные подпрограммы, созданные для работы с конкретным событием. Некоторые называют их *функциями обратного вызова* (*callback routines*), т. к. они выполняются тогда, когда основная программа «вызывает нас обратно» после того, как наступят определенные условия. В случае с модулем `XML::Parser`, события – это явления, такие как «начало обработки потока данных», «найден открывающий тег» и «найден комментарий». А обработчики будут выполнять что-то подобное: «вывести содержимое только что найденного элемента».[2]

Приступая к анализу данных, необходимо сначала создать объект `XML::Parser`. При создании этого объекта следует указать, какой режим анализа или *стиль* (*style*) нужно применить. `XML::Parser` поддерживает несколько стилей, поведение каждого из которых при анализе данных несколько отличается. Стиль анализа определяет, какие обработчики событий вызываются по умолчанию и каким образом структурированы возвращаемые анализатором данные (если они есть).

Некоторые стили требуют, чтобы мы указывали связь между каждым событием, которое хотим обрабатывать вручную, и его обработчиком. Для событий, не подлежащих обработке, никаких особых действий применяться не будет. Эти связи хранятся в простой хэш-таблице, ключи в ней являются именами событий, которые мы хотим обрабаты-

---

[1] Но сделать это можно. Посмотрите, например, на модуль Эрика Прудоме (Eric Prud'hommeaux) на *http://www.w3.org/1999/02/26-modules/W3C-SAX-XmlParser-**

[2] И хотя здесь это не используется, модуль `XML::Node` Чанга Лью позволяет легко запросить функции обратного вызова только для конкретных элементов, тем самым упрощая рассматриваемый процесс.

вать, а значения – ссылками на подпрограммы-обработчики. В стилях, требующих наличия таких связей, мы передаем хэш посредством именованного параметра Handlers (например, Handlers => {Start => \&start_handler}) при создании объекта анализатора.

Мы будем применять стиль stream, который не требует этого шага инициализации. Он просто вызывает группу предопределенных обработчиков событий, если указанные подпрограммы были найдены в пространстве имен программы. Обработчики событий stream, которые мы будем использовать, очень просты: StartTag, EndTag и Text. Все названия, кроме Text, говорят сами за себя. Text в соответствии с документацией XML::Parser «вызывается прямо перед открывающим или закрывающим тегами с накопленным неразмеченным текстом из переменной $_». Мы будем применять его, когда нам понадобится узнать содержимое конкретного элемента.

Вот какой код будет использоваться в нашем приложении для инициализации:

```
use XML::Parser;
use Data::Dumper; # используется для оформления отладочного вывода, а не
                  # для анализа XML
$p = new XML::Parser(ErrorContext => 3,
                     Style        => 'Stream',
                     Pkg          => 'Account::Parse');
```

Этот код возвращает объект анализатора после передачи ему трех параметров. Первый, ErrorContext, передает анализатору требование вернуть три строки контекста из анализируемых данных в случае возникновения ошибки анализа. Второй устанавливает требуемый стиль анализа. Последний параметр, Pkg, сообщает анализатору, что подпрограммы обработчика событий необходимо искать в ином пространстве имен. Устанавливая этот параметр, мы распоряжаемся, чтобы анализатор искал функции &Account::Parse::StartTag(), &Account::Parse::EndTag() и т. д., а не просто &StartTag(), &EndTag() и т. п. В данном случае это не имеет особого значения, но позволяет избежать ситуации, когда анализатор может случайно вызвать другую функцию с тем же именем StartTag(). Вместо того чтобы использовать параметр Pkg, можно было добавить в самое начало приведенной выше программы строку package Account::Parse;.

Теперь посмотрим на подпрограммы, выполняющие функции обработчика событий. Рассмотрим их по очереди:

```
package Account::Parse;

sub StartTag {
    undef %record if ($_[1] eq "account");
}
```

&StartTag() вызывается каждый раз, когда встречается открывающий тег. Эта функция вызывается с двумя параметрами: ссылкой на объект и именем встреченного тега. Поскольку для каждой учетной записи будет создаваться новая запись в хэше, можно использовать StartTag(), чтобы обозначить начало новой записи (например, открывающий тег <account>). В этом случае удаляются значения из существующего хэша. Во всех остальных случаях мы возвращаемся, ничего не выполняя:

```
sub Text {
   my $ce = $_[0]->current_element();
   $record{$ce}=$_ unless ($ce eq "account");
}
```

На этот раз мы используем &Text() для заполнения хэша %record. Как и предыдущая функция, она тоже получает два параметра при вызове: ссылку на объект и «накопленный неразмеченный текст», который анализатор нашел между последним открывающим и закрывающим тегом. Для определения элемента, в котором мы находимся, используется метод current_element(). В соответствии с документацией по XML::Parser::Expat этот метод «возвращает имя внутреннего элемента, открытого в данный момент». То обстоятельство, что имя текущего элемента не «account», гарантирует нам, что мы находимся внутри одного из подэлементов в <account>, поэтому можно записать имя элемента и его содержимое:

```
sub EndTag {
    print Data::Dumper->Dump([\%record],["account"])
        if ($_[1] eq "account");
    # именно сейчас мы должны сделать что-то конкретное вместо
    # того, чтобы просто печатать запись
}
```

Наш последний обработчик, &EndTag(), очень похож на первый &StartTag() с тем лишь исключением, что он вызывается тогда, когда мы находим закрывающий тег. Если мы дойдем до конца соответствующей учетной записи, то сделаем банальную вещь и напечатаем эту запись. Вот как может выглядеть такой вывод:

```
$account = {
             'login' => 'bobf',
             'type' => 'staff',
             'password' => 'password',
             'fullname' => 'Bob Fate',
             'id' => '24-9057'
           };
$account = {
             'login' => 'wendyf',
             'type' => 'faculty',
             'password' => 'password',
```

```
    'fullname' => 'Wendy Fate',
    'id' => '50-9057'
};
```

Если мы захотим использовать это в нашей системе учетных записей, нам, вероятно, понадобится вызвать некую функцию, например `CreateAccount(\%record)`, а не выводить запись при помощи `Data::Dumper`.

Теперь, когда мы познакомились с процедурами инициализации и обработчиками из `XML::Parser`, нам нужно добавить код, чтобы действительно начать анализ:

```
# обрабатывает записи для нескольких учетных записей из
# одного XML-файла очереди
open(FILE,$addqueue) or die "Unable to open $addqueue:$!\n";
# спасибо Джеффу Пиньяну за это мудрое сокращение
read(FILE, $queuecontents, -s FILE);
$p->parse("<queue>".$queuecontents."</queue>");
```

Этот фрагмент кода, вероятно, заставил вас приподнять бровь, а то и обе. В первых двух строчках мы открываем файл очереди и считываем его содержимое в скалярную переменную `$queuecontents`. Третья строка могла бы показаться понятной, если бы не забавный аргумент, переданный функции `parse()`. Почему мы считываем содержимое файла очереди и заключаем его в теги XML вместо того, чтобы перейти к его анализу?

А потому, что это хак. И надо сказать – не плохой. И вот почему эти «фокусы» необходимы для анализа нескольких элементов `<account>` в одном файле очереди.

Каждый XML-документ по определению должен иметь *корневой элемент* (*root element*). Этот элемент служит контейнером для всего документа; все остальные элементы являются его подэлементами. XML-анализатор ожидает, что первый встреченный им тег будет открывающим тегом корневого элемента документа, а последний тег – закрывающим тегом этого элемента. XML-документы, не соответствующие этой структуре, не считаются корректными (well-formed).

Попытка смоделировать очередь в XML заставит нас призадуматься. Если ничего не сделать, то первым тегом, найденным в файле, будет `<account>`. Все будет работать нормально до тех пор, пока анализатор не встретит закрывающий тег `</account>` для этой записи. В этот момент анализатор завершит свою работу, даже если в очереди есть другие записи, потому что он посчитает, что дошел до конца документа.

Мы без труда могли бы добавить открывающий тег (`<queue>`) в начало очереди, но что делать с закрывающим тегом (`</queue>`)? Закрывающий тег корневого элемента всегда должен быть расположен в самом конце документа (и не иначе), а сделать это не просто, учитывая, что мы собираемся постоянно добавлять записи в этот файл.

Можно было бы (но это довольно неприятно) достигать конца файла при помощи функции seek(), а затем двигаться назад (опять же при помощи seek()) и остановиться прямо перед последним закрывающим тегом. Затем мы могли бы записать нашу новую запись перед этим тегом, оставляя закрывающий тег в самом конце данных. Риск повредить данные (что, если мы перейдем не в ту позицию?) должен предостеречь вас от использования этого метода. Кроме того, этот метод сложно применять, если вы не можете точно определить конец файла, например, при чтении данных XML по сетевому соединению. В подобных случаях, вероятно, стоит буферизовать поток данных, чтобы можно было вернуться к концу данных после завершения соединения.

Метод, показанный в предыдущем примере и предлагающий добавлять пару корневых тегов к существующим данным, может показаться хаком, но выглядит он гораздо элегантнее, чем другие решения. Впрочем, вернемся к более приятной теме.

Чтение XML при помощи XML::Simple

Мы уже видели один метод, позволяющий анализировать XML-данные при помощи модуля XML::Parser. Чтобы соответствовать правилу TMTOWTDI, давайте снова обратимся к этой проблеме, немного упростив задачу. Многие писали собственные модули, построенные на XML::Parser, для анализа XML-документов и возврата данных в удобной для работы форме объектов/структур данных, к их числу относятся и XML::DOM Энно Дэрксена (Enno Derksen), XML::Grove, и ToObjects (часть *libxml-perl*) Кена Маклеода (Ken MacLeod), XML::DT Хосе Хоа Диаса де Альмейды (Jose Joao Dias de Almeida), и XML::Simple Гранта Маклина (Grant McLean). Из всех этих модулей, вероятно, проще всего использовать модуль XML::Simple. Он был создан для обработки небольших конфигурационных файлов на XML, что отлично подходит для нашей задачи.

XML::Simple предоставляет две функции. Вот первая (в данном контексте):

```
use XML::Simple;
use Data::Dumper;  # нужен для вывода содержимого структур
                   # данных

$queuefile = "addqueue.xml";
open(FILE,$queuefile) or die "Unable to open $queuefile:$!\n";
read(FILE, $queuecontents, -s FILE);
$queue = XMLin("<queue>".$queuecontents."</queue>");
```

Содержимое $queue мы выводим подобным образом:

```
print Data::Dumper->Dump([$queue],["queue"]);
```

Теперь это ссылка на данные, найденные в файле очереди, сохраненные в виде хэша хэшей, ключами которого являются элементы <id>. Ниже, приведена эта структура данных (рис. 3.3).

```
$queue = {
           'account' => {
                          '24-9057' => {
                                         'login' => 'bobf',
                                         'type' => 'staff',
                                         'password' => 'password',
                                         'fullname' => 'Bob Fate',
                                         'status' =>'to_be_created'
                          }
                          '50-9057' => {
                                         'login' => 'wendyf',
                                         'type' => 'faculty',
                                         'password' => 'password',
                                         'fullname' => 'Wendy Fate',
                                         'status' =>'to_be_created'
                                       }
                          }
        };
```

***Рис 3.3.** Структура данных, созданная функцией XMLin( ) без специальных аргументов*

Мы используем именно такие ключи потому, что XML::Simple позволяет распознавать в данных конкретные теги, выделяя их среди других в процессе преобразования. Если мы отключим эту возможность:

```
$queue = XMLin("<queue>".$queuecontents."</queue>",keyattr=>[]);
```

то получим ссылку на хэш, где единственное значение является ссылкой на анонимный массив. Так хранятся данные в анонимном массиве (рис. 3.4).

Такая структура данных не очень полезна. Этот параметр можно определять по собственному желанию:

```
$queue = XMLin("<queue>".$queuecontents."</queue>",keyattr => ["login"]);
```

Теперь мы получим ссылку на структуру данных (хэш хэшей, ключи которого являются регистрационными именами), как видно из рис. 3.5 – это именно то, что нам нужно.

Замечательно? Теперь мы можем удалить элементы из очереди в памяти, после того как обработаем их всего в одной строке:

```
# например, $login = "bobf";
delete $queue->{account}{$login};
```

Если мы хотим изменить значение, перед тем как записать его на диск (скажем, мы работаем с нашей основной базой данных), то это тоже просто сделать:

```
# например, $login="wendyf"; $field="status";
$queue->{account}{$login}{$field}="created";
```

```
$queue = {
          'account' => [
                         {
                           'login' => 'bobf',
                           'type' => 'staff',
                           'password' => 'password',
                           'status' => 'to_be_created',
                           'fullname' => 'Bob Fate',
                           'id' => '24-9057'
                         },
                         {
                           'login' => 'wendyf',
                           'type' => 'faculty',
                           'password' => 'password',
                           'status' => 'to_be_created',
                           'fullname' => 'Wendy Fate',
                           'id' => '50-9057'
                         }
                       ]
        };
```

Рис. 3.4. *Структура данных, создаваемая функцией XMLin( ) с отключенным параметром keyattr*

```
$queue = {
          'account' => {
                         'bobf' =>    {
                                        'type' => 'staff',
                                        'password' => 'password',
                                        'fullname' => 'Bob Fate',
                                        'status' => 'to_be_created',
                                        'id' => '24-9057'
                                      },
                         'wendyf' =>  {
                                        'type' => 'faculty',
                                        'password' => 'password',
                                        'fullname' => 'Wendy Fate',
                                        'status' => 'to_be_created',
                                        'id' => '50-9057'
                                      }
                       }
        };
```

Рис. 3.5. *Та же структура данных, параметр keyattr определяется пользователем*

Создание XML-данных при помощи XML::Simple

Упоминание «записать его на диск» возвращает нас обратно к методу создания XML-данных, который мы обещали показать. Вторая функция из `XML::Simple` принимает ссылку на структуру данных и генерирует XML-данные:

```
# rootname определяет имя корневого элемента, мы могли бы
# использовать XMLdecl, чтобы добавить объявление XML
print XMLout($queue, rootname =>"queue");
```

В результате получаем (отступы сделаны для удобства чтения):

```
<queue>
  <account name="bobf" type="staff"
           password="password" status="to_be_created"
           fullname="Bob Fate" id="24-9057" />
  <account name="wendyf" type="faculty"
           password="password" status="to_be_created"
           fullname="Wendy Fate" id="50-9057" />
</queue>
```

Мы получили отличный XML-код, но его формат несколько отличается от формата наших файлов с данными. Данные о каждой учетной записи представлены в виде атрибутов одного элемента `<account> </ account>`, а не в виде вложенных элементов. В `XML::Simple` есть несколько правил, руководствуясь которыми, он преобразовывает структуры данных. Два из них можно сформулировать так (а остальные можно найти в документации): «отдельные значения преобразуются в XML-атрибуты», а «ссылки на анонимные массивы преобразуются во вложенные XML-элементы».

Чтобы получить «верный» XML-документ («верный» означает «в том же стиле и того же формата, что и наши файлы данных»), необходимо хранить в памяти подобную структуру данных (рис. 3.6).

```
$queue = {
           'account' => {
                          {
                            'login' => ['bobf'],
                            'type' => ['staff'],
                            'password' => ['password'],
                            'status' => ['to_be_created'],
                            'fullname' => ['Bob Fate'],
                            'id' => ['24-9057']
                          },
                          {
                            'login' => ['wendyf'],
                            'type' => ['faculty'],
                            'password' => ['password'],
                            'status' => ['to_be_created'],
                            'fullname' => ['Wendy Fate'],
                            'id' => ['50-9057']
                          }
                        }
         };
```

***Рис 3.6.** Структура данных, необходимая для создания файла очереди в XML-формате*

Кошмар, не правда ли? Но у нас есть варианты для выбора. Мы можем:

1. Изменить формат наших файлов данных. Это похоже на крайнюю меру.
2. Изменить способ, которым `XML::Simple` анализирует наш файл. Чтобы получить такую структуру данных (рис. 3.6), мы могли бы использовать функцию `XMLin()` несколько иначе:

```
$queue = XMLin("<queue>".$queuecontents."</queue>",
                    forcearray=>1, keyattr => [""]);
```

   Но если мы перекроим способ чтения данных, чтобы упростить запись, то потеряем семантику хэшей, упрощающих поиск и обработку данных.
3. Выполнить некую обработку данных после чтения, но до записи. Мы могли бы прочитать данные в нужную нам структуру (так же, как делали это раньше), применить эти данные в нужном месте, а затем преобразовать структуру данных в один из вариантов, получаемых модулем `XML::Simple`, перед тем как записать ее.

Вариант номер 3 кажется более разумным, так что последуем ему. Вот подпрограмма, которая принимает одну структуру данных (рис. 3.5), и преобразует ее в другую структуру данных (рис. 3.6). Объяснение примера будет приведено позже:

```
sub TransformForWrite{
  my $queueref = shift;
  my $toplevel = scalar each %$queueref;

  foreach my $user (keys %{$queueref->{$toplevel}}){
    my %innerhash =
       map {$_,[$queueref->{$toplevel}{$user}{$_}] }
             keys %{$queueref->{$toplevel}{$user}};
    $innerhash{'login'} = [$user];
    push @outputarray, \%innerhash;
  }

  $outputref = { $toplevel => \@outputarray};
  return $outputref;
}
```

Теперь подробно рассмотрим подпрограмму `TransformForWrite()`.

Если вы сравните две структуры (рис. 3.5, рис. 3.6), то заметите в них кое-что общее: это внешний хэш, ключом которого в обоих случаях является `account`. В следующей строке видно, как получить имя этого ключа, запрашивая первый ключ из хэша, на который указывает `$queueref`:

```
my $toplevel = scalar each %$queueref;
```

Интересно взглянуть на закулисную сторону создания этой структуры данных:

```
my %innerhash =
   map {$_,[$queueref->{$toplevel}{$user}{$_}] }
         keys %{$queueref->{$toplevel}{$user}};
```

В этом отрывке кода мы используем функцию `map()`, чтобы обойти все ключи, найденные во внутреннем хэше для каждой записи (т. е. `login`, `type`, `password` и `status`). Ключи возвращаются в такой строке:

```
keys %{$queueref->{$toplevel}{$user}};
```

Просматривая ключи, можно с помощью `map` вернуть два значения для каждого из них: сам ключ и ссылку на анонимный массив, содержащий его значение:

```
map {$_,[$queueref->{$toplevel}{$user}{$_}] }
```

Список, возвращаемый `map()`, выглядит так:

```
(login,[bobf], type,[staff], password,[password]...)
```

Он имеет формат ключ-значение, где значения хранятся как элементы анонимного массива. Этот список можно присвоить хэшу `%innerhash`, чтобы заполнить внутреннюю хэш-таблицу для получаемой структуры данных (`my %innerhash =`). Кроме того, к хэшу следует добавить ключ `login`, соответствующий рассматриваемому пользователю:

```
$innerhash{'login'} = [$user];
```

Структура данных, которую мы пытаемся создать, – это список подобных хэшей, поэтому после того как будет создан и определен внутренний хэш, необходимо добавить ссылку на него в конец списка, т. к. он и представляет получаемую структуру данных:

```
push @outputarray, \%innerhash;
```

Такую процедуру следует повторить для каждого ключа `login` из первоначальной структуры данных (один на каждую запись об учетной записи). После того как это будет сделано, у нас появится список ссылок на хэши в той форме, которая нам нужна. Мы создаем анонимный хэш с ключом, совпадающим с внешним ключом из первоначальной структуры данных, и значением, равным нашему списку хэшей. Ссылку на этот анонимный хэш можно возвратить обратно вызывающей программе. Вот и все:

```
$outputref = { $toplevel => \@outputarray};
return $outputref;
```

Теперь, располагая `&TransformForWrite()`, мы можем написать программу для чтения, записи наших данных и работы с ними:

```
$queue = XMLin("<queue>".$queuecontents."</queue>",keyattr => ["login"]);
обрабатываем данные…
print OUTPUTFILE XMLout(TransformForWrite($queue),rootname => "queue");
```

Записанные и прочитанные данные будут иметь один и тот же формат.

Перед тем как закрыть тему чтения и записи данных, следует избавиться от несоответствий:

1. Внимательные читатели, наверное, заметили, что одновременное использование `XML::Writer` и `XML::Simple` в одной и той же программе для записи данных в очередь может оказаться непростым делом. Если записывать данные при помощи `XML::Simple`, то они будут вложены в корневой элемент по умолчанию. Если же применять `XML::Writer` (или просто операторы `print`) для записи данных, вложения не произойдет, т. е. нам придется опять прибегнуть к хаку `"<queue>".$queuecontents."</queue>"`. Возникает неудачный уровень синхронизации чтения-записи между программами, анализирующими и записывающими данные в XML-формате.

   Чтобы избежать этой проблемы, надо будет использовать продвинутую возможность модуля `XML::Simple`: если `XMLout()` передать параметр `rootname` с пустым значением или значением `undef`, то возвращаются данные в XML-формате без корневого элемента. В большинстве случаев так поступать не следует, потому что в результате образуется неправильный (синтаксически) документ, который невозможно проанализировать. Наша программа позволяет этим методом воспользоваться, но такую возможность не стоит применять необдуманно.

2. И хотя в примере этого нет, мы должны быть готовы к обработке ошибок анализа. Если файл содержит синтаксически неверные данные, то анализатор не справится и прекратит работу (согласно спецификации XML), остановив при этом и всю программу в случае, если вы не примете мер предосторожности. Самый распространенный способ справиться с этим из Perl – заключить оператор анализа в `eval()` и затем проверить содержимое переменной `$@` после завершения работы анализатора.[1] Например:

   ```
   eval {$p->parse("<queue>".$queuecontents."</queue>")};
   if ($@) { сделать что-то для обработки ошибки перед выходом… };
   ```

   Другим решением было бы применение известного модуля из разряда `XML::Checker`, т. к. он обрабатывает ошибки разбора аккуратнее.

[1] Дэниел Буркхардт (Daniel Burckhardt) сообщил в списке рассылки Perl-XML, что у этого метода есть свои недостатки. В многопоточной программе на Perl проверка значения глобальной переменной `$@` может оказаться небезопасной, если не предпринять никаких мер предосторожности. Подобные вопросы, связанные с многопоточностью, все еще обсуждаются разработчиками.

Низкоуровневая библиотека компонентов

Теперь, когда мы умеем отследить данные на всех этапах, включая то, как они получаются, записываются, читаются и хранятся, можно перейти к рассмотрению их использования глубоко в недрах нашей системы учетных записей. Мы собираемся исследовать код, который действительно создает и удаляет пользователей. Ключевой момент этого раздела заключается в создании библиотеки повторно используемых компонентов. Чем лучше вам удастся разбить систему учетных записей на подпрограммы, тем проще будет внести лишь небольшие изменения, когда придет время переходить на другую операционную систему или что-либо менять. Это предупреждение может показаться ненужным, но единственное, что остается постоянным в системном администрировании, – это постоянные изменения.

Подпрограммы для создания и удаления учетных записей в Unix

Начнем с примеров кода для создания учетных записей в Unix. Большая часть этого кода будет элементарной, поскольку мы избрали легкий путь. Наши подпрограммы для создания и удаления учетных записей вызывают команды с необходимыми аргументами, входящие в состав операционной системы, для «добавления пользователей», «удаления пользователей» и «смены пароля».

Зачем нужна эта очевидная попытка отвертеться? Этот метод приемлем, поскольку известно, что программы, входящие в состав операционной системы, хорошо «уживаются» с другими компонентами. В частности, этот метод:

- Не забывает о блокировке (т. е. позволяет избежать проблем с поврежденными данными, которые могут возникнуть, если две программы пытаются одновременно записать данные в файл паролей).
- Справляется с вариациями в файле паролей (включая шифрование пароля), о чем упоминалось раньше.
- Наверняка справится со схемами авторизации и механизмами распространения паролей, существующими в этой операционной системе. Например, в Digital Unix добавляющая пользователей внешняя программа может напрямую работать и с NIS-картами на основном сервере.

Применение внешних программ для создания и удаления учетных записей обладает такими недостатками:

Различия операционных систем

В каждую операционную систему входит свой собственный набор программ, расположенных в разных местах и принимающих несколько различные аргументы. Это редкий пример совместимости, однако практически во всех распространенных вариантах Unix (включая

Linux, но исключая BSD) используются максимально совместимые программы для удаления и создания пользователей: *useradd* и *userdel*. В вариантах BSD применяются *adduser* и *rmuser*, две программы со сходным назначением, но совершенно разными аргументами. Подобные различия могут значительно усложнить наш код.

Соображения безопасности

Вызываемые программы с переданными им аргументами будут видны всем, кто употребляет команду *ps*. Если создавать учетные записи только на защищенной машине (например, на основном сервере), риск утечки данных значительно снизится.

Зависимость от программы

Если внешняя программа почему-либо изменится или будет удалена, то нашей системе учетных записей настанет «полный капут».

Потеря контроля

Нам приходится считать часть процесса создания учетной записи *неделимым*. Другими словами, когда запущена внешняя программа, мы не можем вмешаться в этот процесс и добавить какие-либо свои собственные операции. Выявление ошибок и процесс восстановления становятся более сложными.

Эти программы редко делают все

Вероятнее всего, что данные программы не выполняют все действия, необходимые для формирования учетной записи на вашей машине. Возможно, вам понадобится добавить некоторых пользователей в некоторые вспомогательные группы, включить их в список рассылки на вашей системе или же добавить пользователей к файлу лицензии коммерческого продукта. Для обработки подобных действий вам придется написать дополнительные программы. Это, конечно, не проблема, наверняка любая система учетных записей, которую вы придумаете, потребует от вас большего, чем просто вызвать пару внешних программ. Более того, это не удивит большинство системных администраторов, потому что их работа меньше всего похожа на беззаботную прогулку по парку.

В случае с нашей демонстрационной системой учетных записей преимущества перевешивают недостатки, поэтому посмотрим на примеры кодов, в которых используется вызов внешних программ. Чтобы ничего не усложнять, мы покажем пример программы, работающей только на локальной машине с Linux и Solaris, и проигнорируем все трудности, вроде NIS и вариаций BSD. Если вам хочется посмотреть на более сложный пример этого метода в действии, поищите семейство модулей `CfgTie` Рэнди Мааса (Randy Maas).

Вот основная программа, необходимая для создания учетной записи:

```
# На самом деле эти переменные надо определить в центральном
# конфигурационном файле
```

```
$useraddex    = "/usr/sbin/useradd";  # путь к useradd
$passwdex     = "/bin/passwd";        # путь к passwd
$homeUnixdirs = "/home";              # корневой каталог
                                      # домашних каталогов
$skeldir      = "/home/skel";         # прототип домашнего
                                      # каталога
$defshell     = "/bin/zsh";           # интерпретатор по
                                      # умолчанию

sub CreateUnixAccount{

    my ($account,$record) = @_;

    ### конструируем командную строку, используя:
    # -c = поле комментария
    # -d = домашний каталог
    # -g = группа (считаем равной типу пользователя)
    # -m = создать домашний каталог
    # -k = и скопировать файлы из каталога-прототипа
    # -s = интерпретатор по умолчанию
    # (можно также использовать -G group, group, group для
    # добавления пользователя к нескольким группам)
    my @cmd = ($useraddex,
          "-c", $record->{"fullname"},
           "-d", "$homeUnixdirs/$account",
           "-g", $record->{"type"},
           "-m",
           "-k", $skeldir,
           "-s", $defshell,
           $account);

    print STDERR "Creating account...";
    my $result = 0xff & system @cmd;
    # код возврата 0 в случае успеха и не 0 при неудаче,
    # поэтому необходимо инвертирование
    if (!$result){
        print STDERR "failed.\n";
        return "$useraddex failed";
    }
    else {
        print STDERR "succeeded.\n";
    }

    print STDERR "Changing passwd...";
    unless ($result = &InitUnixPasswd($account,$record->{"password"})){
        print STDERR "succeeded.\n";
        return "";
    }
    else {
        print STDERR "failed.\n";
        return $result;
    }
}
```

В результате необходимая запись будет добавлена в файл паролей, будет создан домашний каталог для учетной записи и скопированы некоторые файлы окружения (*.profile*, *.tcshrc*, *.zshrc*, и т. д.) из каталога-прототипа.

Обратите внимание, что мы используем отдельный вызов для установки пароля. Команда *useradd* на некоторых операционных системах (например, Solaris) оставляет учетную запись заблокированной до тех пор, пока для этой учетной записи не будет вызвана программа *passwd*. Подобный процесс требует известной ловкости рук, поэтому мы оформим данный шаг как отдельную подпрограмму, чтобы оставить в стороне подробности. Об этой подпрограмме мы еще поговорим, а пока рассмотрим «симметричный» код, удаляющий учетные записи:

```
# На самом деле эти переменные надо устанавливать в центральном
# конфигурационном файле
$userdelex = "/usr/sbin/userdel";  # путь к userdel

sub DeleteUnixAccount{

    my ($account,$record) = @_;

    ### конструируем командную строку, используя:
    # -r = удалить домашний каталог
    my @cmd = ($userdelex, "-r", $account);

    print STDERR "Deleting account...";
    my $result = 0xffff & system @cmd;
    # код возврата 0 соответствует успеху, не 0 - неудаче,
    # поэтому необходимо инвертирование
    if (!$result){
        print STDERR "succeeded.\n";
        return "";
    }
    else {
        print STDERR "failed.\n";
        return "$userdelex failed";
    }
}
```

Перед тем как перейти к операциям с учетными записями в NT, разберемся с подпрограммой `InitUnixPasswd()`, о которой упоминалось раньше. Чтобы завершить создание учетной записи (по крайней мере, в Solaris), необходимо изменить ее пароль при помощи стандартной команды *passwd*. Обращение *passwd <accountname>* изменит пароль для этой учетной записи.

Звучит просто, но тут затаилась проблема. Команда *passwd* запрашивает пароль у пользователя. Она принимает меры предосторожности, чтобы убедиться, что общается с настоящим пользователем, взаимо-

действуя напрямую с его терминалом. В результате следующий код работать *не* будет:

```
# такой код РАБОТАТЬ НЕ БУДЕТ
open(PW,"|passwd $account");
print PW $oldpasswd,"\n";
print PW $newpasswd,"\n";
```

На этот раз мы должны быть искуснее, чем обычно; нам нужно как-то заставить команду *passwd* думать, что она имеет дело с человеком, а не программой на Perl. Этого можно достичь, если использовать модуль *Expect.pm*, написанный Остином Шутцом (Austin Schutz), – ведь он устанавливает псевдотерминал (pty), внутри которого выполняется другая программа. *Expect.pm* основан на известной Tcl-программе *Expect* Дона Либеса (Don Libes). Этот модуль входит в семейство модулей, взаимодействующих с программами. В главе 6 мы рассмотрим его близкого «родственника», модуль `Net::Telnet` Джея Роджерса (Jay Rogers).

Эти модули действуют в соответствии со следующей моделью: они ждут вывода программы, посылают ей на ввод данные, ждут ответа, посылают некоторые данные и т. д. Приведенная ниже программа запускает команду *passwd* в псевдотерминале и ждет до тех пор, пока та запросит пароль. Поддержание «разговора» с *passwd* не должно требовать усилий:

```
use Expect;

sub InitUnixPasswd {
    my ($account,$passwd) = @_;

    # вернуть объект
    my $pobj = Expect->spawn($passwdex, $account);
    die "Unable to spawn $passwdex:$!\n" unless (defined $pobj);

    # не выводить данные на стандартный вывод (т. е.
    # работать молча)
    $pobj->log_stdout(0);

    # Подождать запроса на ввод пароля и запроса на повторение
    # пароля, ответить.
    $pobj->expect(10,"New password: ");
    # Linux иногда выводит подсказки раньше, чем он готов к вводу, поэтому
приостанавливаемся
    sleep 1;
    print $pobj "$passwd\r";
    $pobj->expect(10, "Re-enter new password: ");
    print $pobj "$passwd\r";

    # работает?
    $result = (defined ($pobj->expect(10,
              "successfully changed")) ? "" : "password change
              failed");
```

```
    # закрываем объект, ждем 15 секунд, пока процесс завершится
    $pobj->soft_close();

    return $result;
}
```

Модуль *Expect.pm* очень хорошо подходит для этой подпрограммы, но стоит отметить, что он годится для куда более сложных операций. Подробную информацию можно найти в документации и руководстве по модулю *Expect.pm*.

Подпрограммы для создания и удаления учетных записей в Windows NT/2000

Процесс создания и удаления учетных записей в Windows NT/2000 несколько проще, чем в Unix, поскольку стандартные вызовы API для этой операции существуют в NT. Как и в Unix, мы могли бы вызвать внешнюю программу, чтобы выполнить подобную работу (например, вездесущую команду *net* с ключом USERS/ADD), но проще использовать API-вызовы из многочисленных модулей, о некоторых из которых мы уже говорили. Функции для создания учетных записей есть, например, в `Win32::NetAdmin`, `Win32::UserAdmin`, `Win32API::Net` и `Win32::Lanman`. Пользователям Windows 2000 лучше ознакомиться с материалом по ADSI в главе 6.

Выбор одного из этих модулей, в основном, дело вкуса. Чтобы разобраться в отличиях между ними, рассмотрим существующие вызовы для создания пользователей. Эти вызовы описаны в документации Network Management SDK на *http://msdn.microsoft.com* (если вы ничего не можете найти, поищите «NetUserAdd»). `NetUserAdd()` и другие вызовы принимают в качестве параметра информационный уровень данных. Например, если информационный уровень равен 1, структура данных на C, передаваемая вызову для создания пользователя, выглядит так:

```
typedef struct _USER_INFO_1 {
  LPWSTR    usri1_name;
  LPWSTR    usri1_password;
  DWORD     usri1_password_age;
  DWORD     usri1_priv;
  LPWSTR    usri1_home_dir;
  LPWSTR    usri1_comment;
  DWORD     usri1_flags;
  LPWSTR    usri1_script_path;
}
```

Если используется информационный уровень, равный 2, структура значительно расширится:

```
typedef struct _USER_INFO_2 {
```

```
  LPWSTR    usri2_name;
  LPWSTR    usri2_password;
  DWORD     usri2_password_age;
  DWORD     usri2_priv;
  LPWSTR    usri2_home_dir;
  LPWSTR    usri2_comment;
  DWORD     usri2_flags;
  LPWSTR    usri2_script_path;
  DWORD     usri2_auth_flags;
  LPWSTR    usri2_full_name;
  LPWSTR    usri2_usr_comment;
  LPWSTR    usri2_parms;
  LPWSTR    usri2_workstations;
  DWORD     usri2_last_logon;
  DWORD     usri2_last_logoff;
  DWORD     usri2_acct_expires;
  DWORD     usri2_max_storage;
  DWORD     usri2_units_per_week;
  PBYTE     usri2_logon_hours;
  DWORD     usri2_bad_pw_count;
  DWORD     usri2_num_logons;
  LPWSTR    usri2_logon_server;
  DWORD     usri2_country_code;
  DWORD     usri2_code_page;
}
```

Не обязательно много знать об этих параметрах или даже вообще о C, чтобы понять, что при изменении уровня увеличивается количество информации, которое можно передать при создании пользователя. Кроме того, каждый последующий уровень является надмножеством предыдущего.

Какое это имеет отношение к Perl? Каждый упомянутый модуль требует принять два решения:

1. Нужно ли объяснять программистам на Perl, что такое «информационный уровень»?
2. Какой информационный уровень (т. е. сколько параметров) может использовать программист?

Модули `Win32API::Net` и `Win32::UserAdmin` позволяют программисту выбрать информационный уровень. `Win32::NetAdmin` и `Win32::Lanman` этого не делают. Из всех этих модулей `Win32::NetAdmin` применяет наименьшее число параметров; в частности, вы не можете определить поле `full_name` на этапе создания пользователя. Если вы решите применять модуль `Win32::NetAdmin`, вам, скорее всего, придется дополнить его вызовами из другого модуля, чтобы установить те параметры, которые он устанавливать не позволяет. Если вы остановитесь на комбинации `Win32::NetAdmin` и `Win32::AdminMisc`, вам стоит обратиться к многократно упомянутой книге Рота, поскольку это отличный справочник по моду-

лю Win32::NetAdmin, по которому нет достаточного количества документации.

Теперь читателю должно быть понятно, почему выбор модуля – это дело личных предпочтений. Хорошей стратегией было бы сначала решить, какие параметры важны для вас, а затем найти модуль, который их поддерживает. Для наших демонстрационных подпрограмм мы выбираем модуль Win32::Lanman. Вот какой код можно применить для создания и удаления пользователей в нашей системе учетных записей:

```
use Win32::Lanman;    # для создания учетной записи
use Win32::Perms;     # для установки прав на домашний каталог

$homeNTdirs = "\\\\homeserver\\home";      # корневой каталог
                                           # домашних каталогов

sub CreateNTAccount{

    my ($account,$record) = @_;

    # создаем учетную запись на локальной машине
    # (т. е., первый параметр пустой)
    $result = Win32::Lanman::NetUserAdd("",
                    {'name' => $account,
                     'password'  => $record->{password},
                     'home_dir'  => "$homeNTdirs\\$account",
                     'full_name' => $record->{fullname}});
    return Win32::Lanman::GetLastError() unless ($result);

    # добавляем в нужную ЛОКАЛЬНУЮ группу (предварительно мы
    # получаем SID учетной записи)
    # Мы считаем, что имя группы совпадает с типом учетной
    # записи
    die "SID lookup error: ".Win32::Lanman::GetLastError()."\n"
      unless (Win32::Lanman::LsaLookupNames("", [$account],
               \@info));
    $result = Win32::Lanman::NetLocalGroupAddMember("",
                $record->{type}, ${$info[0]}{sid});
    return Win32::Lanman::GetLastError() unless ($result);

    # создаем домашний каталог
    mkdir "$homeNTdirs\\$account",0777 or
       return "Unable to make homedir:$!";

    # устанавливаем ACL и владельца каталога
    $acl = new Win32::Perms("$homeNTdirs\\$account");
    $acl->Owner($account);

    # мы предоставляем пользователю полный контроль за
    # каталогом и всеми файлами, которые будут в нем созданы
    # (потому и два различных вызова)
```

```
    $acl->Allow($account, FULL,
                    DIRECTORY|CONTAINER_INHERIT_ACE);
    $acl->Allow($account, FULL,
                FILE|OBJECT_INHERIT_ACE|INHERIT_ONLY_ACE);
    $result = $acl->Set();
    $acl->Close();

    return($result ? "" : $result);
}
```

Программа для удаления пользователей выглядит так:

```
use Win32::Lanman;   # для удаления учетной записи
use File::Path;      # для рекурсивного удаления каталогов

sub DeleteNTAccount{

    my($account,$record) = @_;

    # удаляем пользователя только из ЛОКАЛЬНЫХ групп. Если мы
    # хотим удалить их и из глобальных групп, мы можем убрать
    # слово "Local" из двух вызовов Win32::Lanman::NetUser*
    # (например, NetUserGetGroups)
    die "SID lookup error: ".Win32::Lanman::GetLastError()."\n"
          unless (Win32::Lanman::LsaLookupNames("",
                  [$account], \@info));
    Win32::Lanman::NetUserGetLocalGroups($server, $account,'',
                                         \@groups);
    foreach $group (@groups){
        print "Removing user from local group ".
                              $group->{name}."...";
        print(Win32::Lanman::NetLocalGroupDelMember("",
                              $group->{name},
                    ${$info[0]}{sid}) ?
                              "succeeded\n" : "FAILED\n");
    }

   # удалить эту учетную запись с локальной машины
    # (т. е., перый параметр пустой)
    $result = Win32::Lanman::NetUserDel("", $account);

    return Win32::Lanman::GetLastError() if ($result);

    # удалить домашний каталог и его содержимое
    $result = rmtree("$homeNTdirs\\$account",0,1);

    # rmtree возвращает число удаленных файлов, так что если мы
    # удалили более нуля элементов, то скорее всего все прошло
    # успешно
    return $result;
}
```

Заметьте, для удаления домашнего каталога здесь используется переносимый модуль File::Path. Если бы мы хотели сделать что-то специфичное для Win32, например, переместить домашний каталог в корзину, то могли бы сделать это при помощи модуля Win32::FileOp Йенды Крыники (Jenda Krynicky), который можно найти на *http://jenda.krynicky.cz/*. В таком случае мы применили бы Win32::FileOp и изменили бы строку, включающую rmtree(), на:

```
# удалим каталог в корзину, потенциально подтверждая это
# действие пользователем, если для этой учетной записи
# необходимо подтверждать такие операции
$result = Recycle("$homeNTdirs\\$account");
```

В данном модуле есть функция Delete(), которая выполняет то же, что и rmtree() менее переносимым (правда, более быстрым) способом.

Сценарии

Теперь, когда мы разобрались с базой данных, самое время написать сценарии для выполнения периодических или каждодневных действий, необходимых при системном администрировании. Эти сценарии построены на низкоуровневой библиотеке компонентов (*Account.pm*), которую мы создали, объединив в один файл все только что написанные подпрограммы. Такая подпрограмма позволяет убедиться, что все необходимые модули загружены:

```
sub InitAccount{

    use XML::Writer;

    $record   = { fields => [login,fullname,id,type,password]};
    $addqueue   = "addqueue";  # имя файла очереди добавления
    $delqueue   = "delqueue";  # имя файла очереди удаления
    $maindata   = "accountdb"; # имя основной базы данных
                               # учетных записей

    if ($^O eq "MSWin32"){
        require Win32::Lanman;
        require Win32::Perms;
        require File::Path;

        # местоположение файлов учетных записей
        $accountdir = "\\\\server\\accountsystem\\";
        # списки рассылки
        $maillists  = "$accountdir\\maillists\\";
        # корневой каталог домашних каталогов
        $homeNTdirs = "\\\\homeserver\\home";
        # имя подпрограммы, добавляющей учетные записи
        $accountadd = "CreateNTAccount";
        # имя подпрограммы, удаляющей учетные записи
        $accountdel = "DeleteNTAccount";
```

```
    }
    else {
        require Expect;
        # местоположение файлов учетных записей
        $accountdir   = "/usr/accountsystem/";
        # списки рассылки
        $maillists    = "$accountdir/maillists/";
        # местоположение команды useradd
        $useraddex    = "/usr/sbin/useradd";
        # местоположение команды userdel
        $userdelex    = "/usr/sbin/userdel";
        # местоположение команды passwd
        $passwdex     = "/bin/passwd";
        # корневой каталог домашних каталогов
        $homeUnixdirs = "/home";
        # прототип домашнего каталога
        $skeldir      = "/home/skel";
        # командный интерпретатор по умолчанию
        $defshell     = "/bin/zsh";
        # имя подпрограммы, добавляющей учетные записи
        $accountadd   = "CreateUnixAccount";
        # имя подпрограммы, удаляющей учетные записи
        $accountdel   = "DeleteUnixAccount";
    }
}
```

Рассмотрим сценарий, обрабатывающий очередь добавления:

```
use Account;
use XML::Simple;

&InitAccount;     # считываем низкоуровневые подпрограммы
&ReadAddQueue;    # считываем и анализируем очередь добавления
&ProcessAddQueue; # пытаемся создать все учетные записи
&DisposeAddQueue; # записываем учетную запись либо в основную
                  # базу данных, либо обратно в очередь, если
                  # возникли какие-то проблемы

# считываем очередь добавления в структуру данных $queue
sub ReadAddQueue{
    open(ADD,$accountdir.$addqueue) or
      die "Unable to open ".$accountdir.$addqueue.":$!\n";
    read(ADD, $queuecontents, -s ADD);
    close(ADD);
    $queue = XMLin("<queue>".$queuecontents."</queue>",
                   keyattr => ["login"]);
}

# обходим в цикле структуру данных, пытаясь создать учетную
# запись для каждого запроса (т. е. для каждого ключа)
sub ProcessAddQueue{
    foreach my $login (keys %{$queue->{account}}){
```

```
        $result = &$accountadd($login,
                        $queue->{account}->{$login});
        if (!$result){
            $queue->{account}->{$login}{status} = "created";
        }
        else {
            $queue->{account}->{$login}{status} =
                        "error:$result";
        }
    }
}

# теперь снова обходим структуру данных. Каждую учетную запись
# со статусом "created," добавляем в основную базу данных. Все
# остальные записываем обратно в файл очереди, перезаписывая
# все его содержимое.
sub DisposeAddQueue{
    foreach my $login (keys %{$queue->{account}}){
        if ($queue->{account}->{$login}{status} eq "created"){
            $queue->{account}->{$login}{login} = $login;
            $queue->{account}->{$login}{creation_date} = time;
            &AppendAccountXML($accountdir.$maindata,
                              $queue->{account}->{$login});
            delete $queue->{account}->{$login};
            next;
        }
    }

    # То, что осталось сейчас в $queue, - это учетные записи,
    # которые невозможно создать

    # перезаписываем файл очереди
    open(ADD,">".$accountdir.$addqueue) or
      die "Unable to open ".$accountdir.$addqueue.":$!\n";
    # если есть учетные записи, которые не были созданы,
    # записываем их
    if (scalar keys %{$queue->{account}}){
        print ADD XMLout(&TransformForWrite($queue),
                         rootname => undef);
    }
    close(ADD);
}
```

Сценарий, обрабатывающий очередь удаления, очень похож:

```
use Account;
use XML::Simple;

&InitAccount;     # считываем низкоуровневые подпрограммы
&ReadDelQueue;    # считываем и анализируем очередь удаления
&ProcessDelQueue; # пытаемся удалить все учетные записи
&DisposeDelQueue; # удаляем учетную запись либо из основной
```

```
                # базы данных, либо записываем ее обратно в
                # очередь, если возникли какие-то проблемы

# считываем очередь удаления в структуру данных $queue
sub ReadDelQueue{
    open(DEL,$accountdir.$delqueue) or
      die "Unable to open ${accountdir}${delqueue}:$!\n";
    read(DEL, $queuecontents, -s DEL);
    close(DEL);
    $queue = XMLin("<queue>".$queuecontents."</queue>",
                   keyattr => ["login"]);
}

# обходим в цикле структуру данных, пытаясь удалить учетную
# запись при каждом запросе (т. е. для каждого ключа)
sub ProcessDelQueue{
    foreach my $login (keys %{$queue->{account}}){
        $result = &$accountdel($login,
                    $queue->{account}->{$login});
        if (!$result){
            $queue->{account}->{$login}{status} = "deleted";
        }
        else {
            $queue->{account}->{$login}{status} =
                                "error:$result";
        }
    }
}

# считываем основную базу данных и затем вновь обходим в цикле
# структуру $queue. Для каждой учетной записи со статусом
# "deleted," изменяем информацию в основной базе данных. Затем
# записываем в базу данных. Все, что нельзя удалить, помещаем
# обратно в файл очереди удаления. Файл перезаписывается.
sub DisposeDelQueue{
    &ReadMainDatabase;

    foreach my $login (keys %{$queue->{account}}){
        if ($queue->{account}->{$login}{status} eq "deleted"){
            unless (exists $maindb->{account}->{$login}){
                warn "Could not find $login in $maindata\n";
                next;
            }
            $maindb->{account}->{$login}{status} = "deleted";
            $maindb->{account}->{$login}{deletion_date} = time;
            delete $queue->{account}->{$login};
            next;
       }
    }

    &WriteMainDatabase;
```

```
    # все, что сейчас осталось в $queue, - это учетные записи,
    # которые нельзя удалить
    open(DEL,">".$accountdir.$delqueue) or
      die "Unable to open ".$accountdir.$delqueue.":$!\n";
        if (scalar keys %{$queue->{account}}){
        print DEL XMLout(&TransformForWrite($queue),
                         rootname => undef);
    }
    close(DEL);
}

sub ReadMainDatabase{
    open(MAIN,$accountdir.$maindata) or
      die "Unable to open ".$accountdir.$maindata.":$!\n";
    read (MAIN, $dbcontents, -s MAIN);
    close(MAIN);
    $maindb = XMLin("<maindb>".$dbcontents."</maindb>",
                    keyattr => ["login"]);
}

sub WriteMainDatabase{
    # замечание: было бы *гораздо безопаснее* записывать данные
    # сначала во временный файл и только если они были записаны
    # успешно, записывать их окончательно
    open(MAIN,">".$accountdir.$maindata) or
      die "Unable to open ".$accountdir.$maindata.":$!\n";
    print MAIN XMLout(&TransformForWrite($maindb),
         rootname => undef);
    close(MAIN);
}
```

Можно написать еще множество сценариев. Например, мы могли бы применять сценарии, осуществляющие экспорт данных и проверку согласованности. В частности, совпадает ли домашний каталог пользователя с типом учетной записи из основной базы данных? Входит ли пользователь в нужную группу? Нам не хватит места, чтобы рассмотреть весь спектр таких программ, поэтому завершим этот раздел небольшим примером экспортирования данных. Речь уже шла о том, что хотелось бы завести отдельные списки рассылки для пользователей различного типа. В следующем примере из основной базы данных считываются данные и создается набор файлов, содержащих имена пользователей (по одному файлу для каждого типа пользователей):

```
use Account;          # только чтобы найти файлы
use XML::Simple;

&InitAccount;
&ReadMainDatabase;
&WriteFiles;

# читаем основную базу данных в хэш списков хэшей
```

```
sub ReadMainDatabase{
    open(MAIN,$accountdir.$maindata) or
      die "Unable to open ".$accountdir.$maindata.":$!\n";
    read (MAIN, $dbcontents, -s MAIN);
    close(MAIN);
    $maindb = XMLin("<maindb>".$dbcontents."</maindb>",
                            keyattr => [""]);
}

# обходим в цикле списки, собираем списки учетных записей
# определенного типа и сохраняем им в хэше списков. Затем
# записываем содержимое каждого ключа в отдельный файл.
sub WriteFiles {
    foreach my $account (@{$maindb->{account}}){
        next if $account->{status} eq "deleted";
        push(@{$types{$account->{type}}},$account->{login});
    }

    foreach $type (keys %types){
        open(OUT,">".$maillists.$type) or
          die "Unable to write to
               ".$accountdir.$maillists.$type.": $!\n";
        print OUT join("\n",sort @{$types{$type}})."\n";
        close(OUT);
    }
}
```

Если посмотреть в каталог списков рассылки, то можно увидеть:

```
> dir
faculty  staff
```

Каждый из этих файлов содержит соответствующий список учетных записей пользователей.

Система учетных записей. Заключение

Рассмотрев все четыре компонента системы учетных записей, подведем итоги и поговорим о том, что было пропущено (в узком, а не в широком смысле):

Проверка ошибок

В нашей демонстрационной программе выполняется проверка лишь небольшого числа ошибок. Любая уважающая себя система учетных записей увеличивается на 40–50% в объеме из-за проверки ошибок на каждом шаге своего выполнения.

Масштабируемость

Наша программа, скорее всего, сможет работать на мелких и средних системах. Но каждый раз, когда встречается фраза «прочитать весь файл в память», это должно звучать для вас предупреждением.

Чтобы повысить масштабируемость, нужно по крайней мере изменить способ получения и хранилище данных. Модуль `XML::Twig` Мишеля Родригеса (Michel Rodriguez) может разрешить эту проблему, т. к. он работает с большими XML-документами, не считывая их при этом целиком в память.

Безопасность

Это относится к самому первому элементу списка выводов – проверке ошибок. Помимо таких громадных дыр, в смысле безопасности, как хранение паролей открытым текстом, мы также не выполняем никаких других проверок. Нет даже попыток убедиться, что используемым источникам данных, например, файлам очередей, можно доверять. Стоит добавить еще 20–30% кода, чтобы позаботиться о таких моментах.

Многопользовательская среда

В коде не предусмотрена возможность одновременной работы нескольких пользователей или даже нескольких сценариев. И это, вероятно, самый большой недочет созданной программы. Если одновременно запустить один сценарий, добавляющий учетные записи, и другой, дописывающий учетные записи в очередь, то вероятность повредить или потерять данные будет очень велика. Это настолько важная тема, что ее стоит обсудить перед тем, как завершить этот раздел.

Один из способов разобраться в многопользовательской среде с работой – добавить блокировку файлов. Блокировка позволяет нескольким сценариям действовать одновременно. Если сценарий собирается читать или писать в файл, он может попытаться сначала файл заблокировать. Если это возможно, значит, с файлом можно работать. Если его заблокировать нельзя (потому что другой сценарий использует этот файл), то сценарий знает, что запрещено выполнять операции, которые могут повредить данные. С блокировкой и многопользовательской работой связаны гораздо более серьезные сложности; обратитесь к любой информации по операционным или распределенным системам. Серьезные проблемы могут возникнуть при работе с файлами, расположенными на сетевых файловых системах, где может и не быть хорошего механизма блокировки. Вот несколько советов, которые могут вам пригодиться, если вы коснетесь этой темы при использовании Perl.

- Существуют мудрые способы уходить от проблем. Мой любимый способ – использовать программу *lockfile*, входящую в состав популярной программы фильтрации почты *procmail*, которую можно найти на *http://www.procmail.org*. Процедура установки *procmail* принимает усиленные меры, чтобы определить безопасные стратегии блокировки для используемой файловой системы. *lockfile* делает именно то, что можно ожидать, глядя на ее название, скрывая при этом основные сложности.

- Если вы не хотите применять внешнюю программу, существует масса модулей, выполняющих блокировку. Например, `File::Flock` Дэвида Мюир Шарнофа (David Muir Sharnoff), `File::LockDir` из книги «Perl Cookbook» («Perl: Библиотека программиста») Тома Кристиансена (Tom Christiansen) и Натана Торкингтона (Nathan Torkington) (O'Reilly) и его версия для Win95/98 Вильяма Херейры (William Herrera) под названием `File::FlockDir`, `File::Lock` Кеннета Альбановски (Kenneth Albanowski), `File::Lockf` Поля Хенсона (Paul Henson) и `Lockfile::Simple` Рафаеля Манфреди (Raphael Manfredi). В основном, они отличаются интерфейсом, хотя `File::FlockDir` и `Lockfile::Simple` пытаются выполнять блокировку, не используя функцию `flock()` из Perl. Они могут быть полезными на таких платформах, как MacOS, где эта функция не поддерживается. Осмотритесь и выбирайте тот модуль, который больше всего вам подходит.
- Блокировку проще всего выполнить правильно, если не забыть заблокировать файл, перед тем как изменять данные (или считывать данные, которые могли измениться), и снимать блокировку только *после* того, как убедитесь, что данные были записаны (например, после того как файл будет закрыт). Подробную информацию по этой теме можно найти в упомянутой уже «книге рецептов», в списке часто задаваемых вопросов Perl FAQ, в документации по функции `flock()` из модуля `DB_File` и из документации по Perl.

Мы завершаем наш разговор об администрировании пользователей и о том, как можно перевести эти операции на другой уровень, применив подход архитектора. В этой главе мы уделили особое внимание началу и концу жизненного цикла учетной записи. В следующей главе мы поговорим о том, что делают пользователи между этими двумя моментами.

Информация о модулях из этой главы

Название	Идентификатор на CPAN	Версия
`User::pwent` (входит в состав Perl)		
`File::stat` (входит в состав Perl)		
`Win32::AdminMisc` (можно найти на *http://www.roth.net*)		20000117
`Win32::Perms` (можно найти на *http://www.roth.net*)		20000216
`Win32::Lanman` (можно найти на *ftp://ftp.roth.net/pub/ntperl/Others/Lanman/)*		1.05
`IO::File` (входит в состав Perl)	GBARR	1.20
`XML::Writer`	DMEGG	0.30
`XML::Parser`	COOPERCL	2.27

Название	Идентификатор на CPAN	Версия
`Data::Dumper`	GSAR	2.101
`XML::Simple`	GRANTM	1.01
Expect.pm	AUSCHUTZ	1.07
`File::Path` (входит в состав Perl)		1.0401
`Win32::FileOp`	JENDA	0.10.4

Рекомендуемая дополнительная литература

Файлы паролей в Unix

http://www.freebsd.org/cgi/man.cgi. Здесь можно получить доступ в online-режиме к страницам руководств для *BSD и других вариантов Unix. Это очень удобный способ сравнить форматы файлов и команды системного администрирования (*useradd* и пр.) для нескольких операционных систем.

«Practical Unix & Internet Security», (2nd Edition), Simson Garfinkel, Gene Spafford (O'Reilly, 1999). Отличный источник информации о файлах паролей.

Администрирование пользователей в NT

http://Jenda.Krynicky.cz – еще один сайт с полезными модулями в Win32 для администрирования пользователей.

http://windows.microsoft.com/windows2000/en/server/help/ – справка Windows 2000. (Переходите к разделу Active Directory→Concepts→Understanding Active Directory→Understanding Groups). Это хороший обзор новых механизмов групп в Windows 2000.

http://www.activestate.com/support/mailing_lists.htm. Здесь можно найти списки рассылки Perl-Win32-Admin и Perl-Win32-Users. Оба списка и их архивы представляют собой просто бесценный источник информации для программистов Win32.

«Win32 Perl Programming: The Standard Extensions», Dave Roth (Macmillan Technical Publishing, 1999) в настоящее время лучший источник по программированию модулей для Win32 Perl.

«Windows NT User Administration», Ashley J. Meggitt, Timothy D. Ritchey (O'Reilly, 1997).

http://www.mspress.com. Издатели Microsoft NT Resource Kit. Они также предлагают возможность подписки для получения доступа к самым последним утилитам из RK.

http://www.roth.net. Домашняя страница для `Win32::AdminMisc`, `Win32::Perms` и других модулей для Win32, используемых для администрирования пользователей.

XML

За последние два года появилось огромное количество материала по XML. Приведенные ниже источники информации – это лучшее, что, на мой взгляд, существует для тех, кто ничего не знает о XML. Когда я писал эту книгу, изданий по XML для Perl еще не было, но мне известно, что несколько подобных проектов уже существует.

http://msdn.microsoft.com/xml и *http://www.ibm.com/developer/xml* – оба содержат обилие информации. И Microsoft, и IBM очень серьезно настроены по отношению к XML.

http://www.activestate.com/support/mailing_lists.htm – содержит список рассылки Perl-XML. Он (и его архивы) один из лучших источников данной информации.

http://www.w3.org/TR/1998/REC-xml-19980210. Спецификация XML 1.0. Любой, кто делал что-то на XML, наверняка читал спецификацию. Если вам нужно что-либо более подробное, чем справочник, я советую почитать версии с комментариями.

http://www.xml.com. Хороший источник статей и ссылок, посвященных XML. Кроме того, здесь можно найти отличную версию спецификации с комментариями Тима Брэя (Tim Bray), одного из ее авторов.

«XML: The Annotated Specification», Bob DuCharme (Prentice Hall, 1998). Еще одна отличная версия спецификации с комментариями и примерами кода на XML.

«XML Pocket Reference», Robert Eckstein (O'Reilly, 1999). Краткое, но на удивление полное введение в XML для нетерпеливых.

Прочее

http://www.mcs.anl.gov/~evard. Домашняя страница Реми Эварда (Remy Evard). Использование нескольких баз данных для автоматического генерирования конфигурационных файлов – это лучший прием, который показан в нескольких местах моей книги; спасибо Эварду за идею этого метода. И хотя сейчас подобный прием применяется на многих сайтах, я впервые столкнулся с ним при знакомстве со средой Tenwen, которую он создал (как описано в статье, ссылка на которую есть с домашней страницы Эварда). Чтобы ознакомиться с работой этого метода, загляните в раздел «Implemented the Hosts Database».

http://www.rpi.edu/~finkej/. Содержит несколько статей Йона Финки (Jon Finke) по использованию реляционных баз данных в системном администрировании.

4

- *Управление процессами в MacOS*
- *Управление процессами в NT/2000*
- *Управление процессами в Unix*
- *Отслеживание операций с файлами и сетью*
- *Информация о модулях из этой главы*
- *Рекомендуемая дополнительная литература*

Действия пользователей

В предыдущей главе мы обсуждали информацию о пользователях, способы ее хранения и методы обработки. Настало время поговорить о том, как работать с пользователями, когда они активны в системе или сети.

Действия пользователей можно разделить на четыре типа:

Процессы

Пользователь может порождать, завершать, приостанавливать и продолжать работу процессов. Все эти процессы одновременно используют процессор, мощность которого конечна, и добавляют к заботам системного администратора еще и проблемы с ресурсами.

Операции с файлами

Большей частью такие операции, как чтение, запись, создание, удаление и т. д., происходят при взаимодействии какого-либо процесса пользователя с файловой системой. Однако в Unix такая картина не полна. Unix использует файловые системы в качестве шлюза не только к хранилищу файлов. Управление устройствами, каналы ввода/вывода и даже, в некоторых случаях, управление процессами и доступ к сети могут быть операциями с файлами. Администрирование файловых систем обсуждалось в главе 2 «Файловые системы», но сейчас стоит коснуться этой темы уже с точки зрения администрирования пользователей.

Сети

Пользователи могут получать и посылать данные через сетевые интерфейсы машины. Речь о сетях в этой книге еще пойдет, но здесь мы рассмотрим данный вопрос в другом аспекте.

Действия, специфичные для операционной системы

Последняя область – это множество возможностей, зависящих от операционной системы, обращаться к которым пользователь может через различные API. К ним относятся элементы управления графическим интерфейсом, использование разделяемой памяти, API для совместного использования файлов, звук и т. д. Категория возможностей настолько разнообразна, что подробно рассказать о ней в одной главе нельзя. Чтобы получить информацию по этим темам, я рекомендую посещать сайты, посвященные соответствующим операционным системам, например *http://www.macperl.com*.

Посмотрим, как работать с тремя из перечисленных областей в Perl. В каждой из операционных систем, упомянутых в книге, подход к этим способам различный, поэтому рассматривать их надо отдельно. Наиболее сходной среди них является Perl-функция `kill()`, но даже и она не реализована в MacOS. Мы рассмотрим каждую операционную систему, начиная с менее сложной (с точки зрения Perl). Поскольку нас интересует администрирование пользователей, внимание следует уделить работе с процессами, запущенными другими пользователями.

Управление процессами в MacOS

«Управление» – слишком громко сказано для функциональных возможностей, предоставляемых MacOS, поскольку последняя является не многопользовательской, а просто многозадачной операционной системой. Используя модуль `Mac::Processes`, можно взаимодействовать с менеджером процессов Macintosh (Macintosh Process Manager) при помощи MacOS Toolbox API для управления процессами. В случае частого применения этого модуля, стоит поискать руководство «Inside Macintosh:Processes» о работе с менеджером процессов.

При загрузке `Mac::Processes` с помощью стандартной директивы `use Mac::Processes`, инициализируется специальный хэш `%Process`. Этот хэш – магический, в нем всегда отображается состояние *текущего* процесса при помощи возможности Perl, именуемой «связанные переменные». Каждый раз при обращении к содержимому хэша `%Process` возвращается информация о процессах, запущенных в настоящий момент в системе. Чтобы просмотреть список серийных номеров текущих процессов (Process Serial Number , PSN – так в MacOS называются идентификаторы процессов), надо просто запросить список ключей этого хэша:

```
use Mac::Processes;
print map{"$_\n"} keys %Process;
```

Подробную информацию по любому процессу можно получить, поработав со значениями, возвращаемыми для каждого ключа. Каждая запись в хэше содержит объект, представляющий структуру `ProcessInfo`.

Чтобы получить поля этой структуры, необходимо вызвать соответствующие методы объекта. Более подробно о том, что представляет собой каждое поле, можно узнать в книге «Inside Macintosh:Processes». В настоящее время существуют методы `processName()`, `processNumber()`, `processType()`, `processSignature()`, `processSize()`, `processMode()`, `processLocation()`, `processLauncher()`, `processLaunchDate()`, `processActiveTime()` и `processAppSpec()`.

Нетрудно вывести список запущенных процессов с их именами:

```
use Mac::Processes;
while(($psn, $psi) = each (%Process)){
  $name = $psi->processName();
  write;
}

format STDOUT_TOP =
Process Serial Number      Process Name
====================       ================================
.

format STDOUT =
@<<<<<<                    @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$psn,                      $name
.
```

Результат таков:

```
Process Serial Number      Process Name
====================       ========================
8192                       FaxMonitor
8193                       Queue Watcher
8194                       Finder
8195                       Serial Port Monitor
8198                       MacPerl
```

Теперь, когда вы знаете, какие процессы у вас запущенны, совершенно естественно попытаться ими управлять. К сожалению, здесь практически ничего нельзя сделать. Самое большое, на что мы способны, – это перевести процесс в интерактивный режим при помощи `SetFrontProcess($psn)`. У нас даже нет возможности напрямую его завершить (Perl-функция `kill()` не реализована). Лучшее, что можно сделать, – послать выполняющемуся приложению событие AppleEvent, чтобы сообщить, что процесс должен быть завершен. Самый простой способ сделать это – применить модуль `Mac::Apps::Launch` Криса Нандора (Chris Nandor). В нем есть функция `QuitApps()`, которая позволяет завершить работу приложения, располагая его ID. В `Mac::Apps::Launch` есть еще несколько полезных функций для запуска приложений и перевода их из/в интерактивный режим. Делается это так же, как и при использовании `Mac::Processes`.

Теперь наступает время операционной системы, в которой управление процессами менее ограничено.

Управление процессами в NT/2000

Мы вкратце рассмотрим четыре различных способа работы с процессами в NT/2000, поскольку каждый из них открывает перед нами двери к увлекательным возможностям, лежащим за пределами нашего обсуждения. Сначала мы остановимся на двух задачах: поиске всех запущенных процессов и завершении работы некоторых из них.

Используем Microsoft Resource Kit

В главе 3 «Учетные записи пользователей» упоминалось, что NT Resource Kit является отличным источником для сценариев и информации. Из этого пакета будут использоваться две программы: *pulist.exe* и *kill.exe*. Первая выводит список процессов, вторая – «убивает» их. В этом пакете есть еще одна утилита *tlist.exe*, похожая на *pulist.exe*, которая может вывести все процессы в списке, удобном для чтения, но ей не достает некоторых возможностей *pulist.exe*. Например, *pulist.exe* может вывести список процессов не только текущего, но и другого компьютера.

Вот фрагмент из вывода команды *pulist*:

```
Process          PID  User
TAPISRV.EXE      119  NT AUTHORITY\SYSTEM
TpChrSrv.exe     125  NT AUTHORITY\SYSTEM
RASMAN.EXE       131  NT AUTHORITY\SYSTEM
mstask.exe       137  NT AUTHORITY\SYSTEM
mxserver.exe     147  NT AUTHORITY\SYSTEM
PSTORES.EXE      154  NT AUTHORITY\SYSTEM
NDDEAGNT.EXE     46   OMPHALOSKEPSIS\Administrator
explorer.exe     179  OMPHALOSKEPSIS\Administrator
SYSTRAY.EXE      74   OMPHALOSKEPSIS\Administrator
cardview.exe     184  OMPHALOSKEPSIS\Administrator
ltmsg.exe        167  OMPHALOSKEPSIS\Administrator
daemon.exe       185  OMPHALOSKEPSIS\Administrator
```

Применять *pulist.exe* из Perl очень просто. Вот один из способов:

```
$pulistexe = "\\bin\\PULIST.EXE";  # местоположение программы
open(PULIST,"$pulistexe|") or die "Невозможно выполнить  $pulistexe:$!\n";

scalar <PULIST>;                   # удаляем первую строку заголовка
while(defined($_=<PULIST>)){
    ($pname,$pid,$puser) = /^(\S+)\s*(\d+)\s*(.+)/;
    print "$pname:$pid:$puser\n";

close(PULIST);
```

Вторая упомянутая программа – это *kill.exe*. Ее тоже просто использовать. В качестве аргумента она принимает либо идентификатор процесса, либо часть имени задачи. В целях безопасности я рекомендую использовать идентификаторы процессов, потому что иначе очень легко убить не тот процесс, который нужно.

Программа *kill.exe* использует два различных способа завершения работы процессов. Один из них – это так называемая «вежливая смерть»: *kill.exe <process id>* попросит подтверждения на завершение работы процесса. Но если добавить к командной строке ключ */f*, действия *kill.exe /f <process id>* будут скорее напоминать манеру истинных Perl-функций – он завершит работу процесса с особенной предвзятостью.

Использование модуля Win32::IProc

Второй подход – применять модуль `Win32::IProc` Амина Мюлэй Рамдэна (Amine Moulay Ramdane). И хотя название подсказывает, казалось бы, очевидный выбор, но `Win32::Iproc`, в действительности, гораздо полезнее для нас, чем `Win32::Process`. У `Win32::Process` есть один значительный недостаток, который тут же выводит модуль из борьбы: он создан для работы с процессами, которые были запущены им самим. В то время как нас больше интересуют процессы, запущенные *другими* пользователями. Если вам не удается установить модуль `Win32::IProc`, загляните в раздел «Информация о модулях из этой главы» .

Сначала необходимо создать объект процесса подобным образом:

```
use Win32::IProc;

# обратите внимание на регистр. Обязательно должно быть "IProc"
$pobj = new Win32::IProc or die "Невозможно создать объект proccess: $!\n";
```

Такой объект обычно используется в качестве трамплина, с которого запускаются методы объекта. Например, чтобы получить список всех запущенных процессов, можно написать:

```
$pobj-> EnumProccesses(\@processlist) or
   die "Невозможно получить список процессов:$!\n";
```

`@processlist` – это массив ссылок на анонимные хэши. В каждом анонимном хэше есть два ключа: `ProcessName` и `ProcessId` с их значениями. Такой код позволяет аккуратно вывести нужную информацию:

```
use Win32::IProc;

$pobj=new Win32::IProc or die "Невозможно создать объект process: $!\n";

$pobj->EnumProcesses(\@processlist) or
   die "Невозможно получить список процессов:$!\n";
```

```
foreach $process (@processlist){
  $pid  = $process->{ProcessId};
  $name = $process->{ProcessName};
  write;
}

format STDOUT_TOP =
Process ID      Process Name
==========      ================================
.
format STDOUT =
@<<<<<<<        @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$pid,           $name
.
```

Получаем результат:

```
Process ID      Process Name
==========      ================================
0               System-Idle
2               System
25              smss.exe
39              winlogon.exe
41              services.exe
48              lsass.exe
78              spoolss.exe
82              DKSERVICE.EXE
...
```

Отличие от действий *pulist.exe* заключается в том, что `Win32::IProc` не может сообщить вам пользовательский контекст для процесса. Если эта информация для вас важна, то следует применять *pulist.exe*.

***pulist.exe* может вывести только один тип информации, а что может `Win32::IProc` — сейчас будет ясно. Допустим, вам хочется узнать не только о запущенных процессах, но и о том, какие исполняемые программы и динамически загружаемые библиотеки (*.dll*) использует каждый процесс. Получить эту информацию просто:**

```
# импортируем постоянную FULLPATH, чтобы показывать пути к
# библиотекам, может быть и NOPATH
use Win32::IProc "FULLPATH";
$pobj = new Win32::IProc;

$pobj->EnumProcesses(\@processlist) or
  die "Невозможно получить список процессов:$!\n";

foreach $process (@processlist){
  print "\n",$process->{ProcessName},
        "\n",('=' x length($process->{ProcessName})),"\n";

  $pobj->GetProcessModules($process->{ProcessId},
```

```
                                \@modules,FULLPATH);
    print join("\n",map {lc $_->{ModuleName}} @modules),"\n";

}
```

`GetProcessModules()` получает идентификатор процесса, ссылку на массив и флаг, говорящий о том, возвращать ли полный путь к библиотеке. Элементами массива, на который мы ссылаемся, являются ссылки на анонимные хэши, содержащие информацию о каждой библиотеке, используемой этим процессом. В нашем примере собираются имена всех библиотек. `map()` используется для того, чтобы обойти весь массив ссылок, разыменовать каждый анонимный хэш и получить значение ключа `ModuleName`.

Вот отрывок полученных данных:

```
smss.exe
========
\systemroot\system32\smss.exe
c:\winnt\system32\ntdll.dll

winlogon.exe
============
\??\c:\winnt\system32\winlogon.exe
c:\winnt\system32\ntdll.dll
c:\winnt\system32\msvcrt.dll
c:\winnt\system32\kernel32.dll
c:\winnt\system32\advapi32.dll
c:\winnt\system32\user32.dll
c:\winnt\system32\gdi32.dll
c:\winnt\system32\rpcrt4.dll
c:\winnt\system32\userenv.dll
c:\winnt\system32\shell32.dll
c:\winnt\system32\shlwapi.dll
c:\winnt\system32\comctl32.dll
c:\winnt\system32\netapi32.dll
c:\winnt\system32\netrap.dll
c:\winnt\system32\samlib.dll
c:\winnt\system32\winmm.dll
c:\winnt\system32\cwcmmsys.dll
c:\winnt\system32\cwcfm3.dll
c:\winnt\system32\msgina.dll
c:\winnt\system32\rpclts1.dll
c:\winnt\system32\rpcltc1.dll...
```

Но давайте пойдем еще дальше. Совсем немного усилий следует приложить, чтобы больше узнать о запущенных процессах. Для получения необходимой информации сначала нужно определить дескриптор этого процесса.

Дескриптор процесса можно рассматривать как открытое соединение с данным процессом. Чтобы выяснить разницу между дескриптором процесса и его идентификатором, проведем аналогию со стоянкой прицепов. Если каждый прицеп на стоянке считать процессом, то адрес прицепа можно считать идентификатором процесса. Это способ найти нужный прицеп. Дескриптор процесса – это что-то наподобие телефонных линий, труб и проводов, подведенных к прицепу. Когда прицеп подсоединен к этим коммуникациям, вы можете не только найти нужный прицеп, но и передавать в него информацию.

Для получения дескриптора процесса, если у нас есть его идентификатор, используем метод `Open()` из модуля `Win32::IProc`. `Open()` принимает идентификатор процесса, флаг доступа, флаг наследования и ссылку на скалярное значение, в котором хранится дескриптор. В следующем примере запроса мы будем использовать флаги доступа, которых достаточно для получения информации о процессе. Подробную информацию об этих флагах можно найти в документации по `Win32::IProc` и в разделе «Processes and Threads» документации Win32 SDK по основным службам, которую можно найти на *http://msdn.microsoft.com*. Дескрипторы процессов, открытые при помощи `Open()`, необходимо закрыть при помощи `CloseHandle()`.

Зная дескриптор процесса, можно использовать метод `Kill()` для завершения его работы:

```
# завершить процесс и заставить его вернуть именно этот код
$pobj->Kill($handle,$exitcode);
```

Но дескрипторы процессов следует применять не только для того, чтобы завершать работу процесса. Например, можно использовать такие методы, как `GetStatus()`, чтобы больше узнать о процессе. Вот пример кода, который выводит информацию о времени для заданного идентификатора процесса:

```
use Win32::IProc qw(PROCESS_QUERY_INFORMATION INHERITED DIGITAL);

$pobj = new Win32::IProc;

$pobj->Open($ARGV[0],PROCESS_QUERY_INFORMATION,INHERITED, \$handle) or
  warn "Невозможно получить дескриптор:".$pobj->LastError()."\n";

# DIGITAL = время в понятном формате
$pobj->GetStatus($handle,\$statusinfo,DIGITAL);

$pobj->CloseHandle($handle);

while (($procname,$value)=each %$statusinfo){
  print "$procname: $value\n";
}
```

В результате получается что-то приблизительно следующее:

```
KernelTime: 00:00:22:442:270
ExitDate:
ExitTime:
CreationDate: 29/7/1999
CreationTime: 17:09:28:100
UserTime: 00:00:11:566:632
```

Теперь известно, когда процесс был запущен и сколько системного времени он занимает. Поля `ExitDate` и `ExitTime` пусты, поскольку процесс все еще активен. Вы могли бы спросить, как эти поля, в принципе, могут оказаться не пустыми, если для получения дескриптора нужно использовать идентификатор работающего процесса? На этот вопрос есть два ответа. Во-первых, можно получить дескриптор для работающего процесса, а затем заставить этот процесс завершиться до того, как вы закроете дескриптор. `GetStatus()` в таком случае вернет информацию о завершении работы для умершего процесса. Вторая возможность получить эту информацию – использовать метод `Create()`, о котором мы пока еще не знаем.

`Create()` позволяет запускать процессы из `Win32::IProc` так же, как и в случае с `Win32::Process`. Если запустить процесс при помощи модуля, то объект процесса (`$pobj`), который до сих пор не обсуждался, будет содержать информацию о самом процессе и потоках. Обладая этой информацией, вы сможете делать любопытные вещи, например, манипулировать приоритетами потоков и окнами этого процесса. Мы не собираемся рассматривать эти возможности, но упомянуть о них следует, чтобы спокойно перейти к следующему модулю.

Использование модуля Win32::Setupsup

Если упоминание о манипуляции окнами процесса, приведенное в конце предыдущего раздела, возбудило ваше любопытство, вам понравится наш следующий подход. На этот раз мы рассмотрим модуль `Win32::Setupsup` Йена Хелберга (Jens Helberg). Он называется «Setupsup», потому что первоначально был разработан для использования при установке программного обеспечения (при частом применении программы *Setup.exe*).

Некоторые инсталляторы можно запускать в так называемом «тихом режиме» для полной автоматизации установки. В этом режиме они не задают никаких вопросов и не просят нажимать кнопки «OK», освобождая администратора от необходимости сидеть нянькой при инсталляторе. Если такой режим не поддерживается механизмом установки (а подобных случаев очень много), это сильно усложняет жизнь системного администратора. `Win32::Setupsup` помогает справиться с такими трудностями. Он позволяет найти информацию о работающих про-

цессах и работать с ними (либо завершить эти процессы, если вы того пожелаете).

Обратитесь к разделу «Информация о модулях из этой главы», чтобы узнать, как получить и установить модуль Win32::Setupsup.

Используя Win32::Setupsup, получить список выполняемых процессов очень просто. Вот слегка измененная версия примера, который можно увидеть в последнем разделе:

```
use Win32::Setupsup;

$machine = ""; # получаем список на текущей машине

Win32::Setupsup::GetProcessList($machine, \@processlist, \@threadlist) or
die "Ошибка получения списка процессов: ".Win32::Setupsup::GetLastError()."\n";

pop(@processlist); # удалить фальшивую запись, всегда
                   # добавляемую к списку
foreach $processlist (@processlist){
  $pid  = $processlist->{pid};
  $name = $processlist->{name};
  write;
}

format STDOUT_TOP =
Process ID      Process Name
==========      ================================
.
format STDOUT =
@<<<<<<<        @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$pid,           $name
.
```

Завершение процессов тоже очень просто:

```
KillProcess($pid, $exitvalule, $systemprocessflag) or
  die "Невозможно завершить процесс: ".Win32::Setupsup::GetLastError()."\n";
```

Два последних аргумента необязательны. Первый завершает процесс и, соответственно, устанавливает его код завершения (по умолчанию это 0). Второй аргумент позволяет вам завершать системные процессы (при условии, что у вас есть право Debug Programs).

Но все это очень скучно. Мы можем перейти на другой уровень манипулирования процессами, взаимодействуя с окнами, которые открыты запущенным процессом. Чтобы вывести список окон, доступных на рабочем столе, применим:

```
Win32::Setupsup::EnumWindows(\@windowlist) or
die "Ошибка получения списка процессов:
       ".Win32::Setupsup::GetLastError()."\n";
```

@windowlist теперь содержит список дескрипторов окон, которые выглядят как обычные числа, если их напечатать. Чтобы узнать больше о каждом окне, можно использовать несколько различных функций. Например, чтобы прочитать заголовки всех окон, воспользуемся функцией GetWindowText():

```
use Win32::Setupsup;

Win32::Setupsup::EnumWindows(\@windowlist) or
die "Ошибка получения списка процессов:
       ".Win32::Setupsup::GetLastError()."\n";

foreach $whandle (@windowlist){
    if (Win32::Setupsup::GetWindowText($whandle,\$text)){
      print "$whandle: $text","\n";
    }
    else {
      warn "Невозможно получить текст для $whandle" .
            Win32::Setupsup::GetLastError()."\n";
    }
}
```

Вот небольшой отрывок получаемых данных:

```
66130: chapter02 - Microsoft Word
66184: Style
194905150:
66634: setupsup - WordPad
65716: Fuel
328754: DDE Server Window
66652:
66646:
66632: OleMainThreadWndName
```

Как видите, у некоторых окон есть заголовки, а у некоторых их нет. Внимательные читатели могли заметить в этом отрывке еще кое-что любопытное. Окно 66130 принадлежит сеансу Microsoft Word, запущенному в настоящий момент (в нем набиралась эта глава). Окно 66184 смутно напоминает название еще одного окна, связанного с Microsoft Word. Как мы можем узнать, действительно ли эти окна взаимосвязаны?

В Win32::Setupsup также есть функция EnumChildWindows(), которая позволяет вывести список дочерних окон для любого окна. Используем ее для вывода иерархии текущего окна:

```
use Win32::Setupsup;

# получаем список окон
Win32::Setupsup::EnumWindows(\@windowlist) or
die "Ошибка получения списка процессов:
       ".Win32::Setupsup::GetLastError()."\n";
```

```
# превращаем список дескрипторов окон в хэш
# ЗАМЕЧАНИЕ: в результате преобразования
# элементами хэша становятся обычные числа,
# а не дескрипторы окон. Некоторые функции,
# например GetWindowProperties
# (которую мы скоро рассмотрим),
# не могут использовать эти преобразованные числа.
# Будьте осторожны.
for (@windowlist){$windowlist{$_}++;}

# проверяем наличие дочерних окон для каждого окна
foreach $whandle (@windowlist){
    if (Win32::Setupsup::EnumChildWindows($whandle,\@children)){
       # сохраняем отсортированный список дочерних окон
       $children{$whandle} = [sort {$a <=>$b} @children];

       # удаляем все дочерние окна из главного хэша,
       # в результате всех итераций %windowlist будет
       # содержать только родительские окна без
       # соответствующих дочерних[1]
       foreach $child (@children){
         delete $windowlist{$child};
       }
    }
}

# обходим в цикле список родительских окон
# и тех окон, у которых нет дочерних,
# и рекурсивно печатаем дескриптор каждого
# окна и его дочерние окна (если они есть)
foreach my $window (sort {$a <=> $b} keys %windowlist){
 &printfamily($window,0);
}

# выводим дескриптор заданного окна и его дочерних окон
# (рекурсивно)
sub printfamily {
  # начальное окно, насколько глубоко мы ушли по дереву?
  my($startwindow,$level) = @_;

  # выводим дескриптор окна с соответствующим отступом
  print((¨  ¨ x $level).¨$startwindow\n¨);

  return unless (exists $children{$startwindow}); # дочерних окон нет,
                                                  # дело сделано
```

[1] **Авторский комментарий здесь выглядел так: «# remove all children from the hash, we won't iterate over them» (удаляем все дочерние окна из хэша, мы не будем напрямую рассматривать их в цикле).** – *Примеч. науч. ред.*

```
    # в противном случае мы должны рекурсивно обойти дочерние окна
    $level++;
    foreach $childwindow (@{$children{$startwindow}}){
       &printfamily($childwindow,$level);
    }
}
```

Есть еще одна функция, о которой надо сказать, перед тем как двигаться дальше: `GetWindowProperties()`. Функция `GetWindowProperties()` вмещает в себя остальные свойства окон. Например, используя `GetWindowProperties()`, можно получить идентификатор процесса, создавшего конкретное окно. Это разумно совместить с некоторыми из только что рассмотренных возможностей модуля `Win32::IProc`.

В документации к модулю `Win32::Setupsup` есть список свойств, и к ним можно обратиться. Используем одно из них для написания очень простой программы, которая выведет размеры окна на экране. `GetWindowProperties()` принимает три аргумента: дескриптор окна, ссылку на массив, содержащий имена запрашиваемых свойств, и ссылку на хэш, где будут храниться результаты запроса. Вот какой код мы применим для этого:

```
Win32::Setupsup::GetWindowProperties($ARGV[0],["rect","id"],\%info);

print "\t" . $info{rect}{top} . "\n";
print $info{rect}{left} . " -" . $ARGV[0] .
      "- " . $info{rect}{right} . "\n";
print "\t" . $info{rect}{bottom} . "\n";
```

Вывод получается несколько вычурным. Вот как выглядит вывод размеров (координат верхнего, левого, правого и нижнего края) окна с дескриптором `66180`:

```
    154
272 -66180- 903
    595
```

`GetWindowProperties()` возвращает специальную структуру данных только для одного свойства `rect`. Все остальные будут представлены в хэше в виде обычных ключей и значений. Если вы не уверены в свойствах, возвращаемых Perl для конкретного окна, воспользуйтесь утилитой *windowse*, которую можно найти на *http://greatis.virtualave.net/products.htm*.

Разве теперь, когда мы знаем, как определить различные свойства окон, не было бы логично научиться изменять некоторые из этих свойств? Например, было бы полезно изменять заголовок окна. С такими возможностями мы могли бы создавать сценарии, использующие заголовок окна в качестве индикатора состояния:

```
"Prestidigitation In Progress ... 32% complete"
```

Чтобы внести эти изменения, достаточно одного вызова функции:

```
Win32::Setupsup::SetWindowText($handle,$text);
```

Свойство `rect` тоже можно установить таким образом. Следующие строки заставляют указанное окно переместиться в заданную позицию экрана:

```
use Win32::Setupsup;

$info{rect}{left}  = 0;
$info{rect}{right} = 600;
$info{rect}{top}   = 10;
$info{rect}{bottom}= 500;
Win32::Setupsup::SetWindowProperties($ARGV[0],\%info);
```

Самую впечатляющую функцию я приберег напоследок. При помощи `SendKeys()` можно послать произвольные комбинации клавиш любому окну на рабочем столе. Например:

```
use Win32::Setupsup;

$texttosend = "\\DN\\Low in the gums";
Win32::Setupsup::SendKeys($ARGV[0],$texttosend,'',0);
```

В результате, в указанное окно будет послан текст, предваряемый символом «курсор вниз». Аргументы `SendKeys()` очень просты: дескриптор окна, посылаемый текст, флаг, определяющий, нужно ли активизировать окно для каждого сочетания клавиш, и необязательный интервал между сочетаниями клавиш. Коды специальных символов, таких как «курсор вниз», окружаются обратными слэшами. Список допустимых кодов можно найти в документации к модулю.

С помощью этого модуля мы попадаем на иной уровень управления процессами. Теперь мы можем удаленно управлять приложениями (и частями операционной системы), не взаимодействуя явно с этими приложениями. Нам не нужна поддержка командной строки или специальных API. У нас есть возможность писать сценарии для GUI, что очень полезно во множестве задач системного администрирования.[1]

[1] Для создания сценариев по графическому интерфейсу может быть полезен модуль `Win32Guitest` Эрнесто Гуисадо (Ernesto Guisado). Он поддерживает те же возможности, что и `Win32::Setupsup`.

Использование инструментария управления окнами (Window Management Instrumentation, WMI)

Перед тем как перейти еще к одной операционной системе, рассмотрим последний подход к управлению процессами в NT/2000. Этот подход следовало бы назвать «Страной будущего», поскольку в нем используется пока еще не очень распространенная, но уже пробивающаяся технология. Инструментарий управления окнами (WMI) доступен в Windows 2000 (и в NT4.0 с установленным SP4+).[1] Со временем, когда Windows 2000 широко распространится, WMI вполне может стать важной частью администрирования NT/2000.

К сожалению, WMI относится к числу технологий не для слабонервных, потому что очень быстро становится чересчур сложной. Она основана на объектно-ориентированной модели, которая позволяет представить не только данные, но и отношения между объектами. Например, можно создать связь между веб-сервером и дисковым массивом RAID, в котором хранятся данные с этого сервера, обеспечивающую, в случае неисправности массива RAID, сообщение и о проблеме с веб-сервером. Не желая вдаваться во все эти сложности, мы дадим здесь лишь поверхностный обзор WMI в небольшом и простом введении, сопровождаемом примерами.

Если вы хотите познакомиться с этой технологией подробнее, я рекомендую загрузить документацию по WMI, обучающее руководство «LearnWBM» и WMI SDK из раздела «WMI» сайта *http://msdn.microsoft.com/developer/sdk*. Также взгляните на информацию, представленную на веб-сайте Distributed Management Task Force на *http://www.dtmf.org*. Тем временем, начнем с краткого экскурса.

WMI – это реализация и расширение (от Microsoft) неудачно названной инициативы Web-Based Enterprise Management или WBEM. И хотя такое название вызывает в воображении что-то связанное с броузерами, эта технология не имеет практически ничего общего с World Wide Web. Компании, входившие в состав Distributed Management Task Force (DMTF), хотели придумать что-то, что могло бы упростить выполнение задач управления при помощи броузеров. Забыв про название, можно сказать, что WBEM определяет модель данных для информации управления. WBEM обеспечивает спецификацию для организации, доступа и перемещения этих данных. WBEM также предлагает связующий интерфейс для работы с данными, доступными из других протоколов, например, Simple Network Management Protocol (SNMP) (о котором мы будем говорить в главе 10 «Безопасность и наблюдение за сетью») и Common Management Information Protocol (CMIP).

---

[1] Раздел «Download SDK» из секции о WMI на *http://msdn.microsoft.com/developer/sdk* позволяет загрузить библиотеки, необходимые для запуска WMI на машине с NT4.0SP4 (или выше).

Данные в WBEM организованы при помощи *Общей информационной модели* (Common Information Model, CIM). CIM – источник силы и сложности в WBEM/WMI. Она предоставляет расширяемые модели данных, содержащие объекты и классы объектов для любой физической или логической сущности, которой можно захотеть управлять. Например, в ней есть классы объектов для целых сетей и объекты для отдельного слота машины. Существуют объекты для настроек как аппаратного обеспечения, так и приложений программного обеспечения. Помимо этого CIM позволяет определять классы объектов, описывающие связи между другими объектами.

Модель данных документирована в двух частях: в *спецификации* CIM и *схеме* CIM. Первая описывает, *как* (как данные определяются, их связь с другими стандартами управления и т. д.); вторая – *что* (т. е. сами объекты). Это деление может напомнить о связи SNMP SMI и MIB (подробный материал в главе 10).

На практике вы будете чаще обращаться к схеме CIM, чем к спецификации, когда вы освоитесь с тем, как представляются данные. Формат схемы, названный *форматом управляемых объектов* (Managed Object Format, MOF), довольно просто читать.

Схема CIM состоит из двух слоев:

- *Центральная модель* (*core model*) для объектов и классов, полезна для всех типов взаимодействия WBEM.
- *Общая модель* (*common model*) для объектов, которые не зависят от создателя и операционной системы. Внутри общей модели в настоящее время определены пять областей: системы, устройства, приложения, сети и физический уровень.

На вершине этих двух слоев может быть любое число *расширенных схем* (*Extension schema*), определяющих объекты и классы для информации, зависящей от создателя и информационной системы.

Самая важная часть WMI, которая отличает ее от обычных реализаций WBEM, – это схема Win32, расширенная схема для информации, специфичной для Win32, построенная на центральной и общей модели. WMI также добавляется к общей структуре WBEM, обеспечивая механизмы доступа к данным CIM, специфичные для Win32.[1] Используя это расширение схемы и набор методов доступа к данным, мы можем выяснить, как управлять процессами из Perl средствами WMI.

Два из этих методов доступа: ODBC (открытый интерфейс взаимодействия с базами данных) и COM/DCOM (модель составных компонентов/распределенная модель составных компонентов) будут более полно

[1] До тех пор пока Microsoft будет стремиться сделать эти механизмы повсеместными, вероятность найти их не в Win32-окружении очень мала. Вот почему я называю их «специфичными для Win32».

рассмотрены в других главах этой книги. В примерах будет использоваться модель COM/DCOM, поскольку ODBC разрешает лишь запрашивать информацию у WMI (хотя и в простой, похожей на присущую базам данных манере). COM/DCOM позволяет и запрашивать информацию, и взаимодействовать с ней, что очень важно для «управляющей» части контроля над процессами.

Приведенные далее примеры программ на Perl не выглядят такими уж трудными, и вас могут удивить слова «очень быстро становится чересчур сложным». Приведенный ниже код выглядит простым, потому что:

- Мы касаемся только поверхности WMI. Мы даже не затрагиваем таких понятий, как «ассоциации» (т. е. связи между объектами и классами объектов).
- Мы выполняем только простые операции управления. Управление процессами в таком контексте состоит из опроса исполняемых процессов и возможности их завершения. Эти операции легко осуществляются в WMI при использовании расширения схемы Win32.
- В наших примерах спрятана вся сложность перевода документации WMI и примеров программ с VBscript/JScript на Perl.
- В примерах скрыта неясность процессов отладки. Когда код на Perl, имеющий отношений к WMI, завершается с ошибками, выдается очень мало информации, которая могла бы помочь найти их причину. Да, вы получите сообщения об ошибках, но в них никогда не будет сказано `ОШИБКА: ПРОБЛЕМА ЗАКЛЮЧАЕТСЯ В СЛЕДУЮЩЕМ...` Скорее всего, вы получите что-нибудь подобное `wbemErrFailed 0x8004100` или вообще пустую структуру данных. Справедливости ради надо сказать, что бо́льшая часть такой неясности возникает благодаря участию Perl в этом процессе. Он действует в качестве интерфейса к целому ряду довольно сложных многоуровневых операций, которые не утруждают себя передачей содержательных сообщений в случае возникновения проблем.

Это звучит довольно мрачно. Поэтому позвольте предложить совет, воспользоваться которым стоит перед тем, как рассматривать сами примеры:

- Изучите любые примеры, использующие модуль `Win32::OLE`, которые сможете найти. Список рассылки *Win32-Users* на ActiveState и его архивы на *http://www.activestate.com* – хороший источник подобной информации. Если сравнить их с эквивалентными примерами на VBscript, то станут понятны необходимые идиомы трансляции. Кроме того, вам может помочь раздел «ADSI (Интерфейсы активных служб каталогов)» главы 6.
- Подружитесь с отладчиком Perl и используйте его для тщательной проверки фрагментов кода в качестве части процесса обучения. Другой способ тестирования на платформе Win32 отрывков кода на Perl – применение программы TurboPerl Вильяма Смита (William

P. Smith), ее можно найти на *http://users.erols.com/turboperl/*, совместно с модулями *dumpvar.pl* или `Data::Dumper`. В ней бывают сбои (я советую чаще сохранять исправления), но обычно она может помочь в создании заготовок кода на Perl. Другие инструменты интегрированной среды разработки также могут обладать подобными возможностями.

- Всегда держите под рукой копию WMI SDK. Его документация и примеры кода на VBscript очень полезны.
- Чаще используйте броузер объектов WMI в WMI SDK. Он поможет вам разобраться со структурой.

Теперь перейдем к Perl. Первоначальная наша задача – определить, какую информацию о процессах в Win32 можно получить и как ее использовать.

Сначала нужно установить соединение с *пространством имен* (*namespace*) WMI. Пространство имен определяется в WMI SDK как «единица для группировки классов и экземпляров для управления их областью действия и видимостью». Нам необходимо соединение с корнем стандартного пространства имен `cimv2`, в котором содержатся все интересующие нас данные.

Кроме того, потребуется установить соединение с соответствующим уровнем привилегий. Программа должна иметь право отлаживать процесс и представлять нас; другими словами, она должна выполняться от имени пользователя, запустившего сценарий. Установленное соединение позволит получить объект `Win32_Process` (как это определяется в схеме Win32).

Существуют как простой, так и сложный способы создать это соединение. В первом примере будут приведены оба способа, так что читатель сможет решить, чего стоит каждый из них. Вот сложный способ с объяснениями.

```
use Win32::OLE('in');

$server = ''; # соединение с локальной машиной

# получаем объект SWbemLocator
$lobj = Win32::OLE->new('WbemScripting.SWbemLocator')
   or die "Невозможно создать объект локатор: ".Win32::OLE->LastError()."\n";

# определяем, что сценарий выполняется с правами пользователя
$lobj->{Security_}->{impersonationlevel} = 3;

# используем это для получения объекта SWbemServices
$sobj = $lobj->ConnectServer($server, 'root\cimv2')
  or die "Невозможно создать объект сервер:
         ".Win32::OLE->LastError()."\n";
```

```
# получаем объект схемы
$procschm = $sobj->Get('Win32_Process');
```

Сложный способ включает в себя:

- Получение объекта локатора, используемого для нахождения соединения с объектом-сервером
- Установку прав, т. е. программа будет выполняться с нашими привилегиями
- Использование этого объекта для получения соединения с `cimv2` – пространством имен WMI
- Применение этого соединения для получения объекта `Win32_Process`

Все это можно сделать за один шаг, если использовать COM *moniker's display name*. В соответствии с WMI SDK, «в модели составных объектов (COM) моникер – это стандартный механизм для инкапсуляции местоположения другого COM-объекта и связи с ним. Текстовое представление моникера называется отображаемым именем». Вот и простой способ в действии:

```
use Win32::OLE('in');

$procschm = Win32::OLE->GetObject(
  'winmgmts:{impersonationLevel=impersonate}!Win32_Process')
    or die "Невозможно создать объект сервера: ".Win32::OLE->LastError()."\n";
```

Теперь, когда у нас есть объект `Win32_Process`, можно с его помощью получить нужные части схемы, представляющие собой процессы в Win32. В их число входят все доступные свойства и методы `Win32_Process`, которые годятся к употреблению. Применяемая программа довольно проста; единственно, что не вполне очевидно, – это использование оператора `in` в `Win32::OLE`. Чтобы объяснить это, нам придется немного отклониться от темы.

Объект `$procschm` имеет два специальных свойства: `Properties_` и `Methods`. В каждом из них хранится специальный дочерний объект, известный как *collection object* в терминологии COM. Объект *collection object* является родительским контейнером для других объектов; в этом случае в них хранятся объекты описания свойств (`Properties_`) и методов (`Methods_`) схемы. Оператор `in` возвращает массив ссылок на каждый дочерний объект контейнера. Располагая таким массивом, можно обойти все его элементы в цикле, возвращая на каждой итерации свойство `Name` каждого дочернего объекта. О других известных применениях `in` можно узнать из раздела «ADSI (Интерфейсы активных служб каталогов)» главы 6. Вот как выглядит сама программа:

```
use Win32::OLE('in');

# соединяемся с пространством имен, даем указание действовать
# с правами пользователя и получаем объект Win32_process,
```

```
# просто используя отображаемое имя
$procschm = Win32::OLE->GetObject(
    'winmgmts:{impersonationLevel=impersonate}!Win32_Process')
    or die "Невозможно создать объект сервера: ".Win32::OLE->LastError()."\n";

print "--- Properties ---\n";
print join("\n",map {$_->{Name}}(in $procschm->{Properties_}));
print "\n--- Methods ---\n";
print join("\n",map {$_->{Name}}(in $procschm->{Methods_}));
```

Вывод (на машине с NT4.0) выглядит примерно так:

```
--- Properties ---
Caption
CreationClassName
CreationDate
CSCreationClassName
CSName
Description
ExecutablePath
ExecutionState
Handle
InstallDate
KernelModeTime
MaximumWorkingSetSize
MinimumWorkingSetSize
Name
OSCreationClassName
OSName
PageFaults
PageFileUsage
PeakPageFileUsage
PeakWorkingSetSize
Priority
ProcessId
QuotaNonPagedPoolUsage
QuotaPagedPoolUsage
QuotaPeakNonPagedPoolUsage
QuotaPeakPagedPoolUsage
Status
TerminationDate
UserModeTime
WindowsVersion
WorkingSetSize
--- Methods ---
Create
Terminate
GetOwner
GetOwnerSid
```

Рассмотрим это подробнее. Чтобы получить список запущенных процессов, нужно запросить все экземпляры объектов `Win32_Process`:

```
use Win32::OLE('in');

# выполняем все первоначальные шаги в одном цикле

$sobj = Win32::OLE->GetObject(
                    'winmgmts:{impersonationLevel=impersonate}')
     or die "Невозможно создать объект сервера: ".Win32::OLE->LastError()."\n";

foreach $process (in $sobj->InstancesOf("Win32_Process")){
  print $process->{Name}." имеет pid #".$process->{ProcessId},"\n";
}
```

Первоначальное отображаемое имя не включает путь к определенному объекту (т. е. мы отбросили `!Win32_Process`). Итак, получен объект связи с сервером. В результате вызова метод `InstancesOf()` возвращает объект-коллекцию (collection object), который содержит все экземпляры конкретного объекта. Наш код обращается к каждому объекту и выводит его свойства `Name` и `ProcessId`. В итоге, у нас есть список всех запущенных процессов.

Чуть менее великодушный подход к обойденным в цикле процессам позволил бы использовать один из методов, указанных в приведенном выше списке:

```
foreach $process (in $sobj->InstancesOf("Win32_Process")){
  $process->Terminate(1);
}
```

В результате, все работающие процессы будут завершены. Я не рекомендую вам запускать эту программу в таком виде; подправьте ее в соответствии с вашими нуждами, сделав более конкретной.

Теперь у вас есть необходимые знания, чтобы начать использовать WMI для управления процессами. В WMI есть Win32-расширения для многих других частей операционной системы, включая реестр и журнал событий.

Вот и все, что мы хотели сказать об управлении процессами в WinNT и 2000. Теперь перейдем к последней операционной системе.

Управление процессами в Unix

Стратегии управления процессами в Unix представляют собой иную ситуацию, включающую несколько вариантов для выбора. К счастью, эти варианты даже отдаленно по своей сложности не напоминают то, что мы видели в NT. В разговоре об управлении процессами в Unix следует иметь в виду три операции:

1. Нумерацию списка запущенных на машине процессов.
2. Изменение их приоритета или группы.
3. Завершение работы процесса.

Для последних двух операций в Perl существуют функции `setpriority()`, `setpgrp()` и `kill()`. В случае с первой у нас есть несколько возможностей. Чтобы вывести список запущенных процессов, можно:

- Вызвать внешнюю программу, например *ps*
- Поломать голову над расшифровкой */dev/kmem*
- Изучить файловую систему */proc*
- Использовать модуль `Proc::ProcessTable`

Обсудим каждый из этих подходов. Нетерпеливым читателям могу сказать прямо сейчас, что я сам предпочитаю модуль `Proc::ProcessTable`, и вы можете пропустить все рассуждения и перейти сразу к рассказу о его использовании. Но рекомендую все же прочитать материал и о других технологиях, т. к. в будущем они могут вам пригодиться.

Вызов внешней программы

Во всех современных вариантах операционной системы Unix есть команда *ps*, применяемая для получения списка запущенных процессов. Однако в каждом конкретном случае она расположена в различных местах, а аргументы командной строки, которые она принимает, тоже не совпадают. Отсюда и проблема с ее применением: она недостаточно переносима.

Еще более неприятная проблема – это сложность анализа вывода (который тоже отличается в различных версиях). Вот как выглядит вывод команды *ps* на машине с SunOS:

```
USER    PID %CPU %MEM   SZ  RSS TT STAT START  TIME COMMAND
dnb     385  0.0  0.0  268    0 p4 IW   Jul  2  0:00 /bin/zsh
dnb   24103  0.0  2.610504 1092 p3 S    Aug 10 35:49 emacs
dnb     389  0.0  2.5 3604 1044 p4 S    Jul  2 60:16 emacs
remy  15396  0.0  0.0  252    0 p9 IW   Jul  7  0:01 -zsh (zsh)
sys     393  0.0  0.0   28    0 ?  IW   Jul  2  0:02 in.identd
dnb   29488  0.0  0.0   68    0 p5 IW   20:15   0:00 screen
dnb   29544  0.0  0.4   24  148 p7 R    20:39   0:00 less
dnb    5707  0.0  0.0  260    0 p6 IW   Jul 24  0:00 -zsh (zsh)
root  28766  0.0  0.0  244    0 ?  IW   13:20   0:00 -:0 (xdm)
```

Обратите внимание на третью строку. Два столбца в ней слились вместе и при анализе вывода разобраться в этом будет непросто. Нет, это возможно, но просто действует на нервы. В некоторых вариантах Unix дела обстоят лучше, но это обстоятельство следует учитывать.

Программа на Perl, применяемая в этом случае, прямолинейна: в ней используются `open()` для запуска *ps*, `while(<FH>){…}` для чтения вывода

и `split()`, `unpack()` или `substr()` для анализа данного вывода. Совет по этому поводу можно найти в книге Тома Кристиансена (Tom Christiansen) и Натана Торкингтона (Nathan Torkington) «Perl: Библиотека программиста», изд-во «Питер», 2000 г. («Perl Cookbook», O'Reilly).

Изучение структуры ядра

Я упомянул эту возможность только для полноты картины. Можно написать программу, которая будет открывать устройство, подобное */dev/kmem*, и обращаться к структурам памяти ядра. Таким образом можно добраться до текущей таблицы процессов и прочитать ее. Учитывая, что сделать это трудно (разобраться вручную в сложной двоичной структуре), а полученный результат не будет обладать абсолютно никакой переносимостью (любое изменение даже версии операционной системы сделает, скорее всего, вашу программу неработоспособной), я настоятельно рекомендую не пользоваться такой возможностью.

Тем, кто все же не прислушается к этому совету, придется вспомнить документацию по функциям `pack()`, `unpack()` и заголовочным файлам для вашего ядра. Откройте файл памяти ядра (часто это */dev/kmem*), затем выполняйте `read()` и `unpack()`. Вам может понадобиться изучить исходники таких программ, как *top* (ищите на *ftp://ftp.groupsys.com/pub/top*), выполняющих эти же задачи, используя большое количество кода на C. В следующем разделе мы рассмотрим слегка улучшенную версию этого метода.

Использование файловой системы /proc

В большинстве современных вариантов Unix существует интересное добавление – файловая система */proc*. Эта загадочная файловая система не имеет ничего общего с хранением данных. Она обеспечивает «файлоподобный» интерфейс к таблице запущенных процессов. Для каждого из них в этой файловой системе существует «каталог», название которого совпадает с идентификатором процесса. В этом каталоге есть целый ряд «файлов», предоставляющих информацию о данном процессе. В один из этих файлов разрешена запись, что и позволяет управлять самим процессом.

Это действительно мудрая идея и это замечательно. Плохо то, что каждый производитель/команда разработчиков, поддержав эту мудрую концепцию, разбежались каждый в своем направлении. В результате файлы, которые можно найти в каталогах */proc*, часто специфичны для различных вариантов операционной системы, отличаясь как по именам, так и по формату. Описание того, какие файлы доступны и что в них хранится, вам придется искать на страницах руководства (обычно в разделах 4, 5 или 8) по *procfs* или *mount_procfs*.

Единственное переносимое использование файловой системы */proc* – это нумерация запущенных процессов. Если нам нужно только пере-

числить идентификаторы процессов и их владельцев, мы можем применять операторы Perl по работе с каталогами и `lstat()`:

```
opendir(PROC,"/proc") or die "Невозможно открыть /proc:$!\n";
while (defined($_= readdir(PROC))){
    next if ($_ eq "." or $_ eq "..");
    next unless /^\d+$/; # отфильтровываем все случайные файлы, названия
                         # которых могут являться идентификаторами процессов
    print "$_\t". getpwuid((lstat "/proc/$_")[4])."\n";
}
closedir(PROC);
```

Для того чтобы получить подробную информацию о процессе, следует открыть нужный двоичный файл из каталогов в */proc* и воспользоваться функцией `unpack()`. Обычно это файл *status* или *psinfo*. На страницах только что упомянутых руководств есть подробная информация о C-структуре, которую можно найти в этом файле, или, по крайней мере, есть ссылка на включаемый (include) файл, в котором эта структура документирована. Поскольку эти форматы зависят от операционной системы (и версии ОС), вы снова столкнетесь с проблемами переносимости программы.

Вероятно, вы уже чувствуете себя растерянным, поскольку все рассмотренные варианты требуют, чтобы в программе были учтены все версии каждой операционной системы, которую нам надо поддерживать. К счастью, в запасе у нас есть еще одна возможность, и она может помочь.

Использование модуля Proc::ProcessTable

Дэниел Дж. Урист (Daniel J. Urist) (с помощью нескольких добровольцев) написал модуль `Proc::ProcessTable`, предоставляющий единый интерфейс к таблице процессов для всех основных вариантов операционной системы Unix. Он скрывает от вас причуды различных реализаций */proc* и *kmem*, позволяя писать более переносимые программы.

Просто загрузите модуль, создайте объект `Proc::ProcessTable::Process` и используйте методы этого объекта:

```
use Proc::ProcessTable;

$tobj = new Proc::ProcessTable;
```

Этот объект использует механизм связанных переменных (tied variable) для представления текущего состояния системы. Для обновления этого объекта не требуется вызывать специальную функцию – он перечитывает таблицу процессов при каждом обращении к нему. Это похоже на хэш `%Process`, знакомый нам по обсуждению модуля `Mac::Processes` ранее в этой главе.

Чтобы получить нужную информацию, следует вызвать метод `table()`:

```
$proctable = $tobj->table();
```

table() возвращает ссылку на массив, элементы которого представляют собой ссылки на объекты процессов. Каждый из этих объектов имеет свой собственный набор методов, возвращающих информацию об этом процессе. Например, вот как можно получить список идентификаторов процессов и их владельцев:

```
use Proc::ProcessTable;

$tobj = new Proc::ProcessTable;
$proctable = $tobj->table();
for (@$proctable){
    print $_->pid."\t". getpwuid($_->uid)."\n";
}
```

Список методов, доступных в вашей операционной системе, можно получить при помощи метода fields() объекта Proc::ProcessTable (т. е. $tobj).

В Proc::ProcessTable также есть три дополнительных метода у каждого объекта процесса: kill(), priority() и pgrp(), которые являются всего лишь интерфейсом к встроенным функциям, упомянутым в начале этого раздела.

Чтобы опять вернуться к общей задаче, посмотрим на применение способов контроля над процессами. Мы начали изучать управление процессами в контексте действий пользователя, поэтому сейчас рассмотрим несколько маленьких сценариев, посвященных этим действиям. В примерах мы будем использовать Proc::ProcessTable в Unix, но сами идеи не зависят от операционной системы.

Первый пример из документации по Proc::ProcessTable:

```
use Proc::ProcessTable;

$t = new Proc::ProcessTable;
foreach $p (@{$t->table}){
  if ($p->pctmem > 95){
    $p->kill(9);
  }
}
```

Эта программа «отстреливает» все процессы, занимающие 95% памяти в тех вариантах операционной системы Unix, где поддерживается метод pctmem() (а он поддерживается в большинстве случаев). В таком виде пример, вероятно, слишком «безжалостен», чтобы использовать его в реальной жизни. Было бы благоразумно добавить перед командой kill() что-то подобное:

```
print "собираемся убрать ".$p->pid."\t". getpwuid($p->uid)."\n";
print "выполнять? (yes/no) ";
```

```
chomp($ans = <>);
next unless ($ans eq "yes");
```

Здесь может возникнуть состояние перехвата: не исключено, что во время задержки, вызванной вопросом к пользователю, состояние системы изменится. Учитывая, что мы в данном случае работаем только с «большими» процессами, которые вряд ли меняют свое состояние в течение короткого времени, такой вариант, скорее всего, пройдет нормально. Если вы хотите подойти к этому вопросу более педантично, вам, наверное, стоит получить сначала список процессов, которые вы хотите завершить, спросить пользователя, а затем проверить еще раз состояние таблицы процессов и только потом их завершать.

Бывают случаи, когда завершение процесса – это слишком легкая расплата. Иногда важно засечь, что процесс действительно работает, чтобы предпринять необходимые меры (скажем, поставить пользователя на место). Например, политика нашего сайта запрещает применять IRC-*роботы*. Роботы – это процессы-демоны, которые соединяются с IRC-серверами и выполняют автоматические действия. И хотя роботы могут использоваться в благих целях, в настоящее время они, в основном, играют асоциальную роль в IRC. Кроме того, мы обращали внимание на взлом системы безопасности из-за того, что первое (и часто единственное), что делал взломщик, – это запускал IRC-робота. Короче говоря, нам важно заметить присутствие таких процессов, а не завершать их работу.

Чаще других сейчас используется робот под названием *eggdrop*. Выяснить, запущен ли в системе процесс с таким именем, можно при помощи следующей программы:

```
use Proc::ProcessTable;

open(LOG,">>$logfile") or die "Невозможно открыть журнал для дозаписи:$!\n";

$t = new Proc::ProcessTable;
foreach $p (@{$t->table}){
  if ($p->fname() =~ /eggdrop/i){
    print LOG time."\t".getpwuid($p->uid).
              "\t".$p->fname()."\n";
  }
}
close(LOG);
```

Тот, кто подумает: «Эта программа не так уж и хороша! Все, что нужно сделать, чтобы ускользнуть от этой проверки, так это всего лишь переименовать исполняемый файл», будет абсолютно прав. Мы попытаемся написать менее простодушный код, ищущий роботов, в самом последнем разделе этой главы.

А пока рассмотрим еще один пример, в котором Perl помогает управлять процессами пользователей. До сих пор все наши примеры каса-

лись отрицательных явлений. Рассмотренные программы имели дело со злонамеренными или жадными к ресурсам процессами. Теперь посмотрим на что-нибудь более жизнерадостное.

Существуют ситуации, когда системному администратору необходимо узнать, какие программы применяются пользователями в системе. Иногда это необходимо сделать для программного обеспечения, лицензия которого запрещает его одновременное использование сверхнормативным числом потребителей. В таких случаях обычно применяется специальный механизм. Иногда подобные сведения необходимы, чтобы иметь возможность перейти на другую систему. Если вы переносите пользователей с одной системы на другую, вам необходимо убедиться, что все программы, работающие на старом месте, будут доступны и на новом.

Один подход к решению этой задачи – заменить каждую доступную пользователям исполняемую программу, не входящую в состав операционной системы, на оболочку, которая сначала запишет, какая программа была вызвана, а затем и запустит ее. Это сложно реализовать, если в системе доступно множество программ. Кроме того, есть и побочный эффект – запуск каждой программы требует большего времени.

Если точность не важна и достаточно знать только приблизительную оценку набора работающих программ, можно применить `Proc::ProcessTable`. Ниже приведена программа, которая активизируется каждые пять минут и проверяет состояние текущих процессов. Она просто ведет учет всех найденных имен процессов, причем те процессы, которые встречались в предыдущий раз, она во второй раз не учитывает. Ежечасно программа записывает результаты и начинает подсчет заново. Пятиминутное ожидание объясняется тем, что обход таблицы процессов является ресурсоемкой операцией (обычно), а мы хотим, чтобы эта программа как можно меньше загружала систему:

```
use Proc::ProcessTable;

$interval    = 600; # 5 минут перерыва
$partofhour  =   0; # отмечаем позицию часа, в которой мы находимся

$tobj = new Proc::ProcessTable; # создаем новый объект

# вечный цикл, сбор данных каждые $interval секунд
# и сброс этих данных один раз в час
while(1){
    &collectstats;
    &dumpandreset if ($partofhour >= 3600);
    sleep($interval);
}

# сбор статистики по процессу
sub collectstats {
```

```
    my($process);
    foreach $process (@{$tobj->table}){

        # мы должны игнорировать себя
        next if ($process->pid() == $$);

        # сохраняем информацию о процессе для следующего запуска
        push(@last,$process->pid(),$process->fname());

        # игнорируем процесс, если мы его уже видели
        next if ($last{$process->pid()} eq $process->fname());

        # если не видели, запоминаем его
        $collection{$process->fname()}++;
    }
    # устанавливаем хэш %last, используя текущую таблицу
    %last = @last;
    $partofhour += $interval;
}

# выводим результаты и сбрасываем значения счетчиков
sub dumpandreset{
    print scalar localtime(time).("-"x50)."\n";
    for (sort reverse_value_sort keys %collection){
        write;
    }

    undef %collection;
    $partofhour = 0;
}

# (обратная) сортировка по значениям хэша %collection и по
# именам ключей
sub reverse_value_sort{
    return $collection{$b} <=> $collection{$a} || $a cmp $b;
}

format STDOUT =
@<<<<<<<<<<<<<  @>>>>
$_,             $collection{$_}
.

format STDOUT_TOP =
Name            Count
--------------  -----
.
```

Существует множество способов улучшить эту программу. Она могла бы отслеживать процессы для каждого пользователя (т. е. записывать один экземпляр вызова программы для каждого пользователя), соби-

рать ежедневную статистику, выводить информацию в виде диаграммы и т. д. Все зависит только от того, где вы хотите ее применять.

Отслеживание операций с файлами и сетью

В последнем разделе этой главы следует объединить две сферы действий пользователей. Процессы, управление которых так долго обсуждалось, заняты не только поглощением процессорного времени и памяти. Помимо этого они выполняют операции с файловыми системами и от имени пользователей загружают сеть. Администрирование пользователей требует не забывать и об этом.

Мы сосредоточимся на довольно узком круге вопросов и будем обращать внимание только на операции с файлами и сетью, которые выполняют *другие* пользователи. Кроме того, мы будем обращать внимание только на те операции, владельца которых можно отследить (другими словами, на конкретные процессы, запущенные конкретными пользователями). Что ж, учитывая это, двинемся дальше.

Отслеживание операций в Windows NT/2000

Попытка найти файлы, открытые другими пользователями, вернее всего сработает, если применять программу, работающую в командной строке, – *nthandle* Марка Руссиновича (Mark Russinovich), ее можно найти на *http://www.sysinternals.com*. Она позволяет показать все открытые дескрипторы на определенной системе. Вот как выглядит ее вывод:

```
System pid: 2
     10: File    C:\WINNT\SYSTEM32\CONFIG\SECURITY
     84: File    C:\WINNT\SYSTEM32\CONFIG\SAM.LOG
     cc: File    C:\WINNT\SYSTEM32\CONFIG\SYSTEM
     d0: File    C:\WINNT\SYSTEM32\CONFIG\SECURITY.LOG
     d4: File    C:\WINNT\SYSTEM32\CONFIG\DEFAULT
     e8: File    C:\WINNT\SYSTEM32\CONFIG\SYSTEM.ALT
     fc: File    C:\WINNT\SYSTEM32\CONFIG\SOFTWARE.LOG
    118: File    C:\WINNT\SYSTEM32\CONFIG\SAM
    128: File    C:\pagefile.sys
    134: File    C:\WINNT\SYSTEM32\CONFIG\DEFAULT.LOG
    154: File    C:\WINNT\SYSTEM32\CONFIG\SOFTWARE
    1b0: File    \Device\NamedPipe\
    294: File    C:\WINNT\PROFILES\Administrator\ntuser.dat.LOG
    2a4: File    C:\WINNT\PROFILES\Administrator\NTUSER.DAT
-------------------------------------------------------------------------------
SMSS.EXE pid: 27 (NT AUTHORITY:SYSTEM)
      4: Section       C:\WINNT\SYSTEM32\SMSS.EXE
      c: File          C:\WINNT
     28: File          C:\WINNT\SYSTEM32
```

Можно также запросить информацию по конкретным файлам и каталогам:

```
> nthandle c:\temp
Handle V1.11
Copyright (C) 1997 Mark Russinovich
http://www.sysinternals.com

WINWORD.EXE        pid: 652    C:\TEMP\~DFF2B3.tmp
WINWORD.EXE        pid: 652    C:\TEMP\~DFA773.tmp
WINWORD.EXE        pid: 652    C:\TEMP\~DF913E.tmp
```

Программа *nthandle* позволяет получить эту информацию по конкретному процессу при помощи ключа *–p*.

Использовать ее из Perl очень просто, поэтому не будем приводить примеры. Вместо этого посмотрим на подобную, но более интересную операцию – аудит.

NT/2000 позволяет эффективно отслеживать изменения в файлах, каталогах или иерархии каталогов. Вы могли бы учитывать постоянное повторение операции `stat()` над нужным объектом, но это потребовало бы слишком больших затрат процессорного времени. В NT/2000 отслеживание изменений можно поручить операционной системе.

Относительно безболезненно эту работу выполняют два модуля: `Win32::ChangeNotify` Кристофера Мадсена (Christopher J. Madsen) и `Win32::AdvNotify` Амина Мюлей Рамдана (Amine Moulay Ramdane). В примерах этого раздела будет использоваться второй, т. к. он более гибкий.

Работа с модулем `Win32::AdvNotify` – это многошаговый процесс. Для начала следует загрузить модуль и создать новый объект `AdvNotify`:

```
# также импортируем две постоянные, которые скоро будем
# использовать
use Win32::AdvNotify qw(All %ActionName);
use Data::Dumper;

$aobj = new Win32::AdvNotify()
     or die "Невозможно создать новый объект\n";
```

На следующем шаге нужно создать следящий поток (monitoring thread) для интересующего нас каталога. `Win32::AdvNotify` позволяет следить сразу за набором каталогов, для этого необходимо лишь создать несколько потоков. Мы же будем следить только за одним каталогом:

```
$thread = $aobj->StartThread(Directory => 'C:\temp',
                             Filter => All,
                             WatchSubtree => 0)
   or die "Невозможно начать поток\n";
```

Первый параметр этого метода говорит сам за себя; рассмотрим остальные.

Установив `Filter` в одно из приведенных значений (табл. 4.1) или в их комбинацию (`SETTING1 | SETTING2 | SETTING3...`), можно следить за различными типами изменений.

Таблица 4.1. Параметры Filter в Win32::AdvNotify

Параметр	Отмечает
FILE_NAME	Создание, удаление, переименование файла(ов)
DIR_NAME	Создание или удаление каталога(ов)
ATTRIBUTES	Изменение атрибутов любого каталога
SIZE	Изменение размера любого файла
LAST_WRITE	Изменение даты модификации файла(ов)
CREATION	Изменение даты создания файла(ов)
SECURITY	Изменение информации безопасности (ACL и пр.) файла(ов)

Значение `All` из приведенного примера – это всего лишь постоянная, объединяющая все варианты выбора. Если не указать параметр `Filter` при вызове метода, то по умолчанию будет использоваться `All`. Параметр `WatchSubtree` определяет, необходимо ли следить только за указанным каталогом или за каталогом и всеми его подкаталогами.

`StartThread()` создает поток, но проверка начинается только после того, как поступает распоряжение об этом:

```
$thread->EnableWatch() or die "Невозможно начать наблюдение\n";
```

Существует также функция `DisableWatch()`, которую необходимо использовать в программе для завершения проверки.

Мы следим за нужным объектом, но как узнать, изменилось ли что-нибудь? Надо придумать что-то, что позволило бы потоку сообщить нам об изменениях, за которыми мы наблюдаем. Здесь тот же подход, что и в главе 9 «Журналы» при обсуждении сетевых сокетов. Обычно следует вызывать функции, которые заблокированы до тех пор, пока ничего не происходит:

```
while($thread->Wait(INFINITE)){
    print "Что-то изменилось!\n";
    last if ($changes++ == 5);
}
```

Этот цикл `while()` вызовет метод `Wait()` для нашего потока. До тех пор пока потоку нечего сообщить, вызов будет заблокирован. Обычно `Wait()` принимает в качестве параметра число миллисекунд, равное времени ожидания. Мы же передаем специальное значение, которое соответствует «бесконечному ожиданию». Когда `Wait()` возвращает значение,

следует вывести сообщение и ждать дальше, если только уже не были замечены пять других изменений. Теперь можно закончить:

```
$thread->Terminate();
undef $aobj;
```

Наша программа пока еще не очень полезна. Нам известно, что *что-то* изменилось, но мы не знаем ни что изменилось, ни как это произошло. Чтобы исправить эту ситуацию, изменим тело цикла `while()` и добавим определение формата:

```
while($thread->Wait(INFINITE)){
    while ($thread->Read(\@status)){
        foreach $event (@status){
            $filename = $event->{FileName};
            $time     = $event->{DateTime};
            $action   = $ActionName{$event->{Action}};
            write;
        }
    }
}

format STDOUT =
@<<<<<<<<<<<<<<<<<<< @<<<<<<<<<<<<<<<<<<< @<<<<<<<<<<<<<<<<<<<
$filename,$time,$action
.

format STDOUT_TOP =
File Name            Date                 Action
-------------------  -------------------- --------------------
.
```

Основное изменение – это добавление метода `Read()`. Он получает информацию об изменении и заполняет элементы списка `@status` ссылками на хэш. Каждая ссылка указывает на анонимный хэш, который выглядит примерно так:

```
{'FileName' => '~GLF2425.TMP',
 'DateTime' => '11/08/1999 06:23:25p',
 'Directory' => 'C:\temp',
 'Action' => 3 }
```

Для каждого изменения могут произойти несколько событий, отсюда и необходимость вызывать `Read()` в цикле `while()`. Если соответствующим образом разыменовать содержимое этих хэшей и применить к ним форматирование, то получатся примерно такие данные:

```
File Name            Date                 Action
-------------------  -------------------- --------------------
~DF40DE.tmp          11/08/1999 07:29:56p FILE_ACTION_REMOVED
~DF6E5C.tmp          11/08/1999 07:29:56p FILE_ACTION_ADDED
```

```
~DF6E66.tmp          11/08/1999 07:29:56p FILE_ACTION_ADDED
~DF6E5C.tmp          11/08/1999 07:29:56p FILE_ACTION_REMOVED
```

К сожалению, отслеживание операций с сетью в NT/2000 впечатляет намного меньше. В идеале, как администратор вы хотели бы знать, какой процесс (а, следовательно, какой пользователь) открыл сетевой порт. Печально, но я не знаю ни одного модуля и ни одного инструмента, которые могли бы предоставить такую информацию. Существует один коммерческий инструмент, работающий в командной строке, под названием *TCPVstat*, который может показать связь процессов с использованием сети. *TCPVstat* можно найти в пакете TCPView Professional Edition, который доступен на *http://www.winternals.com*.

Если использовать только некоммерческие инструменты, то придется иметь дело лишь со списком сетевых портов, открытых в настоящее время. Для этого следует применять другой модуль Рамдана – `Win32::IpHelp`. Вот как выглядит код, печатающий нужную информацию:

```
use Win32::IpHelp;

# замечание: в данном случае регистр "IpHelp" имеет значение[1]
my $iobj = new Win32::IpHelp;

# заполняем список хэшем хэшей
$iobj->GetTcpTable(\@table,1);

foreach $entry (@table){
    print $entry->{LocalIP}->{Value} . ":" .
          $entry->{LocalPort}->{Value}. " -> ";
    print $entry->{RemoteIP}->{Value} . ":" .
          $entry->{RemotePort}->{Value}."\n";
}
```

Посмотрим, как можно сделать то же самое в Unix.

Отслеживание операций в Unix

Для отслеживания операций с файлами и сетью в Unix можно использовать один и тот же подход. Это один из тех редких случаев, когда вызов внешней программы намного предпочтительней. Вик Абель (Vic Abell) преподнес чудесный подарок системным администраторам, написав программу *lsof* (LiSt Open Files), которую можно найти на *ftp://vic.cc.purdue.edu/pub/tools/unix/lsof*. *lsof* позволяет отобразить подробную информацию об открытых в настоящий момент файлах и сетевых соединениях на Unix-машине. По-настоящему удивительной эту

[1] Следует отметить, что регистр в Perl всегда имеет значение. – *Примеч. науч. ред.*

программу делает ее переносимость. Последняя версия программы (на момент написания этой книги) работает по крайней мере на 18 видах Unix и поддерживает различные версии этих операционных систем.

Вот как выглядит вывод *lsof* для одного из запущенного мной процесса. *lsof* выводит очень длинные строки, поэтому, чтобы сделать информацию более читаемой, после каждой строки вывода я добавил пустую строку:

```
COMMAND    PID USER   FD   TYPE     DEVICE  SIZE/OFF       NODE NAME
netscape 21065  dnb  cwd   VDIR   172,2891      8192      12129 /home/dnb

netscape 21065  dnb  txt   VREG   172,1246  14382364     656749 /net/
arch-solaris (fileserver:/vol/systems/arch-solaris)

netscape 21065  dnb  txt   VREG       32,6     54656      35172 /usr (/dev/
dsk/c0t0d0s6)

netscape 21065  dnb  txt   VREG       32,6    146740       6321 /usr/lib/
libelf.so.1

netscape 21065  dnb  txt   VREG       32,6     69292     102611 /usr (/dev/
dsk/c0t0d0s6)

netscape 21065  dnb  txt   VREG       32,6     21376      79751 /usr/lib/
locale/en_US/en_US.so.1

netscape 21065  dnb  txt   VREG       32,6     19304       5804 /usr/lib/
libmp.so.2

netscape 21065  dnb  txt   VREG       32,6     98284      22860 /usr/openwin/
lib/libICE.so.6

netscape 21065  dnb  txt   VREG       32,6     46576      22891 /usr/openwin/
lib/libSM.so.6

netscape 21065  dnb  txt   VREG       32,6   1014020       5810 /usr/lib/
libc.so.1

netscape 21065  dnb  txt   VREG       32,6    105788       5849 /usr/lib/
libm.so.1

netscape 21065  dnb  txt   VREG       32,6    721924       5806 /usr/lib/
libnsl.so.1

netscape 21065  dnb  txt   VREG       32,6    166196       5774 /usr/lib/
ld.so.1

netscape 21065  dnb    0u  VCHR       24,3      0t73       5863 /devices/
pseudo/pts@0:3-> ttcompat ->ldterm->ptem->pts
```

```
netscape 21065 dnb    3u  VCHR      13,12       0t0       5821 /devices/
pseudo/mm@0:zero

netscape 21065 dnb    7u  FIFO 0x6034d264       0t1      47151 PIPE->
0x6034d1e0

netscape 21065 dnb    8u  inet 0x6084cb68 0xfb210ec        TCP host.ccs.neu.
edu:46575->host2.ccs.neu. edu:6000 (ESTABLISHED)

netscape 21065 dnb   29u  inet 0x60642848  0t215868        TCP host.ccs.neu.
edu:46758-> www.mind-bright.se:80 (CLOSE_ WAIT)
```

Из этого примера можно понять, насколько мощна эта команда. Мы можем увидеть текущий рабочий каталог (`VDIR`), обычные файлы (`VREG`), символьные устройства (`VCHR`), каналы (`FIFO`) и сетевые соединения (`inet`), открытые этим процессом.

Самый простой способ применить программу *lsof* из Perl – вызвать ее в специальном режиме «field» (*–F*). В этом режиме вывод программы делится на специальным образом отмеченные и разделенные поля, вместо использования колонок в стиле *ps*. Это позволяет надежно проанализировать и распознать вывод.

У этого способа вывода результатов есть одна особенность. Вывод организован в виде «наборов процессов» (process sets) и «наборов файлов» (file sets), как их называет автор. Набор процессов – это набор полей, относящихся к одному процессу; набор файлов – это подобный же набор для файла. Все приобретет больший смысл, если включить режим разбивки на поля с параметром 0. В этом случае поля будут разделены символом `NUL` (ASCII 0), а наборы – символом `NL` (ASCII 12). Вот как будет выглядеть предыдущий вариант вывода команды, если использовать режим разбивки на поля (`NUL` представлен в виде символов `^@`):

```
p21065^@cnetscape^@u6700^@Ldnb^@

fcwd^@a ^@l ^@tVDIR^@D0x2b00b4b^@s8192^@i12129^@n/home/dnb^@

ftxt^@a ^@l ^@tVREG^@D0x2b004de^@s14382364^@i656749^@n/net/arch-solaris
(fileserver:/vol/systems/arch-solaris)^@

ftxt^@a ^@l ^@tVREG^@D0x800006^@s54656^@i35172^@n/usr (/dev/dsk/c0t0d0s6)^@

ftxt^@a ^@l ^@tVREG^@D0x800006^@s146740^@i6321^@n/usr/lib/libelf.so.1^@

ftxt^@a ^@l ^@tVREG^@D0x800006^@s40184^@i6089^@n/usr (/dev/dsk/c0t0d0s6)^@

ftxt^@a ^@l ^@tVREG^@D0x800006^@s69292^@i102611^@n/usr (/dev/dsk/c0t0d0s6)^@

ftxt^@a ^@l ^@tVREG^@D0x800006^@s21376^@i79751^@n/usr/lib/locale/en_US/
en_US.so.1^@

ftxt^@a ^@l ^@tVREG^@D0x800006^@s19304^@i5804^@n/usr/lib/libmp.so.2^@

ftxt^@a ^@l ^@tVREG^@D0x800006^@s98284^@i22860^@n/usr/openwin/lib/
libICE.so.6^@
```

```
ftxt^@a ^@l ^@tVREG^@D0x800006^@s46576^@i22891^@n/usr/openwin/lib/
libSM.so.6^@

ftxt^@a ^@l ^@tVREG^@D0x800006^@s1014020^@i5810^@n/usr/lib/libc.so.1^@

ftxt^@a ^@l ^@tVREG^@D0x800006^@s105788^@i5849^@n/usr/lib/libm.so.1^@

ftxt^@a ^@l ^@tVREG^@D0x800006^@s721924^@i5806^@n/usr/lib/libnsl.so.1^@

ftxt^@a ^@l ^@tVREG^@D0x800006^@s166196^@i5774^@n/usr/lib/ld.so.1^@
f0^@au^@l ^@tVCHR^@D0x600003^@o73^@i5863^@n/devices/pseudo/
pts@0:3->ttcompat->ldterm->ptem->pts^@

f3^@au^@l ^@tVCHR^@D0x34000c^@o0^@i5821^@n/devices/pseudo/mm@0:zero^@

f7^@au^@l ^@tFIFO^@d0x6034d264^@o1^@i47151^@nPIPE->0x6034d1e0^@

f8^@au^@l ^@tinet^@d0x6084cb68^@o270380692^@PTCP^@nhost.ccs.neu.edu:46575->
host2.ccs.neu.edu: 6000^@TST=ESTABLISHED^@

f29^@au^@l ^@tinet^@d0x60642848^@o215868^@PTCP^@nhost.ccs.neu.edu:46758->
www.mindbright.se: 80^@TST=CLOSE_WAIT^@
```

Давайте разберемся с этими данными. Первая строка – это набор процессов (это можно понять по первому символу `p`):

```
p21065^@cnetscape^@u6700^@Ldnb^@
```

Каждое поле начинается с буквы, определяющей его содержимое (`p` – идентификатор процесса (`pid`), `c` – команда, `u` – идентификатор пользователя (`uid`) и `L` – имя пользователя (`login`)), и заканчивается символом разделителя. Все поля в строке создают набор процесса. Все последующие строки вплоть до очередного набора процесса описывают открытые файлы/сетевые соединения процесса, описываемого своим набором.

Давайте используем этот режим. Если необходимо вывести список всех открытых файлов и процессов, обращающихся к ним, можно применить такую программу:

```
use Text::Wrap;

$lsofexec = "/usr/local/bin/lsof"; # путь к lsof

# режим (F)ield, разделитель NUL (0), показывать (L)ogin,
# тип файла (t)ype и имя файла (n)ame
$lsofflag = "-FL0tn";
open(LSOF,"$lsofexec $lsofflag|") or
die "Невозможно запустить $lsofexec:$!\n";

while(<LSOF>){
    # работаем с набором процесса
    if (substr($_,0,1) eq "p"){
        ($pid,$login) = split(/\0/);
        $pid = substr($pid,1,length($pid));
    }
```

```
    # работаем с набором файла.
    # Замечание: мы интересуемся только обычными файлами
    if (substr($_,0,5) eq "tVREG"){
        ($type,$pathname) = split(/\0/);

        # процесс может дважды открыть один и тот же файл,
        # поэтому мы должны убедиться, что запишем его
        # только один раз
        next if ($seen{$pathname} eq $pid);
        $seen{$pathname} = $pid;

        $pathname = substr($pathname,1,length($pathname));
        push(@{$paths{$pathname}},$pid);
    }
}

close(LSOF);

for (sort keys %paths){
    print "$_:\n";
    print wrap("\t","\t",join(" ",@{$paths{$_}})),"\n";
}
```

В этом случае *lsof* будет показывать только некоторые из полей. Можно обойти в цикле весь вывод, собирая имена файлов и идентификаторы процессов в хэш списков. Когда будут обработаны все выведенные данные, следует ввести имена файлов в виде отформатированного списка процессов (спасибо Дэвиду Шарноффу (David Muir Sharnoff) за модуль `Text::Wrap`):

```
/usr (/dev/dsk/c0t0d0s6):
        115 117 128 145 150 152 167 171 184 191 200 222 232 238
        247 251 276 285 286 292 293 296 297 298 4244 4709 4991
        4993 14697 20946 21065 24530 25080 27266 27603
/usr/bin/tcsh:
        4246 4249 5159 14699 20949
/usr/bin/zsh:
        24532 25082 27292 27564
/usr/dt/lib/libXm.so.3:
        21065 21080
/usr/lib/ld.so.1:
        115 117 128 145 150 152 167 171 184 191 200 222 232 238
        247 251 267 276 285 286 292 293 296 297 298 4244 4246
        4249 4709 4991 4993 5159 14697 14699 20946 20949 21065
        21080 24530 24532 25080 25082 25947 27266 27273 27291
        27292 27306 27307 27308 27563 27564 27603
/usr/lib/libc.so.1:
        267 4244 4246 4249 4991 4993 5159 14697 14699 20949
        21065 21080 24530 24532 25080 25082 25947 27273 27291
        27292 27306 27307 27308 27563 27564 27603
...
```

Чтобы показать последний код, относящийся к отслеживанию операций с файлами и сетью в Unix, вернемся к поиску запущенных IRC-роботов из приведенного ранее примера. Существует более надежный способ найти такие процессы-демоны, чем изучение таблицы процессов. Пользователь может скрыть имя робота, переименовав исполняемый файл, но для того чтобы спрятать сетевое соединение, ему придется очень хорошо потрудиться. В большинстве случаев это соединение с сервером на портах 6660-7000. Программа *lsof* позволяет без труда отыскивать такие процессы:

```
$lsofexec = "/usr/local/bin/lsof";
$lsofflag = "-FL0c -iTCP:6660-7000";

# это срез хэша, используемый для предварительной загрузки
# таблицы хэшей, существование этих ключей мы будем проверять
# позже. Обычно это записывается так:
#  %approvedclients = ("ircII" => undef, "xirc" => undef, ...);
# (но такой вариант - отличная идея, воплощенная Марком-
# Джейсоном Доминусом(Mark-Jason Dominus))
@approvedclients{"ircII","xirc","pirc"} = ();

open(LSOF,"$lsofexec $lsofflag|") or
  die "Невозможно запустить $lsofexec:$!\n";

while(<LSOF>){
    ($pid,$command,$login) = /p(\d+)\000
                              c(.+)\000
                              L(\w+)\000/x;
  warn "$login использует неразрешенный клиент
              с именем $command (pid $pid)!\n"
      unless (exists $approvedclients{$command});
}

close(LSOF);
```

Это самая простая проверка из всех возможных. Она позволяет отловить тех, кто догадается переименовать *eggdrop* в *pine* или *–tcsh*, и тем более тех, кто даже не попытается спрятать своего робота. Тем не менее, она подвержена тому же недостатку, что и предыдущий вариант тестирования. Если пользователь достаточно умен, он может переименовать робота во что-то из списка «разрешенных клиентов». Чтобы продолжить игру в кошки-мышки, можно предпринять еще как минимум два шага:

- Запустить *lsof* для проверки того, что файл, открытый для данной исполняемой программы, известен как тот, который мог быть открыт этой программой, а не произвольный файл из файловой системы.

- Применить методы управления процессами для проверки того, что пользователь запускает программу из существующего интерпретатора. Если это единственный процесс, запущенный пользователем (т. е. если пользователь оставил его, а сам завершил работу), он, вероятно, является демоном, а значит и роботом.

Игра в кошки-мышки привела нас к точке, позволяющей завершить эту главу. В главе 3 мы говорили, что пользователи совершенно непредсказуемы. Они делают такие вещи, которые системные администраторы не могут даже предвидеть. Известно старое изречение: «Защиты от дураков не существует, потому что дураки изобретательны». С этим фактом придется считаться при программировании на Perl для администрирования пользователей. В итоге вы будете писать более надежные программы. Когда одна из ваших программ начнет «ругаться» из-за того, что пользователь сделал что-то неожиданное, вы сможете спокойно сидеть и наслаждаться человеческой изобретательностью.

Информация о модулях из этой главы

Модуль	Идентификатор на CPAN	Версия
`Mac::Processes` (входит в состав MacPerl; измененную версию можно найти в пакете Mac-Glue)	CNANDOR	1.01
`Win32::API`		0.011
`Win32::ISync` (можно найти на *http://www.generation.net/~aminer/Perl/*)		1.11
`Win32::IProc` (можно найти на *http://www.generation.net/~aminer/Perl/*)		1.32
`Win32::Setupsup` (можно найти на *ftp://ftp.roth.net/pub/NTPerl/Others/SetupSup/* или на *http://Jenda.Krynicky.cz*)		980320
`Win32::Lanman` (можно найти на *ftp://ftp.roth.net/pub/ntperl/Others/Lanman/*)		1.05
`Win32::OLE` (входит в состав ActiveState Perl)	JDB	1.11
`Proc::ProcessTable`	DURIST	0.26
`Win32::AdvNotify` (можно найти на *http://www.generation.net/~aminer/Perl/*)		1.01
`Data::Dumper`	GSAR	2.101
`Win32::IpHelp` (можно найти на *http://www.generation.net/~aminer/Perl/*)		1.02
`Text::Wrap` (входит в состав Perl)	MUIR	98.112902

Установка Win32::IProc

Получение и установка модуля `Win32::IProc` происходят несколько сложнее, чем бывает с другими модулями. Сам модуль вместе с остальными модулями Рамдэна можно найти на *http://www.generation.net/~aminer/Perl/*. Чтобы использовать `Win32::IProc`, вам также понадобится загрузить еще два модуля: `Win32::ISync` Рамдэна и `Win32::API` Алдо Калпини (Aldo Calpini). Первый можно найти на сайте Рамдэна, второй в репозитории модулей ActiveState или на *http://dada.perl.it/*.

Некоторые из модулей Рамдэна устанавливаются вручную без помощи команды *ppm* и требуют небольших изменений исходного кода. Вот полный рецепт для установки. Я считаю, что вы распаковали дистрибутивы и собираетесь устанавливать их в Perl, скомпилированный ActiveState и установленный в каталоге *C:\Perl*:

1. *ppm install Win32-API*
2. *md c:\Perl\site\lib\auto\Win32\Sync* и *C:\Perl\site\lib\auto\Win32\Iproc*
3. Скопируйте *timer.dll* и *sync.dll* в *c:\Perl\site\lib\auto\Win32\Sync*
4. Скопируйте *iprocnt.dll*, *psapi.dll* и *iproc.dll* в *C:\Perl\site\lib\auto\Win32\Iproc*
5. Скопируйте *iproc.pm*, *iipc.pm* и *isync.pm* в *C:\Perl\site\lib\Win32*
6. Измените строки `DLLPath` в *iproc.pm* на следующие:

```
my($DLLPath) ="C:\\Perl\\site\\lib\\auto\\Win32\\Iproc\\IProc.dll";
my($DLLPath1)="C:\\Perl\\site\\lib\\auto\\Win32\\Iproc\\IprocNT.dll";
my($DLLPath2)="C:\\Perl\\site\\lib\\auto\\Win32\\Sync\\Sync.dll";
```

7. Измените строку `DLLPath` в *iipc.pm* на:

```
my($DLLPath)="C:\\Perl\\site\\lib\\auto\\Win32\\Sync\\sync.dll";
```

8. Измените строки `DLLPath` в *isync.pm* на:

```
my($DLLPath) ="C:\\Perl\\site\\lib\\auto\\Win32\\Sync\\sync.dll";
my($DLLPath1)="C:\\Perl\\site\\lib\\auto\\Win32\\Sync\\timer.dll";
```

Установка Win32::Setupsup

Если вы хотите установить модуль `Win32::Setupsup` вручную и/или изучить его исходный код, вы можете найти ZIP-архив модуля на *ftp://ftp.roth.net/pub/NTPerl/Others/SetupSup/*. Если же вы предпочитаете установить его простым способом в существующий ActiveState, то можете соединиться с архивом модулей Йенды Крыницки (Jenda Krynicky) и установить его, используя обычный способ *ppm*. Инструкции о том, как это сделать, можно найти на сайте *http://Jenda.Krynicky.cz*.

Сложность в том, что документация в формате pod неверно форматируется, если вызывать ее при помощи *perldoc* или устанавливать в HTML. Документация в конце *setupsup.pm* (вероятнее всего, вы найдете ее в *<ваш каталог Perl >\site\lib\Win32*) гораздо более верная. Если вы попытаетесь узнать, как использовать этот модуль, я советую открыть сам файл в обычном текстовом редакторе и просмотреть те части, которые являются документацией.

Рекомендуемая дополнительная литература

http://pudget.net/macperl – домашняя страница модулей Криса Нандора (Chris Nandor). Нандор один из самых активных разработчиков модулей MacPerl (и соавтор книги, ссылка на которую приведена ниже).

http://www.activestate.com/support/mailing_lists.htm. Здесь находятся списки рассылки *Perl-Win32-Admin и Perl-Win32-Users*. Оба списка и их архивы – просто бесценные ресурсы для программистов Win32.

http://www.microsoft.com/management – домашняя страница всех технологий управления Microsoft, включая WMI.

http://www.sysinternals.com – домашняя страница программы *nthandle* (на этом сайте она называется просто *Handle*) и многих других полезных утилит для NT/2000. На родственном сайте *http://www.winternals.com* продаются отличные коммерческие утилиты.

«MacPerl:Power and Ease», Vicki Brown, Chris Nandor (Prime Time Freeware, 1998) – лучшая книга о модулях для MacPerl. Стоит также обратить внимание на веб-сайт издателя *http://www.macperl.com*.

http://www.dmtf.org – домашняя страница Distributed Management Task Force и просто хороший источник информации по WBEM.

http://www.mspress.com – издатели Microsoft NT Resource Kit. Можно зарегистрироваться и получить доступ к последним утилитам из RK.

- *Файлы узлов*
- *NIS, NIS+ и WINS*
- *Служба доменных имен (DNS)*
- *Информация о модулях из этой главы*
- *Рекомендуемая дополнительная литература*

5

Службы имен TCP/IP

В настоящее время большая часть «разговоров» между компьютерами происходит по *протоколу управления передачей* (*Transmission Control Protocol*), который, в свою очередь, выполняется на более низком уровне, называемом *межсетевым протоколом* (*Internet Protocol*)[1]. Эти два протокола обычно объединяются в один акроним TCP/IP. Каждой машине в TCP/IP-сети должен быть присвоен хотя бы один уникальный численный идентификатор, называемый IP-адресом. IP-адреса обычно записываются в формате *NNN.NNN.N.N*, например, 192.168.1.9.

В то время как машины без затруднений обращаются друг к другу при помощи строк чисел, разделенных точками, большинство людей от этой идеи не в восторге. TCP/IP потерпел бы полное фиаско как протокол, если бы пользователям пришлось запоминать уникальные последовательности из 12 цифр для каждой машины, к которой они обращаются. Необходимо было придумать механизмы для преобразования IP-адресов в имена, понятные людям.

В этой главе рассказано об эволюции сетевых служб имен, позволяющих обращаться к данным на сайте *www.oog.org*, а не 192.168.1.9, а также о том, что происходит за сценой. По ходу дела мы будем сопровождать примеры из истории изрядным количеством полезных советов о том, как Perl помогает работать с этой важной частью любой сетевой инфраструктуры.

---

[1] В этой главе мы будем говорить о версии IPv4, являющейся в настоящее время стандартом. Вероятно, IPv6 (следующее поколение IP) вскоре ее заменит.

Файлы узлов

Первый подход, используемый для решения проблемы связи IP-адресов с именами, является самым простым и очевидным: он заключается в создании специального файла, в котором хранится таблица соответствий IP-адресов и имен компьютеров. В системах Unix это файл */etc/hosts*, в MacOS это *Macintosh HD:System Folder:Preferences:hosts* и *\$systemroot$\System32\Drivers\Etc\hosts* в NT/2000. В NT/2000 также есть файл *lmhosts*, назначение которого несколько иное, но об этом мы поговорим позже. Вот как выглядит файл узлов в Unix:

```
127.0.0.1     localhost
192.168.1.1   everest.oog.org     everest
192.168.1.2   rivendell.oog.org   rivendell
```

Ограниченность такого подхода очень быстро становится очевидной. Если в домене *oog.org* TCP/IP-сети есть две связанные между собой машины, а менеджер сети хочет добавить третью, к которой надо обращаться по имени, ему придется отредактировать соответствующий файл на всех машинах. Если в *oog.org* появится еще одна машина, то придется поддерживать четыре файла узлов (по одному на каждой машине).

И хотя такое решение кажется совершенно непригодным, именно оно использовалось на заре появления Internet/ARPAnet. Если к сети добавлялись новые сайты, то файлы узлов необходимо было обновлять на всех сайтах, которые хотели общаться с новым. Центральный репозиторий, называемый информационным центром сети (NIC) (а точнее SRI-NIC, т. к. находился тогда на SRI), обновлял и распространял файл узлов для всей сети с именем *HOSTS.TXT*. Системные администраторы периодически загружали этот файл по FTP из каталога NETINFO на сервере SRI-NIC.

Файлы узлов используются и по сей день, несмотря на их ограниченность и замену технологиями, которые нам предстоит обсудить. Существуют ситуации, когда файлы узлов даже необходимы. Например, в SunOS машина обращается к собственному файлу */etc/hosts*, чтобы определить свой IP-адрес. Файлы узлов также решают проблему «курицы и яйца», возникающую при загрузке машины. Если используемые машиной сетевые серверы имен определяются именами, то должен существовать способ определить их IP-адреса. Если же сетевые службы имен еще не действуют, то не существует способа (кроме как применять в поисках помощи широковещательные сообщения) получить эту информацию. Обычное решение – создать файл (в нем перечислено только несколько узлов), который будет использоваться для загрузки.

В маленькой сети очень полезно держать постоянно обновляемый файл узлов, в котором перечислены все машины сети. Не обязательно

даже иметь такой файл на каждой машине (т. к. другие механизмы, речь о них пойдет позже, гораздо лучше справляются с задачей распространения этой информации). Достаточно держать под рукой один файл, к которому можно обращаться вручную для просмотра адресов, а также для процедуры выдачи адреса.

Так как эти файлы по-прежнему остаются частью повседневного администрирования, рассмотрим способы их поддержки. Perl и файлы узлов просто созданы друг для друга, если вспомнить предрасположенность Perl к обработке текста. Принимая во внимание их схожесть, будем использовать простой файл узлов в качестве плацдарма для ряда исследований.

Обратите внимание на то, что анализ файлов узлов может быть очень простым:

```
open(HOSTS, "/etc/hosts") or die "Невозможно открыть файл узлов:$!\n";
while (defined ($_ = <HOSTS>)) {
    next if /^#/;  # пропускаем строки, являющиеся комментариями
    next if /^$/;  # пропускаем пустые строки
    s/\s*#.*$//;   # удаляем комментарии и
                   # предваряющие их пробелы
    ($ip, @names) = split;
    die "IP-адрес $ip уже встречался!\n"
                   if (exists $addrs{$ip});
    $addrs{$ip} = [@names];
    for (@names){
       die "Имя узла $_ уже встречалось!\n"
                   if (exists $names{$_});
       $names{$_} = $ip;
    }
}
close(HOSTS);
```

В этом примере просматривался файл */etc/hosts* (пропускались пустые строки и комментарии) и были созданы две структуры данных для дальнейшего использования. Первая структура данных – это хэш списков имен узлов, ключами которого являются IP-адреса. Вот как будет выглядеть такая структура данных для рассмотренного файла узлов:

```
$addrs{'127.0.0.1'} = ['localhost'];
$addrs{'192.168.1.2'} = ['rivendell.oog.org','rivendell'];
$addrs{'192.168.1.1'} = ['everest.oog.org','everest'];
```

Вторая структура данных – хэш-таблица имен узлов, ключами которой являются имена. Для того же самого файла хэш `%names` будет выглядеть так:

```
$names{'localhost'}='127.0.0.1'
$names{'everest'}='192.168.1.1'
```

```
$names{'everest.oog.org'}='192.168.1.1'
$names{'rivendell'}='192.168.1.2'
$names{'rivendell.oog.org'}='192.168.1.2'
```

Заметьте, что в простой процесс анализа этого файла мы добавили дополнительную функциональность. Мы проверяем, не встречаются ли в файле повторяющиеся имена и IP-адреса (и то и другое – тревожный симптом для TCP/IP-сети). Работая с данными, относящимися к сети, используйте каждую возможность, чтобы проверить отсутствие ошибок и неверной информации. Лучше выявить ошибки в самом начале, чем потом пострадать от них, когда данные распространятся уже по всей сети. К такому важному вопросу следует еще раз вернуться в этой главе.

Генерирование файлов узлов

Теперь можно заняться более интересным делом – генерированием файлов узлов. Пусть у нас есть следующий файл базы данных для всех узлов в сети:

```
name: shimmer
address: 192.168.1.11
aliases: shim shimmy shimmydoodles
owner: David Davis
department: software
building: main
room: 909
manufacturer: Sun
model: Ultra60
-=-
name: bendir
address: 192.168.1.3
aliases: ben bendoodles
owner: Cindy Coltrane
department: IT
building: west
room: 143
manufacturer: Apple
model: 7500/100
-=-
name: sulawesi
address: 192.168.1.12
aliases: sula su-lee
owner: Ellen Monk
department: design
building: main
room: 1116
manufacturer: Apple
model: 7500/100
-=-
```

```
name: sander
address: 192.168.1.55
aliases: sandy micky mickydoo
owner: Alex Rollins
department: IT
building: main
room: 1101
manufacturer: Intergraph
model: TD-325
-=-
```

Формат очень простой: *имя_поля: значение*, причем `-=-` используется в качестве разделителя между записями. Вероятно, вам потребуются иные поля или у вас будет слишком много записей, чтобы хранение их в одном «плоском» файле было оправдано. И хотя в этой главе применяется один плоский файл, принципы, приведенные здесь, не зависят от используемой базы данных.

Вот пример программы, которую можно применять для анализа подобного файла и генерирования файла узлов:

```
$datafile ="./database";
$recordsep = "-=-\n";

open(DATA,$datafile) or die "Невозможно открыть файл с данными:$!\n";

$/=$recordsep; # подготовка к чтению файла базы данных по одной записи

print "#\n# host file - GENERATED BY $0\n# DO NOT EDIT BY HAND!\n#\n";
while (<DATA>) {
    chomp;        # удалить разделитель записей
    # разбить на key1,value1,...bingo, хэш записей
    %record = split /:\s*|\n/m;
    print "$record{address}\t$record{name} $record{aliases}\n";
}
close(DATA);
```

Вот что получается:

```
#
# host file - GENERATED BY createhosts
# DO NOT EDIT BY HAND!
#
192.168.1.11    shimmer shim shimmy shimmydoodles
192.168.1.3     bendir ben bendoodles
192.168.1.12    sulawesi sula su-lee
192.168.1.55    sander sandy micky mickydoo
```

Теперь посмотрим на некоторые более интересные Perl-технологии из этого небольшого отрывка программы. Первое необычное наше действие – установка переменной `$/`. Начиная отсюда, Perl считает кусочки текста, заканчивающиеся символами `-=-\n`, одной записью. Это озна-

чает, что while за один раз прочитает всю запись и присвоит ее переменной $_.

Вторая интересная вещь – это технология присвоения значений средствами split. Наша цель состоит в получении хэша, ключами которого являются имена полей, а значениями – значения полей. Зачем нам это надо, станет понятно позже, когда мы будем расширять пример. Первый шаг заключается в разбиении $_ на части при помощи split(). Массив, который получается в результате работы split(), приведен в табл. 5.1.

Таблица 5.1. Массив, возвращенный функцией split( )

Элемент	Значение
$record[0]	Name
$record[1]	Shimmer
$record[2]	Address
$record[3]	192.168.1.11
$record[4]	Aliases
$record[5]	Shim shimmy shimmydoodles
$record[6]	Owner
$record[7]	David Davis
$record[8]	Department
$record[9]	Software
$record[10]	Building
$record[11]	Main
$record[12]	Room
$record[13]	909
$record[14]	Manufacturer
$record[15]	Sun
$record[16]	Model
$record[17]	Ultra60

Присмотримся внимательно к содержимому списка. Начиная с элемента $record[0], у нас есть список пар ключ-значение (т. е. ключ=Name, значение=shimmer\n, ключ=Address, значение=192.168.1.11\n...), который следует просто присвоить хэшу. После создания хэша можно напечатать нужные нам части.

Вы уже приняли религию «Базы данных для системного администрирования»?

В главе 3 «Учетные записи пользователей» я предложил использовать отдельную базу данных для хранения информации об учетных записях. Те же аргументы вдвойне справедливы для данных об узлах в сети. В этой главе будет показано, как можно работать даже с самой простой базой данных в формате плоского файла, чтобы получить впечатляющие результаты, нужные всем обсуждаемым службам. Для сайтов большего размера следует использовать «настоящую» базу данных. Пример получаемых данных можно найти в конце раздела «Улучшение полученного файла узлов» этой главы.

Использование базы данных для узлов в сети выгодно по целому ряду причин. В этом случае вносить изменения нужно только в один файл или источник данных. Вносите изменения, запускайте определенные сценарии и, пожалуйста, у вас есть конфигурационные файлы для множества служб. В таких конфигурационных файлах почти наверняка не будет мелких синтаксических ошибок (скажем, пропущенных точек с запятыми или символов комментариев), поскольку их не коснулись человеческие руки. Если правильно написать код, то практически все остальные ошибки можно обнаружить на стадии обработки данных анализатором (parser).

Если вы еще не осознали всей мудрости этого «лучшего подхода», к концу главы сами в этом убедитесь.

Проверка ошибок в процессе генерирования файла узлов

Напечатать нужные части – это только начало. Значительное преимущество употребления отдельной базы данных, которая преобразовывается в другую форму, состоит в возможности выполнения проверки ошибок во время преобразования. Раньше уже отмечалось, что подобный контроль позволяет избавиться от проблем, вызванных такими мелкими неприятностями, как опечатки, еще *до того*, как данные будут распространены. Вот как будет выглядеть предыдущий пример, если в него добавить проверку опечаток:

```
$datafile ="./database";
$recordsep = "-=-\n";

open(DATA,$datafile) or die "Невозможно открыть файл с данными:$!\n";

$/=$recordsep; # готовы прочитать данные из файла базы данных
```

```
                # по одной записи за один раз

print "#\n\# host file - GENERATED BY $0\n# DO NOT EDIT BY HAND!\n#\n";
while (<DATA>) {
    chomp;  # удаляем разделитель записей
    # разбиваем на key1,value1,...bingo, хэш записей
    %record = split /:\s*|\n/m;

    # проверка на неверные имена узлов
    if ($record{name} =~ /[^-.a-zA-Z0-9]/) {
      warn "!!!! $record{name} содержит недопустимые для имени узла символы,
            пропускаем...\n";
       next;
    }

    # проверка на неверные псевдонимы
    if ($record{aliases} =~ /[^-.a-zA-Z0-9\s]/) {
      warn "!!!! $record{name} содержит недопустимые для псевдонима символы,
            пропускаем...\n";
       next;
    }

    # проверка на пропущенные адреса
    if (!$record{address}) {
       warn "!!!! $record{name} не имеет IP-адреса, пропускаем...\n";
       next;
    }

    # проверка на одинаковые адреса
    if (defined $addrs{$record{address}}) {
       warn "!!!! Дублируется IP-адрес: $record{name} &
             $addrs{$record{address}}, пропускаем...\n";
       next;
    }
    else {
       $addrs{$record{address}} = $record{name};
    }

    print "$record{address}\t$record{name} $record{aliases}\n";
}
close(DATA);
```

Улучшение полученного файла узлов

Позаимствуем из главы 9 «Журналы» процесс анализа выполняемого преобразования. Мы можем автоматически добавить полезные заголовки, комментарии и разделители к получаемым данным. Вот как выглядит результат преобразования той же самой базы данных:

```
#
# host file - GENERATED BY createhosts3
```

```
# DO NOT EDIT BY HAND!
#
# Converted by David N. Blank-Edelman (dnb) on Sun Jun  7 00:43:24 1998
#
# number of hosts in the design department: 1.
# number of hosts in the software department: 1.
# number of hosts in the IT department: 2.
# total number of hosts: 4
#

# Owned by Cindy Coltrane (IT): west/143
192.168.1.3     bendir ben bendoodles

# Owned by Alex Rollins (IT): main/1101
192.168.1.55    sander sandy micky mickydoo

# Owned by Ellen Monk (design): main/1116
192.168.1.12    sulawesi sula su-lee

# Owned by David Davis (software): main/909
192.168.1.11    shimmer shim shimmy shimmydoodles
```

А вот программа (с комментариями), которая позволяет создать такой файл узлов:

```
$datafile ="./database";

# выясняем имя пользователя как в WinNT/2000, так и в Unix
$user = ($^O eq "MSWin32")? $ENV{USERNAME} :
                            (getpwuid($<))[6]." (".(getpwuid($<))[0].")";

open(DATA,$datafile) or die "Невозможно открыть файл с данными:$!\n";

$/=$recordsep; # считываем из базы данных по одной записи

while (<DATA>) {
    chomp;                          # удаляем разделитель записи
    # разбиваем на key1,value1
    @record = split /:\s*|\n/m;

    $record ={};                    # создаем ссылку на пустой хэш
    %{$record} = @record;           # заполняем этот хэш значениями
                                    # из массива @record

    # проверка на неверные имена узлов
    if ($record->{name} =~ /[^-.a-zA-Z0-9]/) {
       warn "!!!! ".$record->{name} .
            " содержит недопустимые для имени узла символы, пропускае...\n";
       next;
    }

    # проверка на неверные псевдонимы
```

```
    if ($record->{aliases} =~ /[^-.a-zA-Z0-9\s]/) {
       warn "!!!! ".$record->{name} .
            " содержит недопустимые для псевдонима символы, пропускаем...\n";
       next;
    }

    # проверка на пропущенные адреса
    if (!$record->{address}) {
       warn "!!!! ".$record->{name} .
             " не имеет IP-адреса, пропускаем...\n";
       next;
    }

    # проверка на совпадающие адреса
    if (defined $addrs{$record->{address}}) {
       warn "!!!! Дублируется IP-адрес:".$record->{name}.
            " & ".$addrs{$record->{address}}.", пропускаем...\n";
       next;
    }
    else {
       $addrs{$record->{address}} = $record->{name};
    }

    $entries{$record->{name}} = $record; # добавляем это в хэш
                                         # хэшей
}
close(DATA);

# печатаем симпатичный заголовок
print "#\n\# host file - GENERATED BY $0\n# DO NOT EDIT BY HAND!\n#\n";
print "# Converted by $user on ".scalar(localtime)."\n#\n";

# подсчитываем число записей для каждого отдела
# и сообщаем об этом
foreach my $entry (keys %entries){
    $depts{$entries{$entry}->{department}}++;
}
foreach my $dept (keys %depts) {
    print "# number of hosts in the $dept department: $depts{$dept}.\n";
}
print "# total number of hosts: ".scalar(keys %entries)."\n#\n\n";

# обходим в цикле все узлы, выводя комментарий и саму запись
foreach my $entry (keys %entries) {
    print "# Owned by ",$entries{$entry}->{owner}," (",
          $entries{$entry}->{department},"): ",
          $entries{$entry}->{building},"/",
          $entries{$entry}->{room},"\n";
    print $entries{$entry}->{address},"\t",
          $entries{$entry}->{name}," ",
          $entries{$entry}->{aliases},"\n\n";
}
```

Самое значительное отличие данного примера от предыдущего – это способ представления данных. Поскольку в предыдущем примере не было необходимости получать информацию из хэша после печати его значений, мы могли использовать единственный хэш `%record`. Но в этом случае мы решили прочитать данные из файла в более сложную структуру данных (хэш хэшей), чтобы проанализировать их перед тем как печатать.

Можно было сохранять отдельную хэш-таблицу для каждого поля (подобно тому, как это было сделано в примере *needspace* из главы 2 «Файловые системы»), но красота приведенного метода состоит в его поддерживаемости. Если затем понадобится добавить в базу данных поле `serial_number`, нам не придется менять используемый для анализа файла код, это поле само по себе появится как `$record->{serial_number}`. Недостаток же в том, что синтаксис Perl таков, что наш код выглядит более сложным, чем он есть на самом деле.

Посмотреть на все это можно еще проще: мы будем анализировать файл точно так же, как и в предыдущем примере. Разница лишь в том, что каждую запись мы будем сохранять в новом анонимном хэше. Анонимные хэши ничем не отличаются от обычных, только обращаться к ним приходится не по имени, а по ссылке.

Чтобы построить большую структуру данных (хэш хэшей), достаточно связать каждый новый анонимный хэш с основной хэш-таблицей `%entries` и создать ключ со связанным с ним значением, являющимся ссылкой на этот только что заполненный анонимный хэш. Когда каждый хэш пройдет обработку, ключами `%entries` будут имена всех машин, а значениями – ссылки на хэш-таблицы, содержащие значения всех полей, связанных с этим именем (IP-адрес, номер кабинета и т. д.).

Вероятно, вам бы хотелось, чтобы вывод был отсортирован по IP-адресам? Никаких вопросов, просто добавьте процедуру сортировки, изменив:

```
foreach my $entry (keys %entries) {
```

на:

```
foreach my $entry (sort byaddress keys %entries) {
```

и добавьте:

```
sub byaddress {
   @a = split(/\./,$entries{$a}->{address});
   @b = split(/\./,$entries{$b}->{address});
   ($a[0]<=>$b[0]) ||
   ($a[1]<=>$b[1]) ||
   ($a[2]<=>$b[2]) ||
   ($a[3]<=>$b[3]);
}
```

Вот как будут выглядеть отсортированные данные:

```
# Owned by Cindy Coltrane (IT): west/143
192.168.1.3     bendir ben bendoodles

# Owned by David Davis (software): main/909
192.168.1.11    shimmer shim shimmy shimmydoodles

# Owned by Ellen Monk (design): main/1116
192.168.1.12    sulawesi sula su-lee

# Owned by Alex Rollins (IT): main/1101
192.168.1.55    sander sandy micky mickydoo
```

Сделайте так, чтобы полученные данные вам нравились. Пусть Perl поддержит ваши профессиональные *и* эстетические стремления.

Внедрение системы контроля исходного кода

Перед тем как перейти к следующему способу преобразования IP-адресов в имена, хотелось бы добавить к процессу создания файла узлов еще одну хитрую возможность обработки, поскольку один-единственный файл приобретает общесетевое значение. Ошибка в этом файле повлияет на всю сеть. Чтобы обезопасить себя, нужен способ, выполняющий откат изменений, нарушивших файл. Особенно необходимо иметь возможность вернуться назад к предыдущим версиям файла.

Самый элегантный способ создать подобную машину времени – добавить к процессу систему контроля исходного кода. Такой контроль используется разработчиками с целью:

- Регистрации всех изменений важных файлов.
- Предотвращения ситуации, когда несколько человек одновременно изменяют один и тот же файл, тем самым ненамеренно уничтожая действия друг друга.
- Получить возможность вернуться к предыдущим версиям файла, т. е. отказаться от изменений, вызвавших проблемы.

Подобные средства контроля очень полезны для системного администратора. Проверка ошибок, добавленная в разделе «Проверка ошибок в процессе генерирования файла узлов», поможет справиться лишь с некоторыми опечатками и синтаксическими ошибками, но не спасет от семантических ошибок (таких как удаление важного имени узла, присвоение ошибочного IP-адреса, ошибка в имени узла). В процесс преобразования можно добавить проверку и семантических ошибок, но отловить все возможные погрешности все равно не удастся. Мы уже говорили, что защиты от дураков не существует, потому что они изобретательны.

Можно, наверное, предположить, что было бы лучше применить систему контроля исходного кода к процессу редактирования первоначальной базы данных, но есть две веские причины, по которым очень важно применить ее к данным, получаемым в результате:

Время

Для большого набора данных процесс преобразования может занять некоторое время. Если ваша сеть «упала», и вам необходимо вернуться к предыдущей версии, то необходимость наблюдать, пока Perl сгенерирует нужный файл, вас просто обескуражит (и все это при условии, что вы смогли сразу добраться до Perl).

База данных

Если для хранения данных вы будете использовать настоящую базу данных (а часто это правильный выбор), то может просто не существовать способа применить к ней систему контроля версий. Вероятно, вам придется писать собственные механизмы контроля версий для процесса редактирования базы данных.

В качестве системы контроля исходного кода я выбрал систему контроля версий RCS. У RCS есть несколько возможностей, дружественных к Perl и системному администрированию:

- RCS работает на многих платформах. Существуют версии GNU RCS 5.7 для большинства Unix-систем, Windows NT, MacOS и т. д.
- У нее вполне определенный интерфейс командной строки. Все действия можно выполнить из командной строки даже в операционной системе, где в основном применяется графический интерфейс.
- Ее очень легко использовать. Небольшой набор команд для выполнения основных операций можно выучить за пять минут (приложение А «Пятиминутное руководство по RCS»).
- В RCS есть ключевые слова. В текст файлов, находящихся под контролем RCS, можно добавлять магические строки, которые будут автоматически раскрываться. Например, любое вхождение в файл строки `$Date:$` будет заменено датой последнего помещения файла в систему RCS.
- RCS бесплатна. Исходный код GNU-версии RCS распространяется свободно, кроме того, доступны уже скомпилированные версии для большинства систем. Исходный код RCS можно найти на *ftp://ftp.gnu.org/gnu/rcs*.

Если вы никогда не работали с RCS, загляните сначала в приложение А. Впредь будем считать, что вы знакомы с основным набором команд RCS.

Крэйг Фретер (Craig Freter) написал объектно-ориентированный модуль `Rcs`, который упрощает применение RCS из Perl. Для этого необходимо:

1. Загрузить модуль.
2. Указать, где расположены команды RCS.
3. Создать новый объект Rcs, настроить его в соответствии с файлом, который вы используете.
4. Вызвать необходимые методы объекта (названные в соответствии с командами RCS).

Добавим модуль Rcs в программу генерации файла узлов, чтобы увидеть, как он работает, и применим другой способ вывода данных. Теперь они будут записываться в определенный файл, а не в стандартный поток вывода STDOUT, как было раньше. Ниже приведен только измененный код. Пропущенные строки, представленные в виде «...», можно найти в предыдущем примере:

```
$outputfile="hosts.$$"; # временный файл для вывода
$target="hosts";        # где сохранить преобразованные данные
...
open(OUTPUT,"> $outputfile") or
  die "Невозможно записать в файл $outputfile:$!\n";

print OUTPUT "#\n\# host file - GENERATED BY $0\n
              # DO NOT EDIT BY HAND!\n#\n";
print OUTPUT "# Converted by $user on ".scalar(localtime)."\n#\n";

...
foreach my $dept (keys %depts) {
    print OUTPUT "# number of hosts in the $dept department:
                  $depts{$dept}.\n";
}
print OUTPUT "# total number of hosts: ".scalar(keys %entries)."\n#\n\n";
# обходим в цикле все узлы и выводим комментарий
# вместе с самой записью
foreach my $entry (sort byaddress keys %entries) {
    print OUTPUT
          "# Owned by ",$entries{$entry}->{owner}," (",
          $entries{$entry}->{department},"): ",
          $entries{$entry}->{building},"/",
          $entries{$entry}->{room},"\n";
    print OUTPUT
          $entries{$entry}->{address},"\t",
          $entries{$entry}->{name}," ",
          $entries{$entry}->{aliases},"\n\n";
}

close(OUTPUT);

use Rcs;
# путь к RCS
Rcs->bindir('/usr/local/bin');
# создаем новый RCS-объект
```

```
my $rcsobj = Rcs->new;
# передаем ему имя  получаемого файла
$rcsobj->file($target);
# получаем его из репозитория RCS (он уже должен быть там)
$rcsobj->co('-l');
# переименовываем вновь созданный файл
rename($outputfile,$target) or
  die "Невозможно переименовать $outputfile в $target:$!\n";
# помещаем его в репозиторий RCS
$rcsobj->ci("-u","-m"."Converted by $user on ".scalar(localtime));
```

В данном примере предполагалось, что целевой файл хотя бы один раз помещался в репозиторий.

Взглянув на фрагмент записей из *rlog hosts*, можно понять, как действует программа:

```
revision 1.5
date: 1998/05/19 23:34:16;  author: dnb;  state: Exp;  lines: +1 -1
Converted by David N. Blank-Edelman (dnb) on Tue May 19 19:34:16 1998
----------------------------
revision 1.4
date: 1998/05/19 23:34:05;  author: eviltwin; state: Exp;  lines: +1 -1
Converted by Divad Knalb-Namlede (eviltwin) on Tue May 19 19:34:05 1998
----------------------------
revision 1.3
date: 1998/05/19 23:33:35;  author: dnb;  state: Exp;  lines: +20 -0
Converted by David N. Blank-Edelman (dnb) on Tue May 19 19:33:16 1998
```

Из предыдущего примера видно, что между версиями файла нет больших различий (обратите внимание на часть, включающую `lines:`), зато отслеживаются все изменения, происходящие при создании файла. При необходимости узнать, что именно произошло, достаточно воспользоваться командой *rcsdiff*. В крайнем случае, всегда можно вернуться к предыдущим версиям, если какие-либо изменения приведут сеть в неработоспособное состояние.

NIS, NIS+ и WINS

Разработчики из Sun Microsystems, осознав, что «редактирование по одному файлу на каждой машине» вряд ли можно назвать масштабируемым подходом, придумали нечто под названием *желтые страницы* (*Yellow Pages* или *YP*). YP были созданы для распространения информации из конфигурационных файлов сетевого масштаба, таких как */etc/hosts*, */etc/passwd*, */etc/services* и т. д. В этой главе мы остановимся на использовании «желтых страниц» в качестве информационной службы сети для предоставления сведений о связи между именем машины и ее IP-адресом.

YP были переименованы в *Информационную службу сети* (Network Information Service или NIS) в 1990 г., когда компания British Telecom (совместно с юристами) заявила о правах на торговую марку «Yellow Pages» в Великобритании. Призрак «желтых страниц» по-прежнему витает во многих Unix-системах, и сегодня для команд и библиотечных вызовов NIS употребляются имена *ypcat*, *ypmatch*, *yppush* и т. д. Все современные разновидности Unix поддерживают NIS. На машинах с NT можно использовать NIS для авторизации, если применить специальные библиотеки авторизации[1], но о существовании NIS-серверов для NT я не знаю. Мне также не известно о существовании NIS для MacOS.

В NIS администратор определяет одну или несколько машин как серверы, от которых другие машины получают клиентский сервис. Один из серверов является *главным* (*master*), все остальные – *подчиненные* (*slave*). На главном сервере хранятся основные копии текстовых файлов (например */etc/hosts* или */etc/passwd*), которые используют все машины. Эти файлы изменяются на главном сервере и затем распространяются на подчиненные.

Теперь, если машине в сети необходимо получить информацию о связи IP-адресов и имен узлов, она обращается к серверу, вместо того чтобы хранить собственную локальную копию этой информации. Клиент может запрашивать информацию как с главного, так и с любого из подчиненных серверов. Запросы клиентов просматриваются в NIS-*картах*. Карты – это другое название основных файлов, уже преобразованных в формат базы данных Unix DBM и распространенных на подчиненные серверы. С подробностями этого процесса преобразования (который включает в себя *makedbm* и некоторые другие изменения текста) можно ознакомиться в файле *Makefile*, в большинстве случаев расположенном в */var/yp*. Группа NIS-серверов и клиентов, использующих одни и те же карты, называется NIS-*доменом*.

NIS значительно упрощает администрирование сетей. Так, если в сети *oog.org* появляются новые машины, то добавить их в существующую сеть совсем не сложно. Менеджер сети редактирует файл узлов на главном NIS-сервере и «проталкивает» новую версию на все подчиненные. Теперь каждый клиент из NIS-домена будет «знать» о существовании новых машин. NIS обеспечивает легкость администрирования, объединенную с избыточностью (если один сервер недоступен, клиент может обратиться к другому) и распределением нагрузки (не все клиенты в сети используют один и тот же сервер).

Теперь, когда мы знакомы с теорией, можно посмотреть, как Perl помогает в вопросах, связанных с NIS. Начнем мы с процесса размеще-

[1] Одна из таких библиотек – NISGINA, первоначально разработанная Найджелом Вильямсом (Nigel Williams); эту библиотеку можно найти на *http://www.dcs.qmw.ac.uk/~williams/*. Но сначала изучите архивы списков рассылки, чтобы выяснить, какая версия является самой новой.

ния данных в NIS. Вы, наверное, удивитесь, но это уже практически сделано. Файлы узлов, созданные в предыдущем разделе, можно импортировать в NIS, просто перенеся их в нужное место в каталоге исходных файлов на главном сервере и активировав обычные механизмы принудительной рассылки (проталкивания, push mechanisms), – как правило, для этого надо выполнить *make* в */var/yp*. По умолчанию *Makefile* из каталога */var/yp* использует в качестве исходников для NIS-карт содержимое конфигурационных файлов главного сервера.

Обычно имеет смысл создать отдельный каталог для исходных файлов NIS-карт и соответствующим образом изменить *Makefile*. Это позволит хранить различные данные для главного сервера и остальных членов NIS-домена. Например, вы можете не захотеть, чтобы файл */etc/passwd* с главного сервера применялся в качестве файла паролей для всего домена, или наоборот.

Более интересная задача – получить данные из NIS, запрашивая NIS-сервер. Проще всего это сделать при помощи модуля `Net::NIS` Рика Хариса (Rik Harris). Данный модуль, начиная с 1995 года, находится в состоянии альфа-версии, но, тем не менее, он вполне рабочий.[1]

Вот пример, позволяющий получить и напечатать содержимое карты при помощи одной функции, применяя `Net::NIS`. Это похоже на команду NIS *ypcat*:

```
use Net::NIS;
# NIS-домен по умолчанию
$domain = Net::NIS::yp_get_default_domain();
# считываем карту
($status, $info) = Net::NIS::yp_all($domain,"hosts.byname");
foreach my $name (sort keys %{$info}){
    print "$name => $info->{$name}\n";
}
```

Сначала необходимо обратиться к локальному узлу для получения имени домена. Используя эту информацию, можно вызвать функцию `Net::NIS::yp_all()` и получить карту. Функция возвращает переменную состояния (фиктивную, как видно из сноски) и ссылку на хэш-таблицу, содержащую данные из этой карты. Мы выводим эту информацию, применяя обычный синтаксис разыменования.

[1] Я знаю всего лишь об одной серьезной ошибке в версии a2. В документации рекомендуется сравнивать статус возврата вызовов модуля с предопределенными константами, например `$Net::NIS::ERR_KEY` и `$Net::NIS::ERR_MAP`. К сожалению, эти константы в модуле никогда не определяются. Самый простой способ выяснить, был ли запрос удачным, заключается в том, чтобы проверить длину возвращенных данных.

Если нужно узнать только IP-адрес одного узла, эффективнее было бы запросить у сервера именно это значение:

```
use Net::NIS;
$hostname = "olaf.oog.org";
$domain = Net::NIS::yp_get_default_domain();
($status,$info) = Net::NIS::yp_match($domain,"hosts.byname",$hostname);
print $info,"\n";
```

Функция `Net::NIS::yp_match()` возвращает еще одну фиктивную переменную состояния и значение (скаляр), соответствующее запрашиваемой информации.

Если не удается скомпилировать модуль `Net::NIS` или он просто не работает, всегда остается возможность вызвать внешнюю программу. Например, так:

```
@hosts=`<путь к>/ypcat hosts`
```

или так:

```
open(YPCAT,"<путь к>/ypcat hosts|");
while (<YPCAT>){...}
```

Завершит данный раздел пример, в котором применяются оба способа. Этот маленький, но полезный фрагмент программы получает список текущих NIS-серверов и опрашивает каждый из них при помощи программы *yppoll*. Если какой-либо из серверов не отвечает как полагается, выводится соответствующее сообщение:

```
use Net::NIS;

$yppollex = "/usr/etc/yp/yppoll"; # полный путь к программе yppoll

$domain = Net::NIS::yp_get_default_domain();

($status,$info) = Net::NIS::yp_all($domain,"ypservers");

foreach my $name (sort keys %{$info}) {
    $answer = `$yppollex -h $name hosts.byname`;
    if ($answer !~ /has order number/) {
        warn "$name отвечает неверно!\n";
    }
}
```

NIS+

В состав операционной системы Solaris входит NIS+ – следующая версия NIS. В NIS+ решены многие из наиболее серьезных проблем, которые были в NIS, в частности, проблема безопасности. К сожалению (а может быть и к счастью, т. к. NIS+ администрировать несколько

сложнее), NIS+ не стала столь популярной в мире Unix, как NIS. До недавнего времени она практически не поддерживалась на машинах, созданных не в Sun. NIS+ постепенно «приживается» в стандартных дистрибутивах Linux, благодаря работе Торстена Кукука (Thorsten Kukuk) (*http://www.suse.de/~kukuk/nisplus/index.html*), но она отнюдь не преобладает в мире Unix и ее просто не существует в NT и MacOS.

Принимая во внимание то, что NIS+ используется незначительно, говорить о ней в книге мы больше не будем. Если вам необходимо работать с NIS+ из Perl, можете применять еще один модуль Хариса – `Net::NISPlus`.

Windows-служба имен Интернета (WINS)

Когда в Microsoft стали использовать свой патентованный сетевой протокол NetBIOS поверх TCP/IP (NetBT), возникла необходимость решать проблему соответствия IP-адресов и имен узлов. Первым решением стало использование файла *lmhosts*, спроектированного по аналогии со стандартным файлом узлов. Но это было быстро дополнено NIS-подобным механизмом. В NT 3.5 появилась централизованная схема под названием Windows-служба имен Интернета (Windows Internet Name Service, или WINS). WINS несколько отличается от NIS:

- WINS специализируется на распространении информации о соответствии имен узлов IP-адресам. В отличие от NIS, эта служба не применяется для централизованного распространения другой информации (например, паролей, карты портов и групп пользователей).
- WINS-сервера получают большую часть из распространяемой информации от предварительно настроенных клиентов (такую информацию можно предварительно загрузить). После получения IP-адреса либо вручную, либо через протокол динамической конфигурации узла (Dynamic Host Configuration Protocol, DHCP) WINS-клиенты ответственны за регистрацию и перерегистрацию своей информации. В этом состоит различие с NIS, там клиенты запрашивают информацию у сервера и, за исключением паролей, не обновляют на нем информацию.

WINS, как и NIS, позволяет иметь несколько серверов, повышающих надежность и разделяющих загрузку, по принципу«тяни-толкай». В Windows 2000 WINS вышла из употребления (читай «от нее избавились»), вместо нее теперь используется служба динамических доменных имен (Dynamic Domain Name Service), являющаяся расширением DNS-системы, о которой мы очень скоро поговорим еще.

Учитывая, что WINS больше не существует, мы не будем приводить примеров для работы с ней. В настоящее время работа с WINS напрямую из Perl поддерживается очень слабо. Я не знаю о существовании модулей, созданных специально для работы с WINS. В этом случае лучше всего вызывать некоторые утилиты, работающие в командной

строке из Windows NT Server Resource Kit, например *WINSCHK* и *WINSCL*.

Служба доменных имен (DNS)

Несмотря на то, что NIS и WINS крайне полезны, им все же недостает некоторых свойств, что делает их непригодными для использования во «всем Интернете».

Масштабируемость

Хотя эти схемы применяются к нескольким серверам, каждый сервер должен обладать полной копией информации о топологии сети.[1] Такую информацию следует скопировать на каждый сервер, а этот процесс требует времени, если сеть становится достаточно большой. Кроме того, WINS страдает из-за своей динамической модели регистрации. Некоторое число WINS-клиентов своими регистрационными запросами может расплавить от перегрузки любое количество WINS-серверов для всего Интернета.

Управление

До сих пор мы говорили только о технических аспектах, но это не единственная сторона администрирования. NIS, в особенности, требует единственного центра администрирования. Тот, кто управляет главным сервером, управляет и всем NIS-доменом, который этот сервер «возглавляет». Любые изменения в пространстве сетевых имен должны пройти через такого сторожа. Этот принцип не будет работать в пространстве имен размером во весь Интернет.

Для борьбы с недостатками, присущими сопровождению файлов узлов или NIS/NIS+/WINS-подобным системам, была создана новая модель под названием *служба доменных имен* (DNS). В DNS пространство имен сети разделено на несколько «доменов верхнего уровня». Любой из них можно разделить на домены меньшего размера и т. д. В каждой точке деления следует назначить сторону, ответственную за контроль над этой частью пространства имен, что позволяет разобраться с вопросами администрирования.

Клиенты в сети обращаются к ближайшему по иерархии серверу имен. Если информацию, которую ищет клиент, можно найти на данном локальном сервере, она возвращается клиенту. В большинстве сетей основная часть запросов, касающихся разыменования адресов, относится к машинам из той же сети, поэтому локальные серверы обрабатывают большую часть локального трафика. Это позволяет избавиться от проблемы масштабируемости. Можно настроить несколько DNS-сер-

[1] NIS+ предлагает механизмы поиска информации для клиентов за пределами локального домена, но они не настолько гибкие, как в DNS.

веров (также известных как вторичные или подчиненные), чтобы распределить загрузку и повысить надежность.

Если запрос к DNS-серверу относится к части пространства имен, не контролируемой или не известной серверу, он может либо сказать клиенту, что поиск необходимо проводить в каком-то другом месте (обычно выше по дереву), либо получить необходимую информацию, обратившись от имени клиента к другим DNS-серверам.

В такой схеме ни один сервер не должен знать о топологии всей сети, большинство запросов обрабатывается локально, за локальными администраторами сохраняется локальный контроль, и в результате все счастливы. У DNS есть преимущество перед другими службами – большинство других систем, подобных NIS и WINS, можно интегрировать с DNS. Например, NIS-серверы в SunOS можно настроить так, чтобы они обращались к DNS-серверу, если клиент запрашивает у них имя узла, о котором сервер не знает. Результаты этого запроса возвращаются как стандартные NIS-ответы, так что клиенты и не догадываются о каких-либо дополнительных действиях. DNS-серверы Microsoft обладают схожей функциональностью: если клиент запрашивает у DNS-сервера Microsoft адрес локальной машины, о которой ему неизвестно, то DNS-сервер можно настроить так, чтобы он пересылал этот запрос WINS-серверу от имени клиента.

Генерирование конфигурационных файлов DNS

Процесс создания конфигурационных файлов DNS очень похож на тот, который мы использовали для создания файлов узлов и исходных файлов NIS:

- Данные хранятся в отдельной базе данных (одна и та же база может и, вероятно, должна быть источником для всех файлов, о которых идет речь).
- Данные преобразуются в формат вывода по нашему выбору, при этом проверяются ошибки.
- Используется RCS (или эквивалентная система контроля версий) для хранения предыдущих версий файлов.

В случае с DNS второй шаг необходимо расширить, поскольку здесь процесс преобразования оказывается более сложным. Сложности нужно преодолевать, поэтому было бы неплохо иметь под рукой книгу Пола Альбица (Paul Albitz) и Крикета Лью (Cricket Liu) «DNS and BIND»[1] («DNS и BIND», O'Reilly), содержащую, в том чисе, сведения о конфигурационных файлах, создание которых рассматривается ниже.

[1] Пол Альбиц, Крикет Лью «DNS и BIND» (перевод 4-го издания 2001 г.), IV кв., издательство «Символ-Плюс», 2001 г. – *Примеч. ред.*

Создание административного заголовка

Конфигурационные файлы DNS начинаются с административного заголовка, в нем представлена информация о сервере и данных, которые он обслуживает. Самая важная часть этого заголовка – запись о ресурсах SOA (Start of Authority). Запись SOA содержит:

- Имя административного домена, обслуживаемого данным DNS-сервером.
- Имя первичного DNS-сервера этого домена.
- Контактную информацию об администраторе (администраторах) DNS-сервера.
- Порядковый номер конфигурационного файла (подробнее об этом рассказывается чуть ниже).
- Значения тайм-аутов регенерации (refresh) и повторного обращения (retry) для вспомогательных серверов (т. е. информация о том, когда необходимо синхронизировать данные с первичным сервером).
- Время жизни (TTL) для данных (т. е. в течение какого времени можно безопасно кэшировать информацию).

Вот как может выглядеть этот заголовок:

```
@ IN SOA   dns.oog.org. hostmaster.oog.org. (
                          1998052900 ; serial
                            10800    ; refresh
                            3600     ; retry
                            604800   ; expire
                            43200)   ; TTL

@                           IN  NS  dns.oog.org.
```

Бульшая часть информации добавляется в начало конфигурационного файла каждый раз при его создании. Единственное, о чем нужно побеспокоиться, – это о порядковом номере. Один раз в *X* секунд (*X* определяется из значения регенерации) вторичные серверы имен сверяются с первичными серверами, чтобы узнать, нужно ли обновить данные. Современные вторичные DNS-серверы (подобные BIND v8+ или Microsoft DNS) могут быть сконфигурированы так, что будут сверяться с основным сервером в то время, когда на последнем меняются данные. В обоих случаях вторичный сервер запрашивает на первичном запись SOA. Если порядковый номер записи SOA первичного сервера больше порядкового номера, хранимого на вторичном сервере, то произойдет перенос информации о зоне (вторичный сервер загрузит новые данные). В итоге, важно увеличивать порядковый номер каждый раз при создании нового конфигурационного файла. Многие из проблем с DNS вызваны неполадками при обновлении порядкового номера.

Существует по крайней мере два способа сделать так, чтобы порядковый номер всегда увеличивался:

1. Считывать предыдущий конфигурационный файл и увеличивать найденное там значение.
2. Вычислять новое значение, основываясь на внешних данных, которые «гарантированно» увеличиваются (это могут быть, например, системные часы или номера версий файла в RCS).

Ниже приведен пример программы, где применяется комбинация этих двух методов для создания допустимого заголовка файла зоны DNS. Порядковый номер будет представлен в виде, который рекомендуют использовать Альбиц и Лью в своей книге (YYYYMMDDXX, где Y=год, M=месяц, D=день и XX=двузначный счетчик, позволяющий вносить более одного изменения за день):

```
# получаем текущую дату в формате YYYYMMDD
@localtime = localtime;
$today = sprintf("%04d%02d%02d",$localtime[5]+1900,
                                $localtime[4]+1,
                                $localtime[3]);

# имя пользователя как в NT/2000, так и в Unix
$user = ($^O eq "MSWin32")? $ENV{USERNAME} :
                            (getpwuid($<))[6]." (".(getpwuid($<))[0].")";

sub GenerateHeader{
    my($header);

    # открываем старый файл, если это возможно, и считываем
    # порядковый номер, принимая во внимание формат старого файла

    if (open (OLDZONE,$target)){
       while (<OLDZONE>) {
           next unless (/(\d{8}).*serial/);
           $oldserial = $1;
           last;
       }
       close (OLDZONE);
    }
    else {
       $oldserial = "00000000"; #иначе начинаем с 0
    }

    # если предыдущий порядковый номер соответствует
    # сегодняшнему дню, то увеличиваем последние 2 цифры, в
    # противном случае используем новый номер для сегодняшнего дня
    $olddate = substr($oldserial,0,8);
    $count = (($olddate == $today) ? substr($oldserial,8,2)+1 : 0);

    $serial = sprintf("%8d%02d",$today,$count);
```

```
    # начало заголовка
    $header .= "; Файл зоны dns - СОЗДАН $0\n";
    $header .= "; НЕ РЕДАКТИРУЙТЕ ВРУЧНУЮ!\n;\n";
    $header .= "; преобразован пользователем $user в
               ".scalar((localtime))."\n;\n";

    # пересчитать число записей для каждого отдела и сообщить
    foreach my $entry (keys %entries){
        $depts{$entries{$entry}->{department}}++;
    }
    foreach my $dept (keys %depts) {
        $header .= "; число узлов в отделе $dept:
                    $depts{$dept}.\n";
    }
    $header .= "; всего узлов: ".scalar(keys %entries)."\n;\n\n";

    $header .= <<"EOH";

@ IN SOA   dns.oog.org. hostmaster.oog.org. (
                            $serial ; serial
                             10800    ; refresh
                             3600     ; retry
                             604800   ; expire
                             43200)   ; TTL

@                            IN  NS  dns.oog.org.

EOH

   return $header;
}
```

В примере осуществляется попытка прочитать предыдущий конфигурационный файл для определения последнего порядкового номера. Затем это значение разбивается на поля даты и счетчика. Если прочитанная дата совпадает с текущей, необходимо увеличить значение счетчика. Если нет, то в новом порядковом номере поле даты совпадает с текущей датой, а значение счетчика равно 00. Теперь, когда порядковый номер проверен, можно вывести заголовок в правильном виде.

Создание нескольких конфигурационных файлов

После того как написан верный заголовок для конфигурационных файлов, осталось решить еще одну проблему. Правильно настроенный DNS-сервер поддерживает как прямое преобразование (имен в IP-адреса), так и обратное преобразование (IP-адресов в имена) для каждого домена (или зоны), который он обслуживает. Для этого надо иметь два конфигурационных файла на каждую зону. Самый лучший способ их синхронизировать – создавать файлы в одно и то же время.

Рассмотрим в данной главе последний пример генерирования файлов, поэтому соберем воедино все, что обсуждали раньше. Приведенный сценарий использует для создания конфигурационных файлов зоны DNS простой файл базы данных.[1]

Чтобы не усложнять сценарий, я сделал ряд предположений относительно данных, самые важные из которых касаются топологии сети и пространства имен. Я считаю, что сеть состоит из одной подсети класса C с одной зоной DNS. В результате, необходимо создать один файл для прямого преобразования имен и один для обратного. Добавить код для работы с несколькими подсетями и зонами (т. е. создать отдельные файлы для каждой) будет несложно.

Вот, вкратце, что мы делаем:

1. Считываем файл базы данных в хэш хэшей, проверяя при этом данные.
2. Генерируем заголовок.
3. Записываем данные в файл для прямого преобразования (из имен в IP-адреса) и помещаем его под контроль RCS.
4. Записываем данные в файл для обратного преобразования (из IP-адресов в имена) и помещаем его под контроль RCS.

Вот как выглядит пример и получаемые в результате файлы:

```
use Rcs;

$datafile   = "./database"; # база данных узлов
$outputfile = "zone.$$";    # временный файл для вывода
$target     = "zone.db";    # получаемый файл
$revtarget  = "rev.db";     # получаемый файл для обратного преобразования
$defzone    = ".oog.org";   # создаваемая по умолчанию зона
$recordsep  = "-=-\n";

# получаем текущую дату в формате YYYYMMDD
@localtime = localtime;
$today = sprintf("%04d%02d%02d",$localtime[5]+1900,
                                $localtime[4]+1,
                                $localtime[3]);

# имя пользователя, как в NT/2000, так и Unix
$user = ($^O eq "MSWin32")? $ENV{USERNAME} :
                            (getpwuid($<))[6]." (".(getpwuid($<))[0].")";
$/ = $recordsep;

# считываем файл базы данных
open(DATA,$datafile) or die "Ошибка! Невозможно открыть datafile:$!\n";
```

[1] Имеется в виду простой текстовый файл с данными, а не файл базы данных в прямом смысле этого слова. – *Примеч. науч. ред.*

```
while (<DATA>) {
    chomp; # удаляем разделитель записей
    # разбиваем на key1,value1
    @record = split /:\s*|\n/m;

    $record ={};                  # создаем ссылку на пустой хэш
    %{$record} = @record;         # заполняем его значениями из @record

    # ищем ошибки в именах узлов
    if ($record->{name} =~ /[^-.a-zA-Z0-9]/) {
       warn "!!!! ",$record->{name} .
         " встретились недопустимые в именах узлов символы, пропускаем...\n";
       next;
    }

    # ищем ошибки в псевдонимах
    if ($record->{aliases} =~ /[^-.a-zA-Z0-9\s]/) {
       warn "!!!! " . $record->{name} .
          " встретились недопустимые в псевдонимах символы, пропускаем...\n";
       next;
    }

    # ищем пропущенные адреса
    unless ($record->{address}) {
       warn "!!!! " . $record->{name} .
            " нет IP-адреса, пропускаем...\n";
       next;
    }

    # ищем повторяющиеся адреса
    if (defined $addrs{$record->{address}}) {
       warn "!!!! Повторение IP-адреса:" . $record->{name}.
            " & " . $addrs{$record->{address}} . ", пропускаем...\n";
       next;
    }
    else {
       $addrs{$record->{address}} = $record->{name};
    }

    $entries{$record->{name}} = $record; # добавляем это в хэш хэшей

}
close(DATA);

$header = &GenerateHeader;

# создаем файл прямого преобразования
open(OUTPUT,"> $outputfile") or
  die "Ошибка! Невозможно записать в $outputfile:$!\n";
print OUTPUT $header;
```

```
foreach my $entry (sort byaddress keys %entries) {
    print OUTPUT
          "; Владелец -- ",$entries{$_}->{owner}," (",
          $entries{$entry}->{department},"): ",
          $entries{$entry}->{building},"/",
          $entries{$entry}->{room},"\n";

    # выводим запись A
    printf OUTPUT "%-20s\tIN A     %s\n",
      $entries{$entry}->{name},$entries{$entry}->{address};

    # выводим записи CNAMES (псевдонимы)
    if (defined $entries{$entry}->{aliases}){
       foreach my $alias (split(' ',$entries{$entry}->{aliases})) {
           printf OUTPUT "%-20s\tIN CNAME %s\n",$alias,
                                                $entries{$entry}->{name};
       }
    }
    print OUTPUT "\n";
}

close(OUTPUT);

Rcs->bindir('/usr/local/bin');
my $rcsobj = Rcs->new;
$rcsobj->file($target);
$rcsobj->co('-l');
rename($outputfile,$target) or
  die "Ошибка! Невозможно переименовать $outputfile в $target:$!\n";
$rcsobj->ci("-u","-m"."Преобразовано пользователем $user в
".scalar(localtime));

# создаем файл обратного преобразования
open(OUTPUT,"> $outputfile") or
  die "Ошибка! Невозможно записать в $outputfile:$!\n";
print OUTPUT $header;
foreach my $entry (sort byaddress keys %entries) {
    print OUTPUT
          "; Владелец -- ",$entries{$entry}->{owner}," (",
          $entries{$entry}->{department},"): ",
          $entries{$entry}->{building},"/",
          $entries{$entry}->{room},"\n";

    printf OUTPUT "%-3d\tIN PTR    %s$defzone.\n\n",
      (split/\./,$entries{$entry}->{address})[3],
      $entries{$entry}->{name};

}

close(OUTPUT);
```

```
$rcsobj->file($revtarget);
$rcsobj->co('-l'); # предполагаем, что целевой файл по крайней
                   # мере один раз извлекался из репозитория
rename($outputfile,$revtarget) or
  die "Ошибка! Невозможно переименовать $outputfile в $revtarget:$!\n";
$rcsobj->ci("-u","-m"."Преобразовано пользователем $user в
".scalar(localtime));

sub GenerateHeader{
    my($header);
    if (open(OLDZONE,$target)){
       while (<OLDZONE>) {
           next unless (/(\d{8}).*serial/);
           $oldserial = $1;
           last;
       }
       close(OLDZONE);
    }
    else {
       $oldserial = "000000";
    }

    $olddate = substr($oldserial,0,6);
    $count = ($olddate == $today) ? substr($oldserial,6,2)+1 : 0;

    $serial = sprintf("%6d%02d",$today,$count);

    $header .= "; файл зоны dns - СОЗДАН $0\n";
    $header .= "; НЕ РЕДАКТИРУЙТЕ ВРУЧНУЮ!\n;\n";
    $header .= "; Преобразован пользователем $user в
               ".scalar(localtime)."\n;\n";

    # подсчитываем число узлов в каждом отделе
    foreach $entry (keys %entries){
        $depts{$entries{$entry}->{department}}++;
    }
    foreach $dept (keys %depts) {
        $header .= "; в отделе $dept $depts{$dept} машин.\n";
    }
    $header .= "; общее число машин: ".scalar(keys %entries)."\n#\n\n";

    $header .= <<"EOH";

@ IN SOA   dns.oog.org. hostmaster.oog.org. (
                           $serial     ; serial
                             10800     ; refresh
                             3600      ; retry
                             604800    ; expire
                             43200)    ; TTL

@                            IN  NS  dns.oog.org.
```

```
EOH

   return $header;
}

sub byaddress {
   @a = split(/\./,$entries{$a}->{address});
   @b = split(/\./,$entries{$b}->{address});
   ($a[0]<=>$b[0]) ||
   ($a[1]<=>$b[1]) ||
   ($a[2]<=>$b[2]) ||
   ($a[3]<=>$b[3]);
}
```

Вот какой файл получается для прямого преобразования (*zone.db*):

```
; файл зоны dns - СОЗДАН createdns
; НЕ РЕДАКТИРУЙТЕ ВРУЧНУЮ!
;
; Преобразован пользователем David N. Blank-Edelman (dnb);
; в Fri May 29 15:46:46 1998
;
; в отделе design 1 машин.
; в отделе software 1 машин.
; в отделе IT 2 машин.
; общее число машин: 4
;

@ IN SOA   dns.oog.org. hostmaster.oog.org. (
                          1998052900 ; serial
                            10800    ; refresh
                            3600     ; retry
                            604800   ; expire
                            43200)   ; TTL

@                           IN  NS  dns.oog.org.

; Владелец -- Cindy Coltrane (marketing): west/143
bendir                IN A     192.168.1.3
ben                   IN CNAME bendir
bendoodles            IN CNAME bendir

; Владелец -- David Davis (software): main/909
shimmer               IN A     192.168.1.11
shim                  IN CNAME shimmer
shimmy                IN CNAME shimmer
shimmydoodles         IN CNAME shimmer

; Владелец -- Ellen Monk (design): main/1116
sulawesi              IN A     192.168.1.12
```

```
sula                    IN CNAME sulawesi
su-lee                  IN CNAME sulawesi

; Владелец -- Alex Rollins (IT): main/1101
sander                  IN A     192.168.1.55
sandy                   IN CNAME sander
micky                   IN CNAME sander
mickydoo                IN CNAME sander
```

А вот как выглядит файл для обратного преобразования (*rev.db*):

```
; файл зоны dns - СОЗДАН createdns
; НЕ РЕДАКТИРУЙТЕ ВРУЧНУЮ!
;
; Преобразован пользователем David N. Blank-Edelman (dnb);
; в Fri May 29 15:46:46 1998
;
; в отделе design 1 машин.
; в отделе software 1 машин.
; в отделе IT 2 машин.
; общее число машин: 4
;

@ IN SOA   dns.oog.org. hostmaster.oog.org. (
                          1998052900 ; serial
                            10800    ; refresh
                            3600     ; retry
                            604800   ; expire
                            43200)   ; TTL

@                           IN  NS  dns.oog.org.

; Владелец -- Cindy Coltrane (marketing): west/143
3  IN PTR    bendir.oog.org.

; Владелец -- David Davis (software): main/909
11 IN PTR    shimmer.oog.org.

; Владелец -- Ellen Monk (design): main/1116
12 IN PTR    sulawesi.oog.org.

; Владелец -- Alex Rollins (IT): main/1101
55 IN PTR    sander.oog.org.
```

Этот метод создания файлов открывает перед нами много возможностей. До сих пор мы генерировали файлы, используя содержимое одного текстового файла базы данных. Запись из базы данных считывалась и записывалась в файл, возможно, подвергаясь при этом форматированию. Таким образом, в создаваемые файлы попадали только записи из базы данных.

Иногда бывает полезно, чтобы сценарий добавлял в процессе преобразования свои предопределенные данные. Например, в случае с конфигурационными файлами DNS можно улучшить сценарий преобразования так, чтобы он добавлял записи MX (Mail eXchange), указывающие на центральный почтовый сервер, для каждого узла из базы данных. Простое изменение нескольких строк кода с таких:

```
# выводим запись A
    printf OUTPUT "%-20s\tIN A     %s\n",
      $entries{$entry}->{name},$entries{$entry}->{address};
```

на следующие:

```
# выводим запись A
printf OUTPUT "%-20s\tIN A     %s\n",
      $entries{$entry}->{name},$entries{$entry}->{address};

# выводим запись MX
print OUTPUT "                    IN MX 10 $mailserver\n";
```

приведет к тому, что почта, посылаемая на любой из узлов домена, будет направляться на машину `$mailserver`. Если эта машина настроена так, что может обрабатывать почту для всего домена, то мы задействовали очень важный компонент инфраструктуры (централизованную обработку почты), добавив всего лишь одну строчку кода на Perl.

Проверка работы DNS: итеративный подход

Мы потратили значительное время на создание конфигурационных файлов, используемых сетевыми службами имен, но это всего лишь одна из задач системного и сетевого администратора. Для поддержания сети в рабочем состоянии необходимо постоянно проверять данные службы, чтобы убедиться, что они ведут себя верно.

Например, для системного/сетевого администратора очень многое зависит от ответа на вопрос «Все ли DNS-серверы работают?». В ситуации, когда необходимо найти неисправности, практически настолько же важно знать, «Все ли серверы работают с одной и той же информацией?», или, более точно, «Отвечают ли они одинаково на одинаковые запросы? Синхронизированы ли они?». Данный раздел посвящен подобным вопросам.

По главе 2 можно судить, как действует основной принцип Perl «Всегда существует несколько способов сделать это». Именно такое свойство делает Perl отличным языком для «итеративной разработки». Итеративная разработка – это один из способов описания эволюционного процесса, имеющего место при создании программ системного администрирования (и не только), выполняющих определенную задачу. В случае с Perl можно быстро написать рабочую программу на скорую руку, а позднее вернуться к сценарию и переписать его более элегант-

ным образом. Возможно, будет еще и третья итерация, на этот раз уже с использованием другого подхода к решению задачи.

Существует три различных подхода к одной и той же проблеме проверки согласованности DNS. Они представлены в том порядке, которому, действительно, мог бы последовать человек, пытаясь найти решение, а затем его совершенствуя. Этот порядок отражает взгляд на то, как решение проблемы может развиваться в Perl; ибо ваше отношение к подходу может меняться. Третий способ, использующий модуль `Net::DNS`, вероятно, самый простой и наиболее защищенный от ошибок. Но существуют ситуации, когда `Net::DNS` применять нельзя, поэтому сначала приведем несколько собственных решений. Обязательно обратите внимание на все за и против, перечисленные после каждого рассмотренного подхода.

Вот наша задача: написать сценарий на Perl, принимающий имя узла и проверяющий список DNS-серверов, чтобы убедиться, что все они возвращают одну и ту же информацию об узле. Чтобы упростить задачу, будем считать, что узел имеет единственный статический IP-адрес (т. е. у него один сетевой адаптер и один IP-адрес).

Перед тем как перейти к рассмотрению всех подходов, взглянем на сердцевину кода, который будем применять:

```
$hostname = $ARGV[0];
@servers = qw(nameserver1 nameserver2 nameserver3); # серверы имен

foreach $server (@servers) {
    &lookupaddress($hostname,$server);          # заполняем %results
}
%inv = reverse %results;                        # инвертируем полученный хэш
if (keys %inv > 1) {
    print "Между DNS-серверами есть разногласия:\n";
    use Data::Dumper;
    print Data::Dumper->Dump([\%results],["results"]),"\n";
}
```

Для каждого из DNS-серверов, перечисленных в списке `@servers`, вызывается подпрограмма `&lookupaddress()`, которая обращается к DNS-серверу, чтобы получить IP-адрес заданного имени узла, и помещает результаты в хэш `%results`. Для каждого DNS-сервера в хэше `%results` есть запись, значением которой является IP-адрес, возвращаемый этим сервером (ключом является имя сервера).

Существует много способов определить, равны ли друг другу значения из хэша `%results` (т. е. убедиться, что все DNS-серверы возвращают одну и ту же информацию в ответ на запрос). Мы инвертируем хэш `%results` в другую хэш-таблицу, преобразовывая все ключи в значения и наоборот. Если все значения из `%results` одинаковы, то в инвертированном хэше должен быть только один ключ. Если ключей несколько, значит, мы выловили прокол, и поэтому вызываем `Data::Dumper->Dump()` для

вывода содержимого %results, над которым будет ломать голову системный администратор.

Вот как может выглядеть примерный результат, если что-то идет не так:

```
Между DNS-серверами есть разногласия:
$results = {
             nameserver1 => '192.168.1.2',
             nameserver2 => '192.168.1.5',
             nameserver3 => '192.168.1.2',
           };
```

Теперь посмотрим на альтернативы подпрограмме &lookupaddress().

Использование nslookup

Если у вас есть опыт работы в Unix или вы уже программировали на других языках сценариев помимо Perl, то первая попытка может сильно походить на сценарий командного интерпретатора. Внешняя программа, вызываемая из Perl сценария, выполняет всю сложную работу:

```
use Data::Dumper;

$hostname = $ARGV[0];
$nslookup = "/usr/local/bin/nslookup";        # путь к nslookup
@servers = qw(nameserver1 nameserver2 nameserver3); # имена серверов имен
foreach $server (@servers) {
    &lookupaddress($hostname,$server);        # заполняем %results
}
%inv = reverse %results;                      # инвертируем полученный хэш
if (scalar(keys %inv) > 1) {
    print "Между DNS-серверами есть разногласия:\n";
    print Data::Dumper->Dump([\%results],["results"]),"\n";
}

# обращаемся к серверу, чтобы получить IP-адрес и прочую
# информацию для имени узла, переданного в программу в
# командной строке. Результаты записываем в хэш %results
sub lookupaddress {
    my($hostname,$server) = @_;

    open(NSLOOK,"$nslookup $hostname $server|") or
      die "Невозможно запустить nslookup:$!\n";

    while (<NSLOOK>) {
        # игнорировать, пока не дойдем до "Name: "
       next until (/^Name:/);
        # следующая строка - это ответ Address:
       chomp($results{$server} = <NSLOOK>);
        # удаляем имя поля
```

```
            die "Ошибка вывода nslookup \n" unless /Address/;
            $results{$server} =~ s/Address(es)?:\s+//;
            # все, с nslookup мы закончили
            last;
        }
        close(NSLOOK);
    }
```

Преимущества такого подхода:

- Это короткая программа, которую можно быстро написать (вероятно, ее даже можно построчно перевести из настоящего сценария командного интерпретатора).
- Нет необходимости писать запутанный код для работы с сетью.
- Применяется подход в стиле Unix, когда язык общего назначения используется для соединения нескольких маленьких специальных программ, выполняющих требуемые задачи в связке, вместо того чтобы писать большую монолитную программу.
- Это может оказаться единственным выходом, если нельзя написать программу для взаимодействия клиента и сервера на Perl; в частности, если вам нужно обратиться к серверу, который требует использования особого клиента без каких-либо альтернатив.

Недостатки такого подхода:

- Появляется зависимость от другой программы за пределами сценария. А если эта программа недоступна? Что делать, если изменится формат вывода данной программы?
- Это работает медленнее. Каждый раз, для того чтобы выполнить запрос, необходимо запустить новый процесс. От подобной нагрузки можно избавиться, если установить двунаправленный канал с процессом *nslookup*, который открыт все то время, когда он нужен. Правда, это потребует несколько большего опыта программирования, но это стоит сделать, если вы решите использовать и улучшать подобный подход.
- У вас меньше контроля. Во всех деталях реализации приходится полагаться на милость внешней программы. Например, в данном случае *nslookup* (если быть более точным, то библиотека разыменования, которую использует *nslookup*) обрабатывает тайм-ауты сервера, повторные попытки запросов и дописывает списки поисков доменов.

Работа напрямую с сетевыми сокетами

Если вы «продвинутый системный администратор», вы можете решить, что вызывать внешнюю программу не следует. Вы можете захотеть реализовать запросы к DNS, не используя ничего, кроме Perl. Это означает, что нужно будет создавать вручную сетевые пакеты, переда-

вать их по сети и затем анализировать результаты, получаемые от сервера.

Вероятно, это самый сложный пример из всех, приведенных в книге. Написан он после обращения к дополнительным источникам информации, в которых можно найти несколько примеров существующего кода (включая модуль Майкла Фура (Michael Fuhr), показанный в следующем разделе). Вот что происходит на самом деле. Запрос к DNS-серверу состоит из создания специального сетевого пакета с определенным заголовком и содержимым, отправки его на DNS-сервер, получения ответа от сервера и его анализа.[1]

Каждый DNS-пакет (из тех, которые нас интересуют) может иметь до пяти различных разделов:

Header (Заголовок)

Содержит флаги и счетчики, относящиеся к запросу или ответу (присутствует всегда).

Question (Запрос)

Содержит вопрос к серверу (присутствует в запросе и повторяется при ответе).

Answer (Ответ)

Содержит все данные для ответа на DNS-запрос (присутствует в пакете DNS-ответа).

Authority (Полномочия)

Содержит информацию о том, можно ли получать авторитетные ответы.

Additional (Дополнительно)

Содержит любую информацию, которую вернет сервер помимо прямого ответа на вопрос.

Наша программа имеет дело только с первыми тремя из этих разделов. Для создания необходимой структуры данных для заголовка DNS-пакета и его содержимого используется набор команд `pack()`. Эти структуры данных передаются модулю `IO::Socket`, который посылает их в виде пакета. Этот же модуль получает ответ и возвращает его для обработки (при помощи `unpack()`). Умозрительно такой процесс не очень сложен.

Но перед тем как посмотреть на саму программу, нужно сказать об одной особенности в этом процессе. В RFC1035 (Раздел 4.1.4) определяются два способа представления доменных имен в DNS-пакетах: несжатые и сжатые. Под несжатым доменным именем подразумевается полное имя домена (например *host.oog.org*) в пакете. Этот способ ничем

[1] За подробной информацией советую обратиться к разделу «Сообщения» из RFC1035.

не примечателен. Но если это же доменное имя встретится в пакете еще несколько раз, то, скорее всего, оно будет представлено в сжатом виде во всех вхождениях, кроме первого. В сжатом представлении информация (или ее часть) о домене заменяется двубайтовым указателем на несжатое представление этого же доменного имени. Это позволяет использовать в пакете *host1*, *host2* и *host3* в *longsubdomain.longsubdomain.oog.org*, вместо того чтобы каждый раз включать лишние байты для *longsubdomain.longsubdomain.oog.org*. Нам необходимо обработать оба представления, поэтому и существует подпрограмма `&decompress`. Дальше обойдемся без фанфар и взглянем на код:

```
use IO::Socket;
$hostname = $ARGV[0];
$defdomain = ".oog.org"; # домен по умолчанию

@servers = qw(nameserver1 nameserver2 nameserver3); # имена серверов имен
foreach $server (@servers) {
    &lookupaddress($hostname,$server);              # записываем значения в
%results
}
%inv = reverse %results;         # инвертируем полученный хэш
if (scalar(keys %inv) > 1) {     # проверяем, сколько в нем элементов
    print "Между DNS-серверами есть разногласия:\n";
    use Data::Dumper;
    print Data::Dumper->Dump([\%results],["results"]),"\n";
}

sub lookupaddress{
    my($hostname,$server) = @_;

    my($qname,$rname,$header,$question,$lformat,@labels,$count);
    local($position,$buf);

    ###
    ### Конструируем заголовок пакета
    ###
    $header = pack("n C2 n4",
         ++$id,  # идентификатор запроса
         1,  # поля qr, opcode, aa, tc, rd (установлено только rd)
         0,  # rd, ra
         1,  # один вопрос (qdcount)
         0,  # нет ответов (ancount)
         0,  # нет записей ns в разделе authority (nscount)
         0); # нет rr addt1 (arcount)

    # если в имени узла нет разделителей,
    # дописываем домен по умолчанию
    if (index($hostname,'.') == -1) {
       $hostname .= $defdomain;
    }
```

```
# конструируем раздел qname пакета (требуемое доменное имя)
for (split(/\./,$hostname)) {
   $lformat .= "C a* ";
   $labels[$count++]=length;
   $labels[$count++]=$_;
}

###
### конструируем вопрос
###
$question = pack($lformat."C n2",
       @labels,
           0,  # конец меток
           1,  # qtype A
           1); # qclass IN

###
### посылаем пакет серверу и читаем ответ
###
$sock = new IO::Socket::INET(PeerAddr => $server,
                             PeerPort => "domain",
                             Proto    => "udp");

$sock->send($header.$question);
# используется UDP, так что максимальный размер пакета известен
$sock->recv($buf,512);
close($sock);

# узнаем размер ответа, так как мы собираемся отслеживать
# позицию в пакете при его анализе (через $position)
$respsize = length($buf);

###
### распаковываем раздел заголовка
###
($id,
 $qr_opcode_aa_tc_rd,
 $rd_ra,
 $qdcount,
 $ancount,
 $nscount,
 $arcount) = unpack("n C2 n4",$buf);

if (!$ancount) {
   warn "Невозможно получить информацию для $hostname с $server!\n";
   return;
}

###
### распаковываем раздел вопроса
```

```
    ###
    # раздел вопроса начинается после 12 байтов
    ($position,$qname) = &decompress(12);
    ($qtype,$qclass)=unpack('@'.$position.'n2',$buf);
    # переходим к концу вопроса
    $position += 4;

    ###
    ### распаковываем все записи о ресурсах
    ###
    for ( ;$ancount;$ancount--){
       ($position,$rname) = &decompress($position);
       ($rtype,$rclass,$rttl,$rdlength)=
             unpack('@'.$position.'n2 N n',$buf);
       $position +=10;
        # следующую строку можно изменить и использовать более
        # сложную структуру данных; сейчас мы подбираем
        # последнюю возвращенную запись rr
        $results{$server}=
           join('.',unpack('@'.$position.'C'.$rdlength,$buf));
       $position +=$rdlength;
    }
}

# обрабатываем информацию, "сжатую" в соответствии с RFC1035
# мы переходим в первую позицию в пакете и возвращаем
# найденное там имя (после того как разберемся с указателем
# сжатого формата) и место, которое мы оставили в конце
# найденного имени
sub decompress {
    my($start) = $_[0];
    my($domain,$i,$lenoct);

    for ($i=$start;$i<=$respsize;) {
       $lenoct=unpack('@'.$i.'C', $buf); # длина метки

       if (!$lenoct){        # 0 означает, что этот раздел обработан
           $i++;
           last;
       }

       if ($lenoct == 192) { # встретили указатель,
                             # следовательно, выполняем рекурсию
           $domain.=(&decompress((unpack('@'.$i.'n',$buf) & 1023)))[1];
           $i+=2;
           last
       }
       else {                # в противном случае это простая метка
           $domain.=unpack('@'.++$i.'a'.$lenoct,$buf).'.';
           $i += $lenoct;
       }
```

```
        }
        return($i,$domain);
    }
```

Надо заметить, что эта программа не является точным эквивалентом предыдущего примера, потому что мы не пытаемся эмулировать все нюансы поведения *nslookup* (тайм-ауты, повторные попытки и списки поиска). Рассматривая все три подхода, представленные здесь, обязательно обратите внимание на такие различия.

Преимущества этого подхода заключаются в следующем:

- Он не зависит от каких-либо других программ. Нет необходимости разбираться в работе других людей.
- Это настолько же быстро, а может быть, и еще быстрее, чем вызов внешней программы.
- Проще обработать параметры ситуации (тайм-ауты и прочее).

Недостатки же такого подхода в том, что:

- Для написания подобной программы понадобится больше времени и, кроме того, она сложнее предыдущей.
- Этот подход требует дополнительных знаний, не имеющих прямого отношения к вашей задаче (т. е. вам, возможно, потребуется узнать, как вручную собирать DNS-пакеты, чего при использовании *nslookup* знать было не нужно).
- Вам, вероятно, придется самостоятельно справиться с различиями между операционными системами (в предыдущем подходе они были скрыты благодаря тому, что эту работу выполнил автор внешней программы).

Использование Net::DNS

Как уже говорилось в главе 1, одна из сильных сторон Perl заключается в поддержке обширным сообществом разработчиков, создающих программы, которые могут применяться другими людьми. Если необходимо сделать на Perl нечто, на ваш взгляд, универсальное, то высока вероятность того, что кто-то уже написал модуль для работы с подобной проблемой. В данном случае можно воспользоваться отличным модулем `Net::DNS` Майкла Фура (Michael Fuhr), который упростит работу. Чтобы справиться с нашей задачей, необходимо создать новый объект, передать ему имя DNS-сервера, к которому следует обратиться, указать, что нужно послать запрос, и затем применить имеющиеся методы для анализа ответов:

```
use Net::DNS;

@servers = qw(nameserver1 nameserver2 nameserver3); # имена серверов имен
foreach $server (@servers) {
```

```
    &lookupaddress($hostname,$server);              # заполняем значениями
%results
}
%inv = reverse %results;         # инвертируем полученный хэш
if (scalar(keys %inv) > 1) {   # проверяем, сколько в нем элементов
    print "Между DNS-серверами есть разногласия:\n";
    use Data::Dumper;
    print Data::Dumper->Dump([\%results],["results"]),"\n";
}

# всего лишь несколько измененный пример из страниц руководства по Net::DNS
sub lookupaddress{
    my($hostname,$server) = @_;

    $res = new Net::DNS::Resolver;

    $res->nameservers($server);

    $packet = $res->query($hostname);

    if (!$packet) {
       warn "Невозможно получить данные о $hostname с $server!\n";
       return;
    }
    # сохраняем последний полученный ответ RR
    foreach $rr ($packet->answer) {
       $results{$server}=$rr->address;
    }
}
```

Преимущества такого подхода:

- Помимо прочего, получаемый код легко читать.
- Написать его можно быстрее.
- В зависимости от того, как реализован применяемый модуль (только на Perl или с использованием библиотечных вызовов из C или C++), написанная программа может выполняться так же быстро, как и вызов внешней программы.
- Потенциально, это переносимая программа – все зависит только от того, что именно сделал автор модуля. Везде, где можно установить модуль, программа будет работать.
- Как и в первом рассмотренном случае, написать программу можно быстро и просто, если кто-то другой сделает за вас всю работу, происходящую «за сценой». Вам не нужно знать, как работает модуль; вы только должны знать, как его применять.
- Код используется повторно. Нет необходимости каждый раз изобретать велосипед.

Недостатки данного подхода:

- Снова появилась зависимость. На этот раз необходимо убедиться, что модуль доступен вашей программе. Приходится поверить, что автор модуля проделал хорошую работу.
- Может не существовать подходящего вам модуля или он может не запуститься на выбранной вами операционной системе.

В большинстве случаев я предпочитаю использовать уже существующие модули. Тем не менее, для выполнения поставленной задачи подойдет любой подход. Существует несколько способов сделать одно и то же – значит, вперед, действуйте!

Информация о модулях из этой главы

Модуль	Идентификатор на CPAN	Версия
`Rcs`	CFRETER	0.09
`Net::NIS`	RIK	a2
`Data::Dumper` (входит в состав Perl)	GSAR	2.101
`IO::Socket` (входит в состав Perl)	GBARR	1.20
`Net::DNS`	MFUHR	0.12

Рекомендуемая дополнительная литература

«DNS and BIND», 3rd Edition, Paul Albitz, Cricket Liu (O'Reilly, 1998).

«RFC849: Suggestions For Improved Host Table Distribution», Mark Crispin, 1983.

«RFC881: The Domain Names Plan and Schedule», J. Postel, 1983.

«RFC882: Domain Names: Concepts And Facilities», P. Mockapetris, 1983.

«RFC1035: Domain Names: Implementation And Specification», P. Mockapetris, 1987.

6

- *Что такое каталог?*
- *Finger: простая служба каталогов*
- *Служба каталогов WHOIS*
- *LDAP: сложная служба каталогов*
- *ADSI (Интерфейсы служб активных каталогов)*
- *Информация о модулях из этой главы*
- *Рекомендуемая дополнительная литература*

Службы каталогов

Чем больше информационная система, тем сложнее найти в ней что-либо конкретное или выяснить, что именно в ней доступно. Когда сети разрастаются и увеличивается их сложность, обслуживать их удобно при помощи тех или иных каталогов. Пользователи в сетях могут применять службы каталогов для поиска других пользователей, для почтовых служб и иных служб сообщений. Такие сетевые ресурсы, как принтеры или доступные по сети дисковые пространства, могут быть объявлены при помощи службы каталогов. Средствами служб каталогов также можно распространять открытые ключи и сертификаты. В этой главе мы рассмотрим, как использовать Perl для взаимодействия с некоторыми из наиболее популярных служб каталогов, включая Finger, WHOIS, LDAP и ADSI.

Что такое каталог?

В главе 7 «Администрирование баз данных SQL» мною сделано предположение, согласно которому мир системного администрирования представляет собой базу данных. Каталоги – хороший пример такой оценки. Отметим некоторые явные характеристики каталогов, чтобы в дальнейшем разговоре различать «базы данных« и «каталоги»:

Использование сети

Каталоги практически всегда связаны сетью. В отличие от некоторых баз данных, расположенных на той же машине, что и их клиенты (как хорошо известный файл */etc/passwd*), службы каталогов обычно предоставляются по сетям.

Простое взаимодействие/манипуляция данными

Базы данных зачастую имеют сложные языки запросов для получения и обработки данных. Самый распространенный из них – SQL, ему уделено внимание в главе 7 и приложении D «Пятнадцатиминутное руководство по SQL». Взаимодействие с каталогами значительно проще. Клиент каталогов обычно выполняет только элементарные операции и не использует для работы с сервером никакого специального языка.

Иерархичность

Современные службы каталогов поддерживают древоподобные информационные структуры, в то время как базы данных в целом – нет.

Читаем много, пишем мало

Современные службы каталогов оптимизированы под очень специфичную передачу данных. При обычном использовании количество операций чтения/запросов к службе каталогов во много раз превосходит количество операций записи/обновлений.

Если вам встретится нечто, похожее на базу данных, но обладающее приведенными выше характеристиками, то, скорее всего, вы имеете дело с каталогом. Во всех четырех службах каталогов, которые мы рассмотрим, эти характеристики несложно заметить.

Finger: простая служба каталогов

Finger и WHOIS – отличные примеры простых служб каталогов. Finger, в особенности, предоставляет доступную только для чтения информацию о пользователях на машине (впрочем, скоро мы увидим более творческий подход к его применению). Более поздние версии Finger, такие как сервер GNU Finger и его производные, имеют расширенную разновидность этой функциональности, они позволяют обращаться к одной машине и получать информацию от всех машин из вашей сети.

Finger был одной из первых широко используемых служб каталогов. Когда-то очень давно при необходимости выяснить адрес электронной почты пользователя на другом узле или даже на вашем собственном лучшим решением было применение команды *finger*. Команда *finger harry@hogwarts.edu* сообщала адрес электронной почты Гарри, будь он `harry`, `hpotter` или даже что-то менее очевидное (правда, эта команда выводила список всех остальных учащихся школы с именем Гарри). И хотя *finger* применяется до сих пор, популярность команды со временем уменьшилась, т. к. вошли в обиход домашние страницы и свободное получение информации о пользователе стало делом проблематичным.

Использование протокола Finger из Perl – еще один хороший пример правила TMTOWTDI. Когда я первый раз искал на CPAN хоть что-то, выполняющее операции с Finger, мне не удалось найти ни одного такого модуля. Сейчас-то там можно натолкнуться на модуль `Net::Finger` Дениса Тейлора (Dennis Taylor), который появился спустя примерно шесть месяцев после моего первого посещения. Скоро вы с ним познакомитесь, а пока предположим, что его не существует, и не упустим случая выяснить, как применять более общий модуль, позволяющий «связываться» по специфическому протоколу.

Протокол Finger – это очень простой текстовый протокол на базе TCP/IP. В соответствии с определением в RFC1288, он ожидает стандартное TCP-соединение на порту 79. После установки соединения клиент передает простую строку, завершающуюся последовательностью CRLF.[1] В этой строке запрашивается информация либо о конкретном пользователе, либо обо всех пользователях на машине, если строка пустая. Сервер отвечает на запрос и закрывает соединение после передачи потока данных. Можно увидеть, как это происходит, если подсоединиться к порту Finger на удаленной машине напрямую при помощи *telnet*:

```
$ telnet kantine.diku.dk 79
Trying 192.38.109.142 ...
Connected to kantine.diku.dk.
Escape character is '^]'.
cola<CR><LF>
Login: cola                             Name: RHS Linux User
Directory: /home/cola                   Shell: /bin/noshell
Never logged in.
No mail.
Plan:

Current state of the coke machine at DIKU
This file is updated every 5 seconds
At the moment, it's necessary to use correct change.
This has been the case the last 19 hours and 17 minutes

Column 1 is currently *empty*.
   It's been 14 hours and 59 minutes since it became empty.
   31 items were sold from this column before it became empty.
Column 2 contains some cokes.
   It's been 2 days, 17 hours, and 43 minutes since it was filled.
   Meanwhile, 30 items have been sold from this column.
Column 3 contains some cokes.
   It's been 2 days, 17 hours, and 41 minutes since it was filled.
   Meanwhile, 11 items have been sold from this column.
Column 4 contains some cokes.
   It's been 5 days, 15 hours, and 28 minutes since it was filled.
```

[1] Возврат каретки + перевод строки, т. е. символы с ASCII-кодами 13 и10.

```
    Meanwhile, 26 items have been sold from this column.
Column 5 contains some cokes.
    It's been 5 days, 15 hours, and 29 minutes since it was filled.
    Meanwhile, 18 items have been sold from this column.
Column 6 contains some coke-lights.
    It's been 5 days, 15 hours, and 30 minutes since it was filled.
    Meanwhile, 16 items have been sold from this column.

Connection closed by foreign host.
$
```

В данном примере мы напрямую соединились с портом Finger, набрали имя пользователя «cola», и сервер вернул требуемую информацию.

Я выбрал этот узел и этого пользователя только затем, чтобы показать, какие чудеса творились на заре появления Интернета. Серверы Finger превратились в службы для задач различного вида. В этом случае кто угодно может посмотреть, заполнен ли автомат газированных напитков, расположенный на факультете компьютерных наук в университете Копенгагена. Примеры странных устройств, подключенных к серверам Finger, можно найти на страницах Беннета Йи (Bennet Yee) в «Internet Accessible Coke Machines» и «Internet Accessible Machines» на *http://www.cs.ucsd.edu/~bsy/fun.html*.

Перенесем сетевое соединение, установленное с помощью *telnet*, в мир Perl. Средствами Perl можно открыть сокет и общаться через него. Вместо того чтобы применять низкоуровневые команды сокета, воспользуемся модулем `Net::Telnet` Джея Роджера (Jay Roger) и познакомимся с семейством модулей, работающих с сетевыми соединениями. Другие модули этого семейства (некоторые из них будут применяться в иных главах) включают *Comm.pl* Эрика Арнольда (Eric Arnold), *Expect.pm* Остина Шатца (Austin Schutz) и хорошо известный, но устаревший и непереносимый модуль *chat2.pl* Рэндала Л. Шварца (Randal L. Schwartz).

`Net::Telnet` устанавливает соединение и обеспечивает четкий интерфейс для отправки и получения данных через это соединение. Кроме того, `Net::Telnet` предоставляет удобные механизмы сканирования шаблонов, позволяющие программам наблюдать за определенными ответами от другого сервера, но в примере подобные свойства использоваться не будут.

Вот `Net::Telnet`-версия простого Finger-клиента. Эта программа принимает аргумент в виде *user@finger_server*. Если имя пользователя пропущено, будет возвращен список всех активных пользователей с сервера. Если пропущено имя узла, будет запрашиваться локальный узел:

```
use Net::Telnet;

($username,$host) = split(/\@/,$ARGV[0]);
```

```
$host = $host ? $host : 'localhost';

# создаем новое соединение
$cn = new Net::Telnet(Host => $host,
                      Port => 'finger');

# посылаем имя пользователя
unless ($cn->print("$username")){ # может быть "/W $username"
    $cn->close;
    die "Невозможно отправить строку: ".$cn->errmg."\n";
}

# собираем все полученные данные, останавливаясь при завершении соединения
while (defined ($ret = $cn->get)[1]) {
    $data .= $ret;
}

# закрываем соединение
$cn->close;

# отображаем полученные данные
print $data;
```

В RFC1288 определено, что перед именем пользователя, отправляемым на сервер, можно добавить ключ /W для «вывода подробной информации о пользователе», поэтому в программу добавлен комментарий об этом ключе.

Если нужно соединиться, используя помимо Finger другой текстовый протокол на основе TCP, можно применить очень похожую программу. Например, для соединения с сервером Daytime (который выводит локальное время) используется очень похожая программа:

```
use Net::Telnet;

$host = $ARGV[0] ? $ARGV[0] : 'localhost';

$cn = new Net::Telnet(Host => $host,
                      Port => 'daytime');

while (defined ($ret = $cn->get)[2]) {
    $data .= $ret;
}
$cn->close;
```

[1] Скобки поставлены в соответствии с требованиями Perl 5.6.1. В тексте оригинала скобки отсутствуют. – *Примеч. науч. ред.*

[2] Скобки поставлены в соответствии с требованиями Perl 5.6.1. В тексте оригинала скобки отсутствуют. – *Примеч. науч. ред.*

```
print $data;
```

Теперь читатель имеет представление о том, насколько легко создавать типовые сетевые клиенты на основе TCP. Если кто-то уже потратил время и написал модуль, специально созданный для работы с протоколом, все окажется еще проще. В случае с Finger можно воспользоваться модулем `Net::Finger` и заменить все вызовом одной функции:

```
use Net::Finger;

# finger() принимает строку user@host и возвращает полученные данные
print finger($ARGV[0]);
```

Желая показать все варианты, упомянем о возможности вызвать внешнюю программу (если она существует):

```
($username,$host) = split('@',$ARGV[0]);
$host = $host ? $host : 'localhost';

# местоположение команды finger executable, пользователи MacOS
# этим методом воспользоваться не могут
$fingerex = ($^O eq "MSWin32") ?
                $ENV{'SYSTEMROOT'}."\\System32\\finger" :
                "/usr/ucb/finger";  # (также может быть и /usr/bin/finger)

print `$fingerex ${username}\@${host}`
```

Вы познакомились с тремя различными способами выполнения Finger-запросов. Третий метод, вероятно, самый неудачный, т. к. в нем порождается другой процесс. `Net::Finger` обрабатывает простые Finger-запросы; все остальное может очень хорошо выполнить `Net::Telnet` или родственные ему модули.

Служба каталогов WHOIS

WHOIS – это еще одна полезная служба каталогов, предоставляющая доступную только для чтения информацию. WHOIS обеспечивает услуги, подобные телефонному справочнику для машин, сетей и людей. Некоторые крупные организации, такие как IBM, UC Berkeley и MIT, предоставляют услуги WHOIS, но самые известные WHOIS-серверы принадлежат InterNIC и другим компаниям, занимающимся вопросами регистрации в Интернете, в том числе RIPE (имеет дело с европейскими IP-адресами) и APNIC (Asia/Pacific, Азиатские/Тихоокеанские адреса).

При необходимости связаться с системным администратором какого-либо узла, чтобы сообщить ему о подозрительных действиях в сетях, следует использовать службу WHOIS для получения контактной информации. В большинстве операционных систем для выполнения

WHOIS-запросов существуют как GUI-инструменты, так и инструменты, запускаемые из командной строки. Типичный запрос в Unix выглядит так:

```
% whois -h whois.networksolutions.com brandeis.edu
<large legal paragraph omitted>
Registrant:
Brandeis University (BRANDEIS-DOM)
   Information Technology Services
   Waltham, MA 02454-9110
   US

   Domain Name: BRANDEIS.EDU

   Administrative Contact:
      Koskovich, Bob  (BK138)  user@BRANDEIS.EDU
      +1-781-555-1212 (FAX) +1-781-555-1212
   Technical Contact, Zone Contact:
      Hostmaster, Brandeis C  (RCG51)  hostmaster@BRANDEIS.EDU
      +1-781-555-1212 (FAX) +1-781-555-1212
   Billing Contact:
      Koskovich, Bob  (BK138)  user@BRANDEIS.EDU
      +1-781-555-1212 (FAX) +1-781-555-1212

   Record last updated on 13-Oct-1999.
   Record created on 27-May-1987.
   Database last updated on 19-Dec-1999 17:42:19 EST.

   Domain servers in listed order:

   LILITH.UNET.BRANDEIS.EDU     129.64.99.12
   FRASIER.UNET.BRANDEIS.EDU    129.64.99.11
   DIAMOND.CS.BRANDEIS.EDU      129.64.2.3
   DNSAUTH1.SYS.GTEI.NET        4.2.49.2
   DNSAUTH2.SYS.GTEI.NET        4.2.49.3
```

Если же нужно выяснить владельца определенного диапазона IP-адресов, то и тут поможет WHOIS:

```
% whois -h whois.arin.net 129.64.2
Brandeis University (NET-BRANDEIS)
   415 South Street
   Waltham, MA 02254

   Netname: BRANDEIS
   Netnumber: 129.64.0.0

   Coordinator:
      Koskovich, Bob  (BK138-ARIN)  user@BRANDEIS.EDU
      617-555-1212
```

```
   Domain System inverse mapping provided by:

   BINAH.CC.BRANDEIS.EDU        129.64.1.3
   NIC.NEAR.NET                 192.52.71.4
   NOC.CERF.NET                 192.153.156.22

   Record last updated on 10-Jul-97.
   Database last updated on 9-Oct-98 16:10:44 EDT.

The ARIN Registration Services Host contains ONLY Internet
Network Information: Networks, ASN's, and related POC's.
Please use the whois server at rs.internic.net for DOMAIN related
Information and nic.mil for NIPRNET Information.
```

В предыдущем примере применялся WHOIS-клиент из Unix, работающий в командной строке. В Windows NT и MacOS подобные клиенты не входят, тем не менее, это не должно остановить пользователей данных систем от получения доступа к нужной информации. Существует много условно бесплатных клиентов, но не так трудно с помощью модуля `Net::Whois` создать на Perl очень простой клиент (модуль `Net::Whois` первоначально был написан Чипом Салзенбергом (Chip Salzenberg), а теперь поддерживается Даной Хьюдес (Dana Hudes)). Следующий код – это лишь несколько измененная версия примера из документации, поставляемой вместе с модулем:

```
use Net::Whois;

# запрашиваем сервер, возвращая объект с результатами
my $w = new Net::Whois::Domain $ARGV[0] or
   die "Невозможно соединиться с сервером Whois\n";
die "Никакой информации о домене $ARGV[0] не найдено\n " unless ($w->ok);

# выводим части этого объекта
print "Домен: ", $w->domain, "\n";
print "Имя: ", $w->name, "\n";
print "Тег: ", $w->tag, "\n";
print "Адрес:\n", map { "    $_\n" } $w->address;
print "Страна: ", $w->country, "\n";
print "Запись создана: ".$w->record_created."\n";
print "Запись обновлена: ".$w->record_updated."\n";

# выводим серверы имен ($w->servers returns a list of lists)
print "Серверы имен:\n", map { "    $$_[0] ($$_[1])\n" } @{$w->servers};

# выводим список контактов ($w->contacts returns a hash of lists)
my($c,$t);
if ($c = $w->contacts) {
    print "Contacts:\n";
    for $t (sort keys %$c) {
        print "    $t:\n";
```

```
            print map { "\t$_\n" } @{$$c{$t}};
        }
    }
```

Запрос WHOIS сервера InterNIC/Network Solutions – это простой процесс. Для возвращения результата применяется мо«дуль `Net::Whois::Domain`. Методы этого объекта, названные в соответствии с полями, которые получает WHOIS-запрос, обеспечивают доступ к данным.

WHOIS предстоит сыграть значительную роль в главе 8 «Электронная почта», а сейчас перейдем к более сложным службам каталогов. Мы уже начали этот переход, переключаясь со службы Finger на WHOIS. Между рассмотренными способами использования Finger и WHOIS существует важное различие – структура.

Вывод Finger отличается от реализации к реализации. И хотя существуют некоторые соглашения, форму он имеет свободную. WHOIS-сервер InterNIC/Network Solutions возвращает данные более постоянной структуры. Можно рассчитывать на то, что у каждой записи будут, по крайней мере, поля `Name`, `Address` и `Domain`. Модуль `Net::Whois` полагается на эту структуру и анализирует результаты, разбивая их на поля. Существует еще один модуль Випула Вед Пракаша (Vipul Ved Prakash) – `Net::Xwhois`, который делает шаг вперед, обеспечивая интерфейс для анализа информации, по-разному отформатированной различными WHOIS-серверами.

И хотя в протоколе WHOIS нет никакого упоминания о полях, вызываемые нами модули начинают полагаться на структуру информации. Службы каталогов, о которых пойдет речь, более серьезно относятся к этой структуре.

LDAP: сложная служба каталогов

Службы LDAP (Lightweight Directory Access Protocol, облегченный протокол доступа к каталогам) и ADSI гораздо богаче и более сложны в обращении. В настоящее время существуют две популярные версии протокола LDAP (версия 2 и версия 3; при необходимости номер версии будет указываться). Этот протокол быстро стал промышленным стандартом для доступа к каталогам. Системные администраторы воспользовались протоколом LDAP, т. к. он предлагал способ централизовать и сделать доступной всю информацию об инфраструктуре. Помимо стандартного «каталога компании» существуют такие примеры приложений:

- Шлюзы NIS-к-LDAP
- Шлюзы Finger-к-LDAP
- Всевозможные базы данных аутентификации (в частности, для использования в Сети)

- Объявления о ресурсах (какие машины и периферийные устройства доступны)

Кроме того, LDAP является базой для других сложных служб каталогов, подобных активным каталогам Microsoft (Microsoft Active Directory), о которых пойдет речь в разделе «ADSI (Интерфейсы служб активных каталогов)».

Даже в случае, когда LDAP применяется только для ведения «домашней» телефонной книги, существуют веские причины научиться использовать этот протокол. LDAP-серверы можно администрировать при помощи этого же протокола; что очень напоминает серверы баз данных SQL, которые тоже можно администрировать средствами SQL. И в этом случае Perl предлагает отличные механизмы склейки для автоматизации задач администрирования LDAP. Но сначала нужно убедиться, что протокол LDAP разобран и понятен.

В приложении B «Десятиминутное руководство по LDAP» приводится краткое введение в LDAP для тех, кто не знаком с протоколом. Самая большая преграда, встающая перед системным администратором при изучении LDAP, – это неуклюжая терминология, унаследованная от родительских протоколов службы каталогов X.500. LDAP является упрощенной версией X.500, но, к сожалению, терминология от этого легче не стала. Стоит потратить время на приложение B и изучение терминов – тогда вам будет проще понять, как использовать LDAP из Perl.

Программирование LDAP на Perl

Как и с многими другими задачами системного администрирования в Perl, первым делом при программировании LDAP следует выбрать нужный модуль. Хотя LDAP еще не самый сложный протокол, но это уже и не обычный текстовый протокол. В результате, создать что-нибудь, «говорящее» на LDAP, задача нетривиальная. К счастью, другие авторы уже сделали эту работу за нас: Грэм Бар (Graham Barr) написал модуль `Net::LDAP`, а Лейф Хедстром (Leif Hedstrom) и Клейтон Донли (Clayton Donley) создали модуль `Mozilla::LDAP` (также известный как `PerLDAP`). Отметим некоторые различия между этими двумя модулями (табл. 6.1).

Таблица 6.1. Сравнение двух LDAP-модулей

Возможность	Net::LDAP	Mozilla::LDAP (PerLDAP)
Переносимость	Только Perl	Требует Mozilla/Netscape LDAP C-SDK (исходный код свободно доступен). SDK компилируется на многих вариантах Unix, NT и MacOS
Зашифрованные SSL-сеансы	Да	Да
Асинхронные операции	Да	Только с не объектно-ориентированными API низкого уровня

Оба модуля обладают функциональностью, необходимой для выполнения обсуждаемых ниже простых задач, связанных с системным администрированием, но предоставляют ее немного различными способами. Это создает, с точки зрения обучения, редкую возможность увидеть, как два разных автора реализовали важные модули, применимые в одной и той же области. Скрупулезное сравнение обоих модулей поможет разобраться с процессом их содания, что и будет показано в главе 10 «Безопасность и наблюдение за сетью». Для облегчения сравнения в большей части примеров из этого раздела приведен синтаксис обоих LDAP-модулей. Строка `use modulename` в тексте каждого примера подскажет, какой модуль используется на этот раз.

В демонстрационных целях мы почти равнозначно будем использовать коммерческий сервер Netscape 4.0 Directory Server и свободно распространяемый сервер OpenLDAP (они находятся на *http://www.netscape.com* и *http:// www.openldap.org*). В состав обоих серверов входят практически идентичные утилиты командной строки, которые можно использовать для прототипирования и проверки программ на Perl.

Первоначальное LDAP-соединение

Соединение с аутентификацией – это, обычно, первый шаг в любой клиент-серверной LDAP-транзакции. На «языке» LDAP это называется «связыванием с сервером» (binding to the server). В LDAPv2 требовалось связаться с сервером перед отправкой команд, в LDAPv3 условия не такие жесткие.

Связывание с LDAP-сервером выполняется в контексте определенного отличительного имени (Distinguished name, DN), описанного как *привязанное отличительное имя* (*bind DN*) для данного сеанса. Такой контекст похож на регистрацию пользователя в многопользовательской системе. В многопользовательской системе текущее регистрационное имя (по большей части) определяет уровень доступа пользователя к данным в этой системе. В LDAP именно привязанное отличительное имя определяет, какие данные на LDAP-сервере доступны для просмотра и изменения. Существует также специальное *корневое отличительное имя* (*root Distinguished Name*), дабы не путать его с относительным (Relative Distinguished Name) именем, для которого не существует акронима. Корневое отличительное имя имеет полный контроль над всем деревом, что очень похоже на регистрацию с правами пользователя *root* в Unix или с правами пользователя *Administrator* в NT/2000. На некоторых серверах это имя называется *manager DN*.

Если клиент не предоставляет аутентификационной информации (например, DN-имя и пароль) во время связывания или вообще не связывается с сервером до оправки команд, это называется *анонимной аутентификацией* (*anonymous authentication*). Анонимно зарегистрированные клиенты обычно получают очень ограниченный доступ к данным на сервере.

В спецификации LDAPv3 определяется два типа связывания: простой и SASL. В простом связывании для аутентификации используются обычные текстовые пароли. SASL (Simple Authentication and Security Layer, слой простой аутентификации и безопасности) – это расширенный интерфейс аутентификации, определенный в RFC2222, позволяющий авторам клиентов/серверов встраивать различные схемы аутентификации, подобные Kerberos и одноразовым паролям. Когда клиент соединяется с сервером, он запрашивает определенный механизм аутентификации. Если сервер его поддерживает, он начнет диалог, соответствующий такому механизму, для аутентификации клиента. Во время этого диалога клиент и сервер могут договориться об уровне безопасности (например, «весь трафик между нами будет зашифрован при помощи TLS»), применяемом после завершения аутентификации.

Некоторые серверы и клиенты LDAP добавляют еще один метод аутентификации к SASL и стандартному простому способу. Этот метод является побочным продуктом использования LDAP по зашифрованным SSL (Secure Socket Layer, уровень защищенных сокетов). Для установки этого канала серверы и клиенты LDAP обмениваются криптографическими сертификатами на основе открытого ключа так же, как веб-сервер и броузеры при работе по протоколу HTTPS. LDAP-серверу можно дать указание использовать в качестве аутентификационной информации только надежный клиентский сертификат (trusted client's certificate). Из доступных Perl-модулей только PerLDAP предлагает LDAPS (зашифрованные SSL-соединения). В примерах, для того чтобы не слишком усложнять их, будем пользоваться только простой аутентификацией и незашифрованными соединениями.

Вот как выполняется простое соединение и его завершение средствами Perl:

```
use Mozilla::LDAP::Conn;
# используем пустые $binddn и $passwd для анонимной связи
$c = new Mozilla::LDAP::Conn($server, $port, $binddn, $passwd);
die "Невозможно соединиться с $server" unless $c;
...
$c->close();
```

или:

```
use Net::LDAP;
$c = Net::LDAP->new($server, port => $port) or
    die " Невозможно соединиться с $server: $@\n";
# не передаем параметры bind() для анонимной связи
$c->bind($binddn, password => $passwd) or
    die "Невозможно соединиться: $@\n";
...
$c->unbind();
```

В `Mozilla::LDAP::Conn` создание нового объекта соединения также связано с сервером. В `Net::LDAP` этот процесс состоит из двух шагов. Для инициализации соединения без выполнения привязывания в `Mozilla::LDAP` необходимо использовать функцию (`ldap_init()`) из не объектно-ориентированного модуля `Mozilla::LDAP::API`.

Приготовьтесь тщательно заключить в кавычки значения атрибутов

Небольшой совет перед тем, как перейти к дальнейшему программированию на Perl: если в относительном отличительном имени есть атрибут, значение которого содержит один из следующих символов: «+», «(пробел)», «,», «'», «>», «<» или «;», необходимо либо заключить значение в кавычки, либо экранировать эти символы обратным слэшем (\). Если значение содержит кавычки, их также нужно экранировать при помощи обратного слэша. Обратные слэши в значениях тоже экранируются обратными слэшами.

Если вы не будете аккуратны, то недостаток кавычек может сыграть с вами злую шутку.

Выполнение поиска в LDAP

Буква «D» в LDAP означает Directory (т. е. каталог), и наиболее распространенной операцией с каталогами является поиск. Для начала знакомства с LDAP неплохо выяснить, как искать информацию. Поиск в LDAP определяется такими понятиями:

Откуда начинать поиск

Это называется *базовым относительным именем* (base DN) или *базой поиска* (*search base*). Базовое DN-имя представляет собой всего лишь DN-имя элемента в дереве каталогов, от которого начинается поиск.

Где искать

Это называется *пространством* (*scope*) поиска. Пространство может быть трех видов: *base* (поиск только по базовому DN-имени), *one* (поиск по уровню, лежащему непосредственно под базовым DN-именем, не включая само базовое DN-имя) или *sub* (поиск по базовому DN-имени и всему дереву, лежащему ниже).

Что искать

Это называется *фильтрами поиска* (*search filter*). О фильтрах и их определении мы поговорим очень скоро.

Что возвращать

С целью ускорения операций поиска можно выбрать, какие атрибуты должны возвращаться для каждого элемента, найденного при помощи фильтров поиска. Кроме того, можно запросить, чтобы воз-

вращались только имена атрибутов, а не их значения. Это полезно, когда нужно узнать, у каких элементов есть данные атрибуты, но совсем не важно, что эти атрибуты содержат.

В Perl поиск выглядит примерно так (процесс соединения заменен многоточиями):

```
use Mozilla::LDAP::Conn;
...
$entry = $c->search($basedn, $scope, $filter);
die "Неуспешный поиск: ". $c->getErrorString()."\n" if $c->getErrorCode();
```

или:

```
use Net::LDAP;
...
$searchobj = $c->search(base => $basedn, scope => $scope,
                        filter => $filter);
die "Неуспешный поиск, номер ошибки #".$searchobj->code() if $searchobj-
>code();
```

Перед тем как перейти к полному примеру, поговорим о загадочном параметре `$filter`. Простые фильтры поиска имеют следующий вид:

```
<attribute name> <comparison operator> <attribute value>
```

где *<comparison operator>* определяется в RFC2254 как один из операторов, перечисленных в табл. 6.2.

Таблица 6.2. Операторы сравнения LDAP

Оператор	Значение
`=`	Точное совпадение значений. Может означать и частичное совпадение, если в определении *<attribute value>* используется * (например `cn=Tim O*`).
`=*`	Соответствует всем элементам, у которых есть значения для атрибута *<attribute name>*, независимо от того, каковы эти значения. Если вместо *<attribute value>* указать *, будет проверяться наличие именно этого атрибута в элементе (например, `cn=*` выберет элементы, у которых есть атрибуты `cn`).
`~=`	Приблизительное совпадение значений.
`>=`	Больше либо равно значению.
`<=`	Меньше либо равно значению.

Это очень похоже на Perl, но не заблуждайтесь. Две конструкции, которые могут смутить знатоков Perl, это `~=` и `=*`. Первая из них не имеет ничего общего с регулярными выражениями; она ищет приблизительное соответствие с указанным значением. В этом случае определение «приблизительное» зависит от сервера. Большинство серверов применяют алгоритм, первоначально используемый в `soundex` для определе-

ния совпадающих значений при поиске слов, которые «произносятся, как» заданное значение (в английском языке), но записываются иначе.[1]

Другая конструкция, которая может конфликтовать с вашими знаниями Perl, – это оператор =. Помимо проверки точного совпадения значений (как строковых, так и численных), оператор = можно использовать вместе с символом * в виде префикса или суффикса в качестве символов подстановки, подобно тому как это происходит в командных интерпретаторах. Например, `cn=a*` получит все элементы, имена которых (common name) начинаются с буквы «a». Строка `cn=*a*` выполнит именно то, чего вы ждете, и найдет все элементы, в атрибуте `cn` которых есть буква «a».

Можно объединить в одну строку два или более простых фильтра *<attribute name>*, *<comparison operator>*, *<attribute value>* при помощи логических операторов, создав таким образом более сложный фильтр. Он имеет следующий вид:

```
(<boolean operator> (<simple1>) (<simple2>) (<simple3>) ... )
```

Те, кто знаком с LISP, без труда разберутся с подобным синтаксисом; всем остальным придется просто запомнить, что оператор, объединяющий простые формы поиска, записывается первым. Чтобы найти элементы, удовлетворяющие обоим критериям поиска A *и* B, нужно использовать запись `(&(A)(B))`. Для элементов, удовлетворяющих критериям A *или* B *или* C, следует применить `(|(A)(B)(C))`. Восклицательный знак отрицает указанный критерий: так, A *и не* B записывается следующим образом: `(&(A)(!B))`. Составные фильтры можно объединять друг с другом, чтобы создавать фильтры поиска произвольной сложности. Вот пример составного фильтра для поиска всех Финкельштейнов, работающих в Бостоне:

```
(&(sn=Finkelstein)(l=Boston))
```

Следующий фильтр ищет человека, чья фамилия либо Финкельштейн, либо Хайндс:

```
(|(sn=Finkelstein)(sn=Hinds))
```

Для поиска всех Финкельштейнов, работающих не в Бостоне:

```
(&(sn=Finkelstein)(!(l=Boston)))
```

Для поиска всех Финкельштейнов или Хайндсов, работающих не в Бостоне:

```
(&(|(sn=Finkelstein)(sn=Hinds))(!l=Boston))
```

[1] Тот, кто захочет поэкспериментировать с алгоритмом soundex, может воспользоваться модулем Марка Милке (Mark Mielke) `Text::Soundex`.

Вот два примера программ, принимающих имя LDAP-сервера и фильтр и возвращающих результаты запроса:

```
use Mozilla::LDAP::Conn;

$server = $ARGV[0];
$port   = getservbyname("ldap","tcp") || "389";
$basedn = "c=US";
$scope  = "sub";

$c = new Mozilla::LDAP::Conn($server, $port, "","");  # анонимное соединение
die "Невозможно связаться с $server\n" unless $c;

$entry = $c->search($basedn, $scope, $ARGV[1]);
die "Ошибка поиска: ". $c->getErrorString()."\n" if $c->getErrorCode();

# обрабатываем полученные от search() значения
while ($entry) {
    $entry->printLDIF();
    $entry = $c->nextEntry();
}
$c->close();
```

```
use Net::LDAP;
use Net::LDAP::LDIF;

$server = $ARGV[0];
$port   = getservbyname("ldap","tcp") || "389";
$basedn = "c=US";
$scope  = "sub";

$c = new Net::LDAP($server, port=>$port) or
    die "Невозможно соединиться с $server: $@\n";
$c->bind() or die "Unable to bind: $@\n"; # анонимное соединение

$searchobj = $c->search(base => $basedn, scope => $scope,
                        filter => $ARGV[1]);
die " Неуспешный поиск, номер ошибки #".$searchobj->code() if $searchobj-
>code();

# обрабатываем полученные от search() значения
if ($searchobj){
    $ldif = new Net::LDAP::LDIF("-");
    $ldif->write($searchobj->entries());
    $ldif->done();
}
```

А вот отрывок из получаемых данных:

```
$ ldapsrch ldap.bigfoot.com '(sn=Pooh)'
...
```

```
dn: cn="bear pooh",mail=poohbear219@hotmail.com,c=US,o=hotmail.com
mail: poohbear219@hotmail.com
cn: bear pooh
o: hotmail.com
givenname: bear
surname: pooh
...
```

Перед тем как улучшить этот пример, посмотрим на код, обрабатывающий результаты, полученные от `search()`. Это одно из тех мест, где модули отличаются моделью программирования. Оба примера возвращают одну и ту же информацию в формате LDIF (LDAP Data Interchange Format, формат обмена данными LDAP), о котором речь пойдет позже, но данные они получают совершенно разными способами.

Модель `Mozilla::LDAP` остается справедливой для подпрограмм анализа поиска, описанных в спецификации C API в RFC1823. Если поиск был успешным, возвращается первый найденный элемент. Для просмотра результатов необходимо последовательно запросить следующие элементы. Вывод содержимого каждого получаемого элемента выполняет метод `printLDIF()`.

Модель программирования `Net::LDAP` имеет больше сходства с определением протокола из RFC2251. Результаты поиска LDAP возвращаются в объекты сообщений. Для получения списка элементов из этих пакетов в предыдущем примере использовался метод `entries()`. Вывод всех элементов вместе выполняет метод из смежного модуля `Net::LDAP::LDIF`. Для последовательного вывода всех элементов, как было с `printLDIF()` в первом примере, можно использовать похожий метод `write()`, но показанный выше вызов более эффективен.

Немного поработаем с предыдущим примером. Как уже отмечалось в данной главе, поиск можно выполнять быстрее, ограничив количество возвращаемых в результате атрибутов. С модулем `Mozilla::LDAP` это настолько же просто, насколько просто добавить дополнительные параметры в вызов метода `search()`:

```
use Mozilla::LDAP::Conn;
...
$entry = $c->search($basedn,$scope,$ARGV[1],0,@attr);
```

Первый дополнительный параметр – это логический флаг, определяющий, будут ли значения атрибутов опущены в результатах поиска. Значение по умолчанию – ложь (0), т. к. в большинстве случаев нас интересуют не только имена атрибутов.

Следующий дополнительный параметр – это список имен возвращаемых атрибутов. Знатоки Perl заметят, что список внутри списка интерполируется, так что последняя строка эквивалентна строке (ее можно так и прочитать):

```
$entry =
  $c->search($basedn,$scope,$ARGV[1],0,$attr[0],$attr[1],$attr[2],...);
```

Если мы изменим строку первоначального примера:

```
$entry = $c->search($basedn,$scope,$ARGV[1]);
```

на:

```
@attr = qw(mail);
$entry = $c->search($basedn,$scope,$ARGV[1],0,@attr);
```

то получим следующий результат, в котором для элемента будут показаны только атрибуты DN и mail:

```
...
dn: cn="bear pooh",mail=poohbear219@hotmail.com,c=US,o=hotmail.com
mail: poohbear219@hotmail.com
...
```

Изменения, которые необходимо внести, чтобы получить определенные атрибуты средствами `Net::LDAP`, тоже не сложны:

```
use Net::LDAP;
...
# можно было бы добавить "typesonly => 1" для получения только
# типов атрибутов, как и в предыдущем случае для первого
# необязательного параметра
$searchobj = $c->search(base => $basedn, filter => $ARGV[1],
                        attrs => \@attr);
```

Обратите внимание, что `Net::LDAP` принимает *ссылку* на массив, а не сами значения массива, как в случае с `Mozilla::LDAP`.

Представление элементов в Perl

Эти примеры программ могут вызвать ряд вопросов о представлении элементов и о работе с ними, – в частности, как сами элементы хранятся и обрабатываются в программе на Perl. Дополняя рассказ о поиске в LDAP, ответим на некоторые из них, хотя позже в разделе о добавлении и изменении элементов подобные вопросы будут рассматриваться подробно.

Если `Mozilla::LDAP` выполняет поиск и возвращает экземпляр объекта элемента, то можно обратиться к отдельным атрибутам этого элемента, применяя синтаксис, используемый в Perl при работе с хэшами списков. `$entry->{attributename}` – это *список*[1] значений атрибута с таким именем. Я выделил слово «список», т. к. атрибуты даже с одним

[1] Если точнее, то ссылка на список (как, впрочем, далее явствует из текста). – *Примеч. науч. ред.*

значением хранятся в анонимном списке, на который ссылается этот ключ хэша. Для получения единственного значения атрибута необходимо использовать запись `$entry->{attributename}->[0]`. Некоторые методы модуля `Mozilla::LDAP::Entry` возвращают атрибуты элемента (табл. 6.3).

Таблица 6.3. Методы Mozilla::LDAP::Entry

Вызов метода	Возвращает
`$entry->exists($attrname)`	*true*, если элемент имеет атрибут с таким именем
`$entry->hasValue($attrname,$attrvalue)`	*true*, если элемент имеет названный атрибут с указанным значением
`$entry->matchValue($attrname,$attrvalue)`	Так же, как и предыдущий, только ищется соответствие регулярному выражению, определенному в качестве значения атрибута
`$entry->size($attrname)`	Количество значений этого атрибута (обычно 1, если только атрибут не обладает несколькими значениями)

Отдельные методы имеют дополнительные параметры, узнать о которых можно в документации по `Mozilla::LDAP::Entry`.

Из программы видно, что методы для доступа к атрибутам элементов в `Net::LDAP` несколько отличаются. После проведения поиска все результаты инкапсулируются в один объект. Получить отдельные атрибуты каждого элемента из этого объекта можно, применив один из двух способов.

Во-первых, модуль может преобразовать все полученные элементы в одну большую структуру данных, доступную пользователям. `$searchobj->as_struct()` возвращает структуру данных, представляющую собой хэш хэшей списков. Метод возвращает ссылку на хэш, ключами которого являются DN-имена полученных элементов. Значения ключей – это ссылки на анонимные хэши, ключами которых являются имена атрибутов. Ключам соответствуют ссылки на анонимные массивы, содержащие значения данных атрибутов (рис. 6.1).

Вывести первое значение атрибута `cn` для всех элементов из структуры данных позволяет такой код:

```
$searchstruct = $searchobj->as_struct;
for (keys %$searchstruct){
    print $searchstruct->{$_}{cn}[0],"\n";
}
```

Можно также сначала использовать один из этих методов и выделить объекты для отдельных элементов из объекта, возвращаемого в результате поиска:

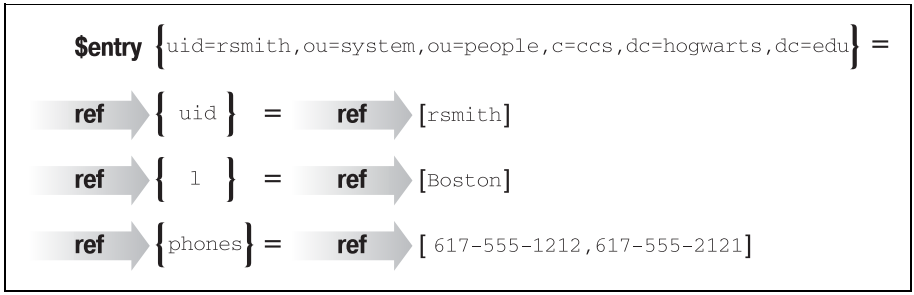

***Рис. 6.1.** Структура данных, возвращаемая методом as_struct( )*

```
# возвращает указанный элемент
$entry   = $searchobj->entry($entrynum);

# действует подобно shift() в Perl для списка элементов
$entry   = $searchobj->shift_entry;

# действет подобно pop() в Perl для списка элементов
$entry   = $searchobj->pop_entry;

# возвращает все элементы в виде списка
@entries = $searchobj->entries;
```

После того как получен объект элемента, можно применить один из указанных методов (табл. 6.4).

Таблица 6.4. Методы элементов Net::LDAP

Вызов метода	Возвращает
`$entry->get($attrname)`	Значение атрибута в указанном элементе
`$entry->attributes()`	Список имен атрибутов для этого элемента

Можно объединить эти методы в довольно четкую цепочку. Например, следующая строка получает значение атрибута `cn` первого возвращаемого элемента:

```
$value = $searchobj->entry(1)->get(cn)
```

Теперь, когда вы умеете получать доступ к отдельным атрибутам и значениям, возвращаемым в результате поиска, посмотрим, как поместить подобные данные в каталог сервера.

Добавление элементов при помощи LDIF

Перед тем как рассматривать общие методы добавления элементов в каталог LDAP, давайте вспомним о названии этой книги и рассмотрим технологию, полезную, в основном, системным администраторам и администраторам каталогов. Она использует формат данных, помогаю-

щий загрузить данные на сервер каталогов. Мы рассмотрим способы записи и чтения LDIF.

LDIF, определенный в нескольких стандартах RFC[1], предлагает простое текстовое представление для элементов каталогов. Вот простой пример LDIF из последнего чернового стандарта Гордона Гуда (Gordon Good):

```
version: 1
dn: cn=Barbara Jensen, ou=Product Development, dc=airius, dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
cn: Barbara Jensen
cn: Barbara J Jensen
cn: Babs Jensen
sn: Jensen
uid: bjensen
telephonenumber: +1 408 555 1212
description: A big sailing fan.

dn: cn=Bjorn Jensen, ou=Accounting, dc=airius, dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
cn: Bjorn Jensen
sn: Jensen
telephonenumber: +1 408 555 1212
```

Формат должен быть вам понятен. После номера версии LDIF перечислены DN-имена каждого элемента, определения `objectClass` и атрибуты. Разделителем элементов является пустая строка.

Наша первоочередная задача – научиться создавать файлы LDIF из существующих элементов каталогов. Кроме того что мы обеспечим себе данные для следующего раздела (в котором рассматривается чтение файлов LDIF), такая возможность позволит использовать LDIF-файлы любым способом при помощи обычных операций Perl, работающих с текстом.

При обсуждении поиска в LDAP было показано, как вывести элементы в формате LDIF. Изменим код предыдущего примера так, чтобы он записывал данные в файл:

```
use Mozilla::LDAP::Conn;
use Mozilla::LDAP::LDIF;

<выполняем связывание и поиск>
```

[1] В период написания книги они были доступны в черновом варианте. – *Примеч. науч. ред.*

```
open(LDIF,">$LDIFfile!") or die ""Невозможно записать в $LDIFfile:$!\n";
# создаем новый объект LDIF и передаем дескриптор
$ldif = new Mozilla::LDAP::LDIF(\*LDIF);

while ($entry) {
    $ldif->writeOneEntry($entry);
    $entry = $c->nextEntry();
}

$c->close();
close(LDIF);
```

Модуль `Mozilla::LDAP` располагает методом `writeEntries()`, позволяющим принять массив элементов и записать их подобным образом.

Используя `Net::LDAP`, изменить первоначальную программу еще проще. Вместо:

```
$ldif = new Net::LDAP::LDIF("-");
```

применим:

```
$ldif = new Net::LDAP::LDIF($filename,"w");
```

для записи выводимых данных в указанный файл, а не на стандартный вывод.

Теперь совершим обратное действие и прочитаем файлы LDIF (вместо того, чтобы в них записывать). Методы объекта из модуля, о котором пойдет речь, позволяют легко добавить элементы в каталог.[2]

При чтении LDIF-данных из Perl осуществляется процесс, обратный тому, который применялся в предыдущих примерах для записи. Каждый список элементов считывается и преобразуется в экземпляр объекта элемента, который затем передается соответствующему методу изменения каталога. Оба модуля считывают и анализируют данные, так что процесс довольно прост. Например, с использованием `Mozilla::LDAP` можно написать такую программу:

```
use Mozilla::LDAP::Conn;
use Mozilla::LDAP::LDIF;

$server   = $ARGV[0];
$LDIFfile = $ARGV[1];
$port     = getservbyname("ldap","tcp") || "389";
```

[1] Предполагается наличие строки «$LDIFfile=$ARGV[1];». – *Примеч. науч. ред.*

[2] Файлы LDIF могут содержать специальную директиву `changetype:`, которая говорит о том, что информацию об элементе необходимо удалить или изменить, а не просто добавить. Из двух применяемых модулей только `Net::LDAP` напрямую поддерживает `changetype:` через метод `Net::LDAP::LDIF::read_cmd()`.

```
$rootdn   = "cn=Manager, ou=Systems, dc=ccs, dc=hogwarts, dc=edu";
$pw       = "secret";

# считываем файл LDIF, указанный вторым аргументом в
# командной строке
open(LDIF,"$LDIFfile") or die "Невозможно открыть $LDIFfile:$!\n";
$ldif = new Mozilla::LDAP::LDIF(\*LDIF);

# анализируем все элементы, сохраняем их в @entries
@entries = $ldif->readEntries();
close(LDIF);

# неанонимное соединение
$c = new Mozilla::LDAP::Conn($server,$port,$rootdn,$pw);
die "Невозможно соединиться с $server\n" unless $c;

# обходим в цикле список элементов, добавляя их на каждой итерации
for (@entries){
    $c->add($_); # добавляем этот элемент в каталог
    warn "Ошибка при добавлении ". $_->getDN().": ".$c->getErrorString()."\n"
      if $c->getErrorCode();
}
$c->close();
```

В этом примере отражено применение методов `getErrorCode()` и `getErrorString()` для получения любых ошибок (и сообщения о них), происходящих в процессе загрузки данных. Ошибки могут появиться по целому ряду причин, включая дублирование DN/RDN-имен, нарушение схемы, проблемы с иерархией и т. д., так что очень важно проверить их при изменении элемента.

И еще одно замечание, перед тем как перейти к рассмотрению `Net::LDAP`: в этом и последующих примерах в демонстрационных целях используется корневое DN-имя (manager DN). Обычно же, если можно избежать применения такого контекста в повседневной работе, это следует делать. Образец правильной настройки LDAP-сервера включает создание могущественной учетной записи или группы учетных записей (которые не являются корневым DN-именем) для управления каталогами. При создании собственных приложений не забывайте этот совет.

Для `Net::LDAP` программа, добавляющая LDIF-элемент, выглядит таким образом:

```
use Net::LDAP;
use Net::LDAP::LDIF;

$server   = $ARGV[0];
$LDIFfile = $ARGV[1];
$port     = getservbyname("ldap","tcp") || "389";
$rootdn   = "cn=Manager, ou=Systems, dc=ccs, dc=hogwarts, dc=edu";
$pw       = "secret";
```

```
# считываем файл LDIF, указанный вторым аргументом в
# командной строке
# последний параметр "r" для чтения, "w" для записи
$ldif = new Net::LDAP::LDIF($LDIFfile,"r");
@entries = $ldif->read();

$c = new Net::LDAP($server, port => $port) or
    die "Невозможно соединиться с $server: $@\n";
$c->bind(dn => $rootdn, password => $pw) or die "Ошибка при связывании:
$@\n";

for (@entries){
    $res = $c->add($_);
    warn "Ошибка при добавлении ". $_->dn().": код ошибки ".$res->code."\n"
      if $res->code();
}

$c->unbind();
```

Несколько замечаний к этому примеру:

- При желании можно объединить два оператора чтения LDIF в одну строку:

```
@entries = new Net::LDAP::LDIF($LDIFfile,"r")->read;
```

- Если попытка `add()` не удалась, следует запросить десятичный код ошибки. Например, возможно такое сообщение:

```
Ошибка при добавлении cn=Ursula Hampster, ou=Alumni Association,
ou=People,
o=University of Michigan, c=US: код ошибки 68
```

  Если сервер возвращает текстовое сообщение, метод `error()` получает его так же, как это было в примере с `Mozilla::LDAP`:

```
print "Сообщение об ошибке: ".$res->error."\n";
```

  Безопаснее было бы проверить код возврата, как в предыдущем примере, поскольку серверы LDAP не всегда передают текстовые сообщения об ошибках в своих ответах. Если нужно преобразовать десятичный код ошибки в сообщение об ошибке или в название сообщения, модуль `Net::LDAP::Util` предлагает для этого две подпрограммы: `ldap_error_text()` и `ldap_error_name()`.

- 68 в десятичной системе счисления – это 44 в шестнадцатеричной. Следовательно, приведенная ниже строка из *Constant.pm* имеет прямое отношение к нашему случаю:

```
sub LDAP_ALREADY_EXISTS          () { 0x44 }
```

Теперь нам известно, что мы пытались добавить элемент из файла LDIF, который уже существовал в каталоге.

Добавление элементов при помощи стандартных операций LDAP

На этот раз мы заглянем вглубь процесса добавления элементов, чтобы научиться создавать и заполнять элементы вручную, не считывая их из файла, как в прошлый раз. Два модуля обрабатывают этот процесс по-разному, поэтому работать с ними следует отдельно. Модуль `Mozilla::LDAP` ближе к классическому стилю объектно-ориентированного программирования. Создадим новый экземпляр объекта:

```
use Mozilla::LDAP::Entry;
$e = new Mozilla::LDAP::Entry()
```

и начнем его заполнять. Следующий шаг – дать элементу отличительное имя DN. Это можно сделать при помощи метода `setDN()`:

```
$e->setDN("uid=jay, ou=systems, ou=people, dc=ccs, dc=hogwarts, dc=edu");
```

Для заполнения других атрибутов, таких как `objectClass`, следует пойти по одному из двух путей. Можно сделать ряд предположений относительно структуры данных, используемой для представления элемента (по существу, это хэш списков), и заполнить ее напрямую:

```
$e->{cn} = ['Jay Sekora'];
```

В данном случае используется имя атрибута в качестве ключа хэша и ссылка на анонимный массив, хранящий данные. Модуль `Mozilla::LDAP` ожидает, что значениями хэша будут ссылки на массив, а не сами данные, так что следующее, хоть и выглядит заманчиво, но будет неверным:

```
# воплощенное зло (или, по крайней мере, просто неверно)
$e->{cn} = 'Jay Sekora';
```

В качестве альтернативы можно действовать наверняка и применять метод объекта для добавления данных:

```
$e->addValue('cn', 'Jay Sekora');
```

Для добавления нескольких значений атрибуту нужно повторно вызывать метод `addValue()`:

```
$e->addValue('title', 'Unix SysAdmin');
$e->addValue('title', 'Part-time Lecturer');
```

Мне больше по душе второй подход, т. к. при его использовании менее вероятно, что программа перестанет работать, если в следующих версиях модуля изменится способ представления данных.

После того как элемент заполнен, можно вызвать метод `add()` для внесения его в каталог. Вот маленький сценарий, который добавляет эле-

мент в каталог. В качестве аргументов командной строки он принимает имя сервера, идентификатор пользователя (будет использоваться как часть отличительного имени) и общее имя:

```
use Mozilla::LDAP::Conn;

$server   = $ARGV[0];
$port     = getservbyname("ldap","tcp") || "389";
$suffix   = "ou=People, ou=Systems, dc=ccs, dc=hogwarts, dc=edu";
$rootdn   = "cn=Manager, ou=Systems, dc=ccs, dc=hogwarts, dc=edu";
$pw       = "secret";

# неанонимное соединение
$c = new Mozilla::LDAP::Conn($server,$port,$rootdn,$pw);
die "Невозможно соединиться с $server\n" unless $c;

$e = new Mozilla::LDAP::Entry;
# DN-имя - это идентификатор пользователя плюс суффикс,
# определяющий, куда поместить его в дереве каталогов
$e->setDN("uid=$ARGV[1],$suffix");
$e->addValue('uid', $ARGV[1]);
$e->addValue('cn', $ARGV[2]);
$c->add($e);
die "Ошибка при добавлении: ". $c->getErrorString()."\n" if $c-
>getErrorCode();
```

Обратите внимание, что в программе не выполняется проверка ошибок при вводе. Если вы пишете сценарий, который действительно может использоваться в интерактивном режиме, необходимо проверять вводимые данные, чтобы убедиться, что в них нет неэкранированных специальных символов, подобных запятым. Обратитесь к ранее приведенному «совету с совой» за разъяснениями о том, как заключать в кавычки значения атрибутов.

Теперь перейдем к `Net::LDAP`. При желании процесс добавления элементов для `Net::LDAP` может быть менее объектно-ориентированным. В него входят модуль `Entry` (`Net::LDAP::Entry`) и конструктор для экземпляра объекта элемента. Однако он содержит еще одну функцию `add()`, которая способна принимать структуру данных для добавления элемента за один шаг:

```
$res = $c->add(
    dn => 'uid=jay, ou=systems, ou=people, dc=ccs, dc=hogwarts, dc=edu',
    attr => [ 'cn'   => 'Jay Sekora',
              'sn'   => 'Sekora',
              'mail' => 'jayguy@ccs.hogwarts.edu',
              'title'=> ['Sysadmin','Part-time Lecturer'],
              'uid'  => 'jayguy',
            ]
            );
die "невозможно добавить, код ошибки #".$res->code() if $res->code();
```

На этот раз add() передается два аргумента.[1] Первый – это DN-имя для элемента; второй – ссылка на анонимный массив пар атрибут-значение. Обратите внимание, что атрибуты с несколькими значениями, например title, определяются при помощи вложенного анонимного массива. Тем, кто привык работать со структурами данных в Perl и кому не нравится объектно-ориентированный стиль программирования, такой подход придется больше по душе.

Удаление элементов

Удаление элементов из каталога – это простое дело (и необратимое, так что будьте осторожны). Вот отрывок программы, из которой, для краткости, снова удален код, реализующий соединение с сервером:

```
use Mozilla::LDAP::Conn;
...
# если у вас есть элемент, вы можете использовать
# $c->delete($entry->getDN())
$c->delete($dn) or
    die "Невозможно удалить элемент: ". $c->getErrorString()."\n";

use Net::LDAP;
...
$res = $c->delete($dn);
die "Невозможно удалить, код ошибки #".$res->code() if $res->code();
```

Важно обратить внимание на то, что в обоих модулях delete()удаляет по одному элементу за один раз. Если необходимо убрать поддерево целиком, сначала следует найти все дочерние элементы этого поддерева, используя пространство sub или one, а затем обойти в цикле возвращаемые значения, удаляя элементы на каждой итерации. После того как уничтожены дочерние элементы, можно удалить вершину этого поддерева.

Изменение имен элементов

Последние операции с LDAP, которые мы рассмотрим, касаются двух типов изменений элементов LDAP. Первый тип – это изменение DN- и RDN-имен. Преобразовать RDN-имя элемента просто, и эта операция поддерживается обоими модулями. Вот версия для Mozilla::LDAP:

```
use Mozilla::LDAP::Conn;
...
$c->modifyRDN($newRDN,$oldDN,$delold) or
    die "Невозможно переименовать элемент:". $c->getErrorString()."\n";
```

[1] Точнее говоря, с точки зрения синтаксиса Perl, передается четыре аргумента. Однако их можно рассматривать как два именованных аргумента. – *Примеч. науч. ред.*

В приведенном отрывке все должно быть понятно, за исключением параметра `$delold` метода `modifyRDN()`. Если он равен `true`, то LDAP-библиотеки удалят из элементов значения, совпадающие с измененными RDN-именами. Например, если первично в RDN-имени элемента содержался атрибут `l` (от «location», местоположение), но само RDN-имя было изменено, то старый атрибут `l` элемента будет удален и останется только новое значение.

Вот эквивалентный вариант для переименования элемента в `Net::LDAP`:

```
use Net::LDAP;
...
$res = $c->moddn($oldDN,
                 newrdn       => $newRDN,
                 deleteoldrdn => 1);
die "Невозможно переименовать, код ошибки #".$res->code() if $res->code();
```

В действительности метод `moddn()` модуля `Net::LDAP` может гораздо больше, чем показано в предыдущем примере. До сих пор изменялось только RDN-имя элемента, в то время как местоположение элемента в иерархии дерева каталогов оставалось прежним. В LDAP версии 3 появилась более мощная операция для переименования, позволяющая произвольным образом менять местоположение элемента в дереве каталогов. Метод `moddn()`, вызванный с дополнительным параметром `newsuperior`, предоставляет доступ к такой возможности. Если добавить параметр таким образом:

```
$result = $c->moddn($oldDN,
                    newrdn       => $newRDN,
                    deleteoldrdn => 1,
                    newsuperior  => $parentDN);
die "Невозможно переименовать, код ошибки #".$res->code() if $res->code();
```

то элемент из `$oldDN` будет перенесен и станет дочерним элементом DN-имени, определенного в `$parentDN`. Гораздо эффективнее использовать этот метод, а не последовательность `add()` или `delete()`, как требовалось раньше, для перемещения элементов в дереве каталогов, но подобная возможность поддерживается не всеми LDAP-серверами. В любом случае, если вы скрупулезно проектируете структуру дерева каталогов, вам реже придется переносить элементы с места на место.

Изменение атрибутов элемента

Теперь перейдем к более распространенным операциям – изменению атрибутов и значений атрибутов элемента. В этом случае тоже существуют значительные различия между модулями `Mozilla::LDAP` и `Net::LDAP`. Применяя `Mozilla::LDAP` для изменения атрибута элемента,

необходимо использовать один из методов, представленных в табл. 6.5.

Таблица 6.5. Методы изменения элементов в Mozilla::LDAP

Метод	Действие
`$entry->addValue($attrname, $attrvalue)`	Добавляет указанное значение заданному атрибуту в указанном элементе.
`$entry-> removeValue($attrname, $attrvalue)`	Удаляет указанное значение для заданного атрибута указанного элемента. Если это значение единственное для атрибута, то удаляется и весь атрибут.
`$entry-> setValue($attrname, $attrvalue1,…)`	Изменяет значения указанного атрибута в заданное значение или значения.
`$entry-> remove($attrname)`	Удаляет указанный атрибут (вместе со значениями) из элемента.

После того как внесены все изменения элементов (при помощи перечисленных методов), нужно вызвать метод `update()` для данного LDAP-соединения, чтобы распространить эти изменения на сервер каталогов. `update()` вызывается со ссылкой на элемент в качестве аргумента (т. е. `$c->update($entry)`).

Применим эти методы для глобального поиска и замены. Рассмотрим такой сценарий: один из отделов вашей компании переводят из Бостона в Индиану. Эта программа изменит все элементы, местоположением которых является Бостон:

```
use Mozilla::LDAP::Conn;

$server = $ARGV[0];
$port   = getservbyname("ldap","tcp") || "389";
$basedn = "dc=ccs,dc=hogwarts,dc=edu";
$scope  = "sub";
$rootdn = "cn=Manager, ou=Systems, dc=ccs, dc=hogwarts, dc=edu";
$pw     = "secret";

# неанонимное соединение
$c = new Mozilla::LDAP::Conn($server,$port,$rootdn,$pw);
die "Невозможно соединиться с сервером $server\n" unless $c;

# обратите внимание, что мы запрашиваем как можно меньше
# информации для ускорения поиска
$entry = $c->search($basedn,$scope,"(l=Boston)",1,'');
die "Ошибка поиска:". $c->getErrorString()."\n" if $c->getErrorCode();

if ($entry){
    while($entry){
        $entry->removeValue("l","Boston");
```

```
            $entry->addValue("l","Indiana");
            $c->update($entry);
            die "Ошибка при обновлении:" . $c->getErrorString() . "\n"
              if $c->getErrorCode();
            $entry = $c->nextEntry();
        };
    }
    $c->close();
```

Для изменения элементов в `Net::LDAP` применяется другой подход. В нем все только что рассмотренные методы модуля `Mozilla::LDAP` объединены в одном «суперметоде» `modify()`. Параметры, передаваемые этому методу, и определяют его функциональность (табл. 6.6).

Таблица 6.6. Методы для изменения элементов в Net::LDAP

Параметр	Действие
`add => {$attrname => $attrvalue}`	Добавляет указанный элемент с заданным значением.
`add => {$attrname => [$attrvalue1, $attrvalue2...]}`	Добавляет указанный атрибут с заданным набором значений.
`delete => {$attrname => $attrvalue}`	Удаляет указанный атрибут с заданным значением.
`delete => {$attrname => []}` `delete => [$attrname1,$attrname2...]`	Удаляет атрибут или набор атрибутов независимо от их значений.
`replace => {$attrname => $attrvalue}`	Действует, как `add`, только заменяет текущее значение указанного атрибута. Если `$attrvalue` является ссылкой на пустой анонимный список (`[]`), метод становится синонимом для приведенной выше операции удаления.

Обязательно обратите внимание на знаки пунктуации в предыдущей таблице. Некоторые параметры принимают ссылки на анонимные хэши, другие – на анонимные массивы. Если их перепутать, трудностей не миновать.

Можно объединять несколько таких параметров в одном и том же вызове `modify()`, но это представляет собой потенциальную проблему. Когда `modify()` вызывается с набором параметров, например, так:

```
$c->modify($dn,replace => {'l' => "Medford"},
               add     => {'l' => "Boston"},
               add     => {'l' => "Cambridge"});
```

нет никаких гарантий, что указанные операции добавления будут выполняться после замены. Если необходимо, чтобы операции выполнялись в определенном порядке, можно применять синтаксис, подобный только что рассмотренному. Вместо использования набора дискрет-

ных параметров можно передать единственный массив, содержащий очередь команд. Вот как это работает: `modify()` принимает параметр `changes`, значение которого – список. Данный список считается набором пар. Первая половина пары – это операция, которую необходимо выполнить, вторая половина – ссылка на анонимный массив, содержащий данные для этой операции. Например, если мы хотим гарантировать, что операции из предыдущего фрагмента кода выполнятся в нужном порядке, то можем написать:

```
$c->modify($dn, changes =>
                  [ replace => ['l' => "Medford"],
                    add     => ['l' => "Boston"],
                    add     => ['l' => "Cambridge"]
                  ]);
```

Внимательно посмотрите на пунктуацию: она отличается от других параметров, которые приводились раньше.

Учитывая информацию, передаваемую функции `modify()`, можно переписать для `Net::LDAP` предыдущую программу, меняющую Бостон на Индиану:

```
use Net::LDAP;

$server   = $ARGV[0];
$port     = getservbyname("ldap","tcp") || "389";
$basedn   = "dc=ccs,dc=hogwarts,dc=edu";
$scope    = "sub";
$rootdn   = "cn=Manager, ou=Systems, dc=ccs, dc=hogwarts, dc=edu";
$pw       = "secret";

$c = new Net::LDAP($server, port => $port) or
  die "Невозможно соединиться с сервером $server: $@\n";
$c->bind(dn => $rootdn, password => $pw) or die "Ошибка при соединении:
$@\n";

$searchobj = $c->search(base  => $basedn, filter => "(l=Boston)",
                        scope => $scope,  attrs  => [''],
                        typesonly => 1);
die "Ошибка поиска: ".$searchobj->error()."\n" if ($searchobj->code());

if ($searchobj){
    @entries = $searchobj->entries;
    for (@entries){
        $res=$c->modify($_->dn(), # dn() получает DN-имя этого элемента
                   delete => {"l" => "Boston"},
                   add    => {"l" => "Indiana"});
        die "Невозможно изменить, код ошибки #".$res->code() if $res->code();
    }
}

$c->unbind();
```

Собираем все вместе

Теперь, когда нам известны все основные LDAP-функции, напишем несколько небольших сценариев для системного администрирования. Мы импортируем базу данных машин из главы 5 «Службы имен TCP/IP» на сервер LDAP и затем сгенерируем некую полезную информацию, основываясь на LDAP-запросах. Вот пара выдержек из этого простого файла (для того только, чтобы напомнить вам формат):

```
name: shimmer
address: 192.168.1.11
aliases: shim shimmy shimmydoodles
owner: David Davis
department: software
building: main
room: 909
manufacturer: Sun
model: Ultra60
-=-
name: bendir
address: 192.168.1.3
aliases: ben bendoodles
owner: Cindy Coltrane
department: IT
building: west
room: 143
manufacturer: Apple
model: 7500/100
-=-
```

Первое, что нужно сделать, – приготовить сервер каталогов для приема этих данных. Мы будем использовать нестандартные атрибуты, так что придется обновить схему сервера. Различные серверы выполняют это по-разному. Например, сервер каталогов Netscape имеет симпатичную графическую консоль Directory Server Console для подобных изменений. Другие серверы требуют внесения изменений в текстовые конфигурационные файлы. Работая с OpenLDAP, можно использовать нечто подобное в файле, включенном основным конфигурационным файлом для определения собственных пользовательских классов объектов для машины:

```
objectclass machine
        requires
                objectClass,
                cn
        allows
                address,
                aliases,
                owner,
                department,
```

```
building,
room,
manufacturer,
model
```

После того как сервер настроен нужным образом, можно подумать об импортировании данных. Один из вариантов – провести загрузку большой единой операцией с помощью LDIF. Если приведенный выше отрывок из базы данных напомнил вам о формате LDIF, значит, вы на правильном пути. Эта схожесть упрощает преобразование. Тем не менее, нужно остерегаться ловушек:

Продолжающиеся строки

В нашей базе данных нет элементов, значения которых занимали бы несколько строк, иначе следовало бы убедиться, что вывод удовлетворяет стандарту LDIF. Стандарт LDIF требует, чтобы все длинные строки начинались строго с одного пробела.

Разделители элементов

Между элементами в базе данных в качестве разделителя используется симпатичная последовательность `-=-`. Два разделителя строк (т. е. пустая строка) должны находиться между элементами LDIF, так что нужно будет удалить эту последовательность из вводимых данных.

Разделители атрибутов

В настоящее время в наших данных есть только один атрибут с несколькими значениями: aliases (псевдонимы). LDIF обрабатывает многозначные атрибуты, перечисляя каждое значение на отдельной строке. Если встретится несколько атрибутов, то понадобится специальный код, печатающий для каждого значения отдельную строку. Если бы не эта особенность, программа, преобразующая наш формат в LDIF, представляла бы собой одну строку кода на Perl.

Но даже и с этими ловушками программа преобразования на удивление проста:

```
$datafile = "database";
$recordsep = "-=-\n";
$suffix   = "ou=data, ou=systems, dc=ccs, dc=hogwarts, dc=edu";
$objectclass = <<EOC;
objectclass: top
objectclass: machine
EOC

open(DATA,$datafile) or die "Невозможно открыть $datafile:$!\n";

# Модули Perl не работают с этим, даже если в спецификации сказано обратное
# print "version: 1\n"; #
```

```
while (<DATA>) {
    # выводим заголовок для каждого элемента
    if (/name:\s*(.*)/){
        print "dn: cn=$1, $suffix\n";
        print $objectclass;
        print "cn: $1\n";
        next;
    }
    # обрабатываем многозначный атрибут aliases
    if (s/^aliases:\s*//){
        @aliases = split;
        foreach $name (@aliases){
            print "aliases: $name\n";
        }
        next;
    }
    # обрабатываем конец разделителя записей
    if ($_ eq $recordsep){
        print "\n";
        next;
    }
    # в противном случае просто печатаем найденный атрибут
    print;
}

close(DATA);
```

Если выполнить эту программу, то она выведет файл LDIF, выглядящий примерно так:

```
dn: cn=shimmer, ou=data, ou=systems, dc=ccs, dc=hogwarts, dc=edu
objectclass: top
objectclass: machine
cn: shimmer
address: 192.168.1.11
aliases: shim
aliases: shimmy
aliases: shimmydoodles
owner: David Davis
department: software
building: main
room: 909
manufacturer: Sun
model: Ultra60

dn: cn=bendir, ou=data, ou=systems, dc=ccs, dc=hogwarts, dc=edu
objectclass: top
objectclass: machine
cn: bendir
address: 192.168.1.3
aliases: ben
```

```
aliases: bendoodles
owner: Cindy Coltrane
department: IT
building: west
room: 143
manufacturer: Apple
model: 7500/100
...
```

Имея этот LDIF-файл, можно применять одну из программ, распространяемых с сервером, для загрузки этих данных на сервер. Например, *ldif2ldbm*, входящий в состав обоих серверов OpenLDAP и Netscape Directory Server, считывает LDIF-файл и напрямую импортирует его в формат сервера каталогов, избавляя от необходимости проходить через LDAP. Хотя эта программа используется только при неработающем сервере, она может обеспечить самый быстрый способ загрузки большого количества данных. Если нельзя остановить сервер, можно применить только что написанную на Perl программу для чтения LDIF-файлов и передать подобный файл на LDAP-сервер.

Добавим еще один способ: вот программа, в которой пропускается промежуточный шаг создания LDIF-файла, и наши данные напрямую импортируются на LDAP-сервер:

```
use Net::LDAP;
use Net::LDAP::Entry;

$datafile  = "database";
$recordsep = "-=-";
$server    = $ARGV[0];
$port      = getservbyname("ldap","tcp") || "389";
$suffix    = "ou=data, ou=systems, dc=ccs, dc=hogwarts, dc=edu";
$rootdn    = "cn=Manager, o=University of Michigan, c=US";
$pw        = "secret";

$c = new Net::LDAP($server,port => $port) or
  die "Невозможно соединиться с сервером $server: $@\n";
$c->bind(dn => $rootdn,password => $pw) or die "Ошибка при соединении: $@\n";

open(DATA,$datafile) or die "Невозможно открыть $datafile:$!\n";

while (<DATA>) {
    chomp;
    # в начале новой записи создаем новый экземпляр объекта
    if (/^name:\s*(.*)/){
        $dn="cn=$1, $suffix";
        $entry = new Net::LDAP::Entry;
        $entry->add("cn",$1);
        next;
    }
    # особый случай -- многозначные атрибуты
```

```
    if (s/^aliases:\s*//){
        $entry->add('aliases',[split()]);
        next;
    }

    # если дошли до конца записи, добавляем ее не сервер
    if ($_ eq $recordsep){
        $entry->add("objectclass",["top","machine"]);
        $entry->dn($dn);
        $res = $c->add($entry);
        warn "Ошибка добавления для " . $entry->dn() . ": код ошибки " .
              $res->code."\n"
          if $res->code();
        undef $entry;
        next;
    }

    # добавляем все остальные атрибуты
    $entry->add(split(':\s*')); # считаем, что у атрибута только одно
значение
}

close(DATA);
$c->unbind();
```

После того как данные были импортированы на сервер, можно приступить к довольно любопытным вещам. В следующих примерах мы будем поочередно обращаться к двум LDAP-модулям. Для краткости в каждом примере не будет повторяться заголовок, в котором устанавливаются конфигурационные переменные, и код для соединения с сервером.

Так что же можно сделать с данными, расположенными на сервере LDAP? Можно на лету создать файл узлов:

```
use Mozilla::LDAP;
...
$entry = $c->search($basedn,'one','(objectclass=machine)',0,
                    'cn','address','aliases');
die "Ошибка поиска:". $c->getErrorString()."\n" if $c->getErrorCode();

if ($entry){
    print "#\n\# host file - GENERATED BY $0\n
           # DO NOT EDIT BY HAND!\n#\n";
    while($entry){
        print $entry->{address}[0],"\t",
              $entry->{cn}[0]," ",
              join(' ',@{$entry->{aliases}}),"\n";
        $entry = $c->nextEntry();
    };
}
$c->close();
```

Вот что получается:

```
#
# host file - GENERATED BY ldap2hosts
# DO NOT EDIT BY HAND!
#
192.168.1.11    shimmer shim shimmy shimmydoodles
192.168.1.3     bendir ben bendoodles
192.168.1.12    sulawesi sula su-lee
192.168.1.55    sander sandy mickey mickeydoo
```

Можно найти имена всех машин, произведенных Apple:

```
use Net::LDAP;
...
$searchobj = $c->search(base  => $basedn,
                        filter => "(manufacturer=Apple)",
                        scope => 'one', attrs => ['cn']);
die "Ошибка поиска: ".$searchobj->error()."\n" if ($searchobj->code());

if ($searchobj){
    for ($searchobj->entries){
        print $_->get('cn'),"\n";
    }
}

$c->unbind();
```

Вот и результат:

```
bendir
sulawesi
```

Можно сгенерировать список владельцев машин:

```
use Mozilla::LDAP;
...
$entry = $c->search($basedn,'one','(objectclass=machine)',0,
                    'cn','owner');
die "Ошибка поиска:". $c->getErrorString()."\n" if $c->getErrorCode();

if ($entry){
    while($entry){
        push(@{$owners{$entry->{owner}[0]}},$entry->{cn}[0]);
        $entry = $c->nextEntry();
    };
}
$c->close();
for (sort keys %owners){
    print $_.":\t".join(' ',@{$owners{$_}})."\n";
}
```

Получилось так:

```
Alex Rollins:   sander
Cindy Coltrane: bendir
David Davis:    shimmer
Ellen Monk:     sulawesi
```

Заодно можно проверить, является ли владельцем машины пользователь с текущим идентификатором (псевдо-аутентификация):

```
use Mozilla::LDAP::Conn;
use Sys::Hostname;

$user = (getpwuid($<))[6];

$hostname = hostname;
$hostname =~ s/^([^.]+)\..*/$1/; # удаляем имя домена из имени узла
...
$entry = $c->search("cn=$hostname,$suffix",'base',"(owner=$user)",1,'');

if ($entry){
    print "Владелец ($user) зарегистрирован на машине $hostname.\n";
}
else {
    print "$user не является владельцем этой машины ($hostname)\n.";
}
$c->close();
```

Эти отрывки должны показать, как можно использовать доступ к LDAP из Perl для системного администрирования, и вдохновить вас на создание собственных программ. В следующем разделе эти идеи будут перенесены на новый уровень, что позволит нам увидеть целый интерфейс администрирования, построенный на основе LDAP.

ADSI (Интерфейсы служб активных каталогов)

В последнем разделе этой главы мы обсудим платформо-зависимый интерфейс службы каталогов, разработанный с учетом только что рассмотренного материала.

В Microsoft создали сложную службу каталогов, основанную на LDAP, под названием Active Directory для использования ее в сердцевине интерфейса администрирования Windows 2000. Служба Active Directory является репозиторием для всей важной конфигурационной информации (пользователи, группы, системные политики, поддержка установки программного обеспечения и т. д.), применяемой в сети машин с Windows 2000.

При разработке активных каталогов в Microsoft осознали, что для этой службы необходимо создать программный интерфейс более высокого уровня. Для этого был разработан интерфейс ADSI (Active Directory Service Interfaces, интерфейсы служб активных каталогов). Надо отдать должное разработчикам Microsoft, им удалось понять, что новый интерфейс нужно будет расширить и охватить такие области системного администрирования, как принтеры и службы NT. Подобный размах делает ADSI чрезвычайно полезным для тех, кто пишет сценарии, автоматизирующие выполнение задач системного администрирования. Перед тем как ознакомиться с его работой, приведем несколько основных концепций и терминов, которые нам понадобятся.

Основы ADSI

ADSI можно считать оболочкой вокруг произвольной службы каталогов, действующей в рамках ADSI. В среде интерфейса есть *провайдеры* (*providers*), которые представляют собой реализации ADSI для LDAP, WinNT 4.0 и Novell Directory Service. Говоря на «языке» ADSI, каждая из этих служб каталогов (WinNT не является службой каталогов) и домены данных (data domains) называются *пространством имен* (*namespaces*). ADSI предоставляет единый способ запроса и изменения данных, найденных в этих пространствах имен.

Чтобы понять ADSI, необходимо иметь представление об объектной модели компонентов COM (Component Object Model), на которой построен интерфейс ADSI. О модели COM существует много книг, но следует остановится на таких ключевых понятиях:

- Все, с чем работают в COM, – это *объекты* (*objects*).[1]
- Объекты имеют *интерфейсы* (*interfaces*), обеспечивающие набор *методов* (*methods*), применяемых для взаимодействия с этими объектами. Из Perl можно использовать методы, предоставляемые или наследуемые от интерфейса под названием *IDispatch*. К счастью, большинство методов ADSI, предоставляемых интерфейсами ADSI и их производными (например, IADsUser, IADsComputer, IADsPrintQueue), унаследованы от *IDispatch*.
- Значения, инкапсулируемые объектом, запрашиваемые и изменяемые посредством этих методов, называются *свойствами* (*properties*). В этой главе будут рассматриваться два типа значений: свойства, *определяемые интерфейсом* (*interface-defined properties*), и свойства, *определяемые схемой* (*schema-defined properties*). Иными словами,

[1] На самом деле COM – это протокол, используемый для связи с этими объектами как часть более крупного стандарта OLE (связывание и встраивание объектов). В данном разделе я постараюсь оградить читателя от трясины акронимов Microsoft, но те, кому нужны подробности, могут обратиться к ресурсам, доступным на *http://www.microsoft.com/com*.

первые будут определяться как часть интерфейса, а вторые – в объекте схемы. Подробнее об этом говорится чуть ниже. Если в данной главе не будут явно упоминаться «свойства схемы», значит, подразумевать следует свойства интерфейса.

Все это относится к стандартным понятиям объектно-ориентированного программирования. Сложности начинаются, когда сталкиваются терминологии ADSI/COM и других объектно-ориентированных миров, подобных LDAP.

Например, в ADSI рассматривается два различных типа объектов: *лист* (*leaf*) и *контейнер* (*container*). Объект-лист содержит данные; объект-контейнер содержит другие объекты, т. е. является для них *родительским* (*parent*). В LDAP самыми точными определениями для этих терминов были бы «элемент» и «точка ветвления». С одной стороны, мы говорим об объектах со свойствами, а с другой – об элементах с атрибутами. Как разобраться с подобными разногласиями, если учесть, что оба названия определяют одни и те же данные?

Вот как можно это понимать: в действительности, сервер LDAP обеспечивает доступ к дереву элементов и связанных с ними атрибутов. Когда интерфейс ADSI используется вместо LDAP для получения элемента дерева, ADSI вытягивает элемент из сервера LDAP, заворачивает его в несколько слоев блестящей оберточной бумаги и передает вам в качестве COM-объекта. Для получения содержимого этой посылки следует применять нужные методы, которые теперь называются «свойствами». Если внести изменения в свойства данного объекта, можно вернуть объект ADSI, чтобы последний распаковал информацию и передал ее обратно в дерево LDAP.

Вполне разумным выглядит вопрос: «А почему бы не обратиться напрямую к серверу LDAP?» На этот вопрос есть два хороших ответа: если мы знаем, как использовать ADSI для работы с одним типом служб каталогов, то мы знаем, как работать со всеми ними (или, по крайней мере, с теми, которые имеют ADSI-провайдеры). Второй ответ будет дан чуть позже, когда станет ясно, как можно упростить программирование служб каталогов при помощи инкапсуляции ADSI.

Необходимо ввести понятие *AdsPaths*, чтобы перейти к ADSI-программированию в Perl. ADsPaths предоставляет нам уникальный способ ссылаться на объекты из любого пространства имен. Они выглядят так:

```
<progID>:<path to object>
```

`<progID>` – это идентификатор провайдера (например WinNT или LDAP), а `<path to object>` – это специфичный для провайдеров способ поиска объекта в пространстве имен. Часть `<progID>` *чувствительна к регистру*. Если использовать `winnt`, `ldap` или `WINNT` вместо `WinNT` и `LDAP`, программы не будут работать.

Вот как выглядят примеры ADsPath из документации ADSI SDK:

```
WinNT://MyDomain/MyServer/User
WinNT://MyDomain/JohnSmith,user
LDAP://ldapsvr/CN=TopHat,DC=DEV,DC=MSFT,DC=COM,O=Internet
LDAP://MyDomain.microsoft.com/CN=TopH,DC=DEV,DC=MSFT,DC=COM,O=Internet
```

Это не совпадение, что они похожи на URL, т. к. и URL и ADsPath служат одним и тем же целям. Они оба пытаются обеспечить недвусмысленный способ сослаться на данные, предоставляемые различными службами данных. В случае с AdsPath из LDAP используется синтаксис LDAP URL из RFC, упомянутых в приложении B (RFC2255).

Более подробно AdsPath будет рассматриваться при обсуждении двух уже упомянутых пространств имен – *WinNT* и *LDAP*. Но сначала разберемся, как, в общих чертах, ADSI используется из Perl.

Использование ADSI из Perl

Семейство модулей `Win32::OLE`, поддерживаемое Жаном Дюбуа (Jan Dubois) и Гурусами Сарати (Gurusamy Sarathy), предоставляет мост от Perl к ADSI (который построен на COM как часть OLE). После загрузки основного модуля он используется для запроса ADSI-объектов:

```
use Win32::OLE;

$adsobj = Win32::OLE->GetObject($ADsPath) or
    die "Невозможно получить объект для $ADsPath\n";
```

`Win32::OLE->GetObject()` принимает *моникер* (*moniker*) OLE (уникальный идентификатор объекта, в данном случае это ADsPath) и возвращает объект ADSI. Этот вызов также обрабатывает процесс *связывания* (*binding*) с объектом, уже рассмотренный при обсуждении LDAP. По умолчанию связывание с объектом производится от имени пользователя, запускающего сценарий.

Этот совет может уберечь вас от испуга. Если выполнить эти две строчки кода в отладчике и изучить содержимое возвращаемой ссылки на объект, то можно увидеть нечто подобное:

```
DB<3> x $adsobj
0  Win32::OLE=HASH(0x10fe0d4)
     empty hash
```

Не волнуйтесь. `Win32::OLE` использует все могущество связанных переменных (tied variables). Кажущаяся пустой структура данных, как по волшебству, передаст информацию из объекта, если верно к ней обратиться.

Для доступа к значениям свойств интерфейса объекта ADSI используется ссылка на хэш:

Инструменты ADSI

Для использования материала из этой главы необходимо установить ADSI хотя бы на одной машине в сети. Эта машина может служить (через DCOM) ADSI шлюзом для остальных машин. Посетите сайт Тоби Эверета (Toby Everett), ссылка на который приведена ниже, чтобы узнать подробнее, как настроить ADSI для работы с DCOM.

Любая машина с Windows 2000 имеет встроенный в операционную систему интерфейс ADSI. Для всех остальных Win32-машин придется загрузить и установить бесплатный дистрибутив ADSI 2.5, находящийся в *http://www.microsoft.com/adsi*. По этой же ссылке вы найдете документацию по ADSI, включая *adsi25.chm*, – сжатую помощь в формате HTML, содержащую лучшую доступную документацию по ADSI.

Даже если вы работаете с Windows 2000, я советую загрузить ADSI SDK с сайта Microsoft по указанной ссылке, поскольку в него входит эта документация и удобный броузер объектов ADSI под названием *AdsVW*. SDK поставляется с примерами программирования ADSI на нескольких языках, включая Perl. К сожалению, примеры из текущего дистрибутива ADSI полагаются на устаревший модуль *OLE.pm*, так что, в лучшем случае, вы сможете получить несколько советов, но не надо использовать эти примеры в качестве стартовой точки.

Перед тем как начать писать программы, стоит загрузить броузер объектов ADSI Тоби Эверета (написанный на Perl) с *http://opensource.activestate.com/authors/tobyeverett*. Он научит вас перемещаться по пространствам имен ADSI. Обязательно посетите этот сайт, начиная карьеру программиста ADSI, поскольку он является одним из лучших доступных сайтов по применению ADSI из Perl.

```
$value = $adsobj->{key}
```

Например, если этот объект имеет свойство `Name`, определенное как часть его интерфейса (а так и есть), вы можете применить:

```
print $adsobj->{Name}."\n";
```

При помощи такой же записи можно присваивать значения свойствам интерфейсов:

```
$adsobj->{FullName}= "Oog";  # устанавливаем свойство в кэше
```

Свойства объекта ADSI хранятся в кэше (называемом *кэшем свойств* (*property cache*)). Первый запрос к свойствам объекта заполняет дан-

ный кэш. Последующие запросы к тем же свойствам позволяют получить информацию из этого кэша, а *не из службы каталогов*. Если вы хотите вручную заполнить кэш, можно вызвать методы `GetInfo()` или `GetInfoEx()` (расширенная версия `GetInfo()`) для данного экземпляра объекта, применяя синтаксис, который скоро будет рассмотрен.

Из-за того что первое считывание информации происходит автоматически, методы `GetInfo()` и `GetInfoEx()` часто остаются незамеченными. Существуют ситуации, когда эти методы следует употреблять, хотя в книге такие случаи рассматриваться не будут. Вот две подобные ситуации:

1. Некоторые свойства объектов можно получить, только явно вызвав `GetInfoEx()`. LDAP-провайдер Microsoft Exchange 5.5 представляет собой самый характерный пример, поскольку многие из его свойств не доступны, если не вызвать сначала `GetInfoEx()`. Детальную информацию об этой несовместимости можно найти на *http://opensource.activestate.com/authors/tobyeverett*.
2. Если несколько человек имеют право изменять в каталоге данные, то вызванный вами объект может быть кем-то преобразован, пока вы с ним работаете. Если это произойдет, данные в кэше свойств этого объекта устареют. `GetInfo()` и `GetInfoEx()` обновят этот кэш.

Для обновления службы каталогов и источников данных, предоставляемых через ADSI, после изменения объекта *нужно* вызвать специальный метод `SetInfo()`. `SetInfo()` сбрасывает изменения из кэша свойств в службу каталогов и источники данных. (Это должно напомнить вам о необходимости вызывать метод `update()` в `Mozilla::LDAP`. В данном случае идея та же.)

Вызывать методы экземпляра объекта ADSI не сложно:

```
$adsobj->Method($arguments...)
```

Поэтому, если бы мы изменили свойства объекта, как это предлагалось сделать в предыдущем предупреждении, то могли бы использовать такую строку сразу же после кода, вносящего изменения:

```
$adsobj->SetInfo();
```

В результате данные из кэша свойств помещаются обратно в службу каталогов или источник данных.

Вы будете часто использовать метод `Win32::OLE->LastError()` из модуля `Win32::OLE`. Он возвращает ошибку, полученную в результате последней операции OLE. Применение ключа *–w* с Perl (т. е. *perl –w script*) также приводит к подробным сообщениям о неудачных попытках OLE-операций. Зачастую эти сообщения об ошибках – единственная помощь, которая вам доступна, так что попытайтесь с толком ее использовать.

ADSI-код, который до сих пор рассматривался, выглядел как обычный код на Perl, поскольку внешне они похожи. Теперь перейдем к более сложным вопросам.

Работа с объектами контейнер/коллекция

Ранее в этом разделе уже упоминались два типа объектов ADSI: лист и контейнер. Объект-лист представляет собой только данные, тогда как контейнер (известный еще как коллекция – в терминах OLE/COM) содержит другие объекты. Еще одно отличие двух типов объектов в контексте ADSI состоит в том, что объект-лист не имеет дочерних объектов в иерархии, а у контейнеров такие объекты есть.

Объекты-контейнеры требуют специальной обработки, т. к. в большинстве случаев нас интересуют данные, инкапсулированные их дочерними объектами. Существует два способа обратиться к таким объектам из Perl. `Win32::OLE` имеет специальную функцию под названием `in()`, которая недоступна по умолчанию, если модуль загружается стандартным способом. Если необходимо получить к ней доступ, надо в начале программы использовать следующее:

```
use Win32::OLE 'in';
```

`in()` возвращает список ссылок на дочерние объекты, хранящиеся в этом контейнере. Это позволяет писать легко читаемые программы на Perl:

```
foreach $child (in $adsobj){
    print $child->{Name}
}
```

Другой путь заключается в том, чтобы загрузить один из полезных потомков `Win32::OLE` под названием `Win32::OLE::Enum`. `Win32::OLE::Enum->new()` создает объект-перечислитель из какого-либо объекта-контейнера:

```
use Win32::OLE::Enum;

$enobj = Win32::OLE::Enum->new($adsobj);
```

Для этого объекта можно вызвать несколько методов и получить дочерние объекты `$adsobj`. Подобный подход должен напомнить вам способ, применяемый в операциях поиска с `Mozilla::LDAP`; процесс тот же самый.

`$enobj->Next()` возвращает ссылку на следующий экземпляр дочернего объекта (или следующие X объектов, если задан необязательный параметр). `$enobj->All` возвращает список ссылок на экземпляры объектов. `Win32::OLE::Enum` предлагает несколько больше методов (подробнее о них сказано в документации), но этими вы будете пользоваться чаще всего.

Идентификация объекта-контейнера

Заранее нельзя узнать, является ли объект контейнером. Не существует способа из Perl «спросить» объект, не контейнер ли он. Максимум, что можно сделать, – попытаться создать объект-перечислитель и, если эта попытка не удастся, фиксировать данный результат. Вот короткий пример, который делает именно это:

```
use Win32::OLE;
use Win32::OLE::Enum;

eval {$enobj = Win32::OLE::Enum->new($adsobj)};
print "Объект " . ($@ ? "не " : "") . "является контейнером \n";
```

Второй способ – посмотреть на другие источники, описывающие этот объект. Все это плавно перетекает в третью сложность.

Как же узнать что-нибудь об объекте?

До сих пор мы избегали одного большого и, возможно, самого важного вопроса. Скоро нам придется работать с объектами из двух пространств имен. Уже понятно, как получить и установить свойства объектов и как вызвать методы для этих объектов, но все справедливо только в случае, если известны названия этих свойств и методов. Откуда берутся эти названия? Как их можно найти?

Нет единого места, в котором можно найти ответы на эти вопросы, но существует несколько источников, из которых можно почерпнуть нужную информацию для формирования практически всей картины. Первое место – это документация по ADSI, особенно та помощь, о которой говорилось во врезке «Инструменты для ADSI». В этом файле содержится огромное количество материала. Для ответа на наш вопрос о названиях свойств и методов нужно начать с *Active Directory Service Interfaces 2.5→ADSI Reference→ADSI System Providers*.

Иногда имена методов можно найти только в документации, но существует другой, более интересный подход для поиска названий свойств. Можно использовать метаданные, предоставляемые самим ADSI. Именно здесь на сцену выходят свойства схемы, о которых говорилось раньше.

Каждый объект ADSI имеет свойство под названием `Schema`, которое связывает ADsPath с его объектом схемы. В частности, следующий пример:

```
use Win32::OLE;

$ADsPath = "WinNT://BEESKNEES,computer";
$adsobj  = Win32::OLE->GetObject($ADsPath) or
    die "Невозможно получить объект для $ADsPath\n";
print "Это объект ".$adsobj->{Class}.", схема находится в:\n".
```

```
        $adsobj->{Schema},"\n";
```

выведет:

```
Это объект Computer, схема находится в: WinNT://DomainName/Schema/Computer
```

Значение `$adsobj->{Schema}` – это путь ADsPath к объекту, описывающему схему для объектов класса `Computer` в этом домене. Здесь мы используем термин «схема» в том же смысле, что и в разговоре про схемы LDAP. В LDAP схемы определяют, какие атрибуты могут и должны присутствовать в элементах определенных классов объектов. В ADSI схема содержит ту же информацию об объектах определенного класса и их свойства схемы.

При желании посмотреть на возможные имена атрибутов объекта следовало бы взглянуть на значения двух свойств объекта схемы: `MandatoryProperties` и `OptionalProperties`. Изменим предыдущий оператор `print`:

```
$schmobj = Win32::OLE->GetObject($adsobj->{Schema}) or
             die "Невозможно получить объект для $ADsPath\n";
print join("\n",@{$schmobj->{MandatoryProperties}},
                @{$schmobj->{OptionalProperties}}),"\n";
```

Тогда получится:

```
Owner
Division
OperatingSystem
OperatingSystemVersion
Processor
ProcessorCount
```

Теперь известны возможные имена свойств схемы в пространстве имен WinNT для объектов `Computer`. Отлично.

Свойства схемы получаются и устанавливаются несколько иначе, чем свойства интерфейсов. Свойства интерфейсов обрабатываются примерно так:

```
# получение и установка свойств ИНТЕРФЕЙСОВ
$value = $obj->{property};
$obj->{property} = $value;
```

Свойства схемы получаются и устанавливаются при помощи специальных методов:

```
# получение и установка свойств СХЕМЫ
$value = $obj->Get("property");
$obj->Put("property","value");
```

Все, что касается свойств интерфейсов, о чем говорилось до сих пор, остается справедливым и для свойств схемы (т. е. кэш свойств, `SetInfo()`,

и т. д.). Помимо необходимости применения специальных методов для получения и установки значений, единственное, что отличает данные свойства, – это их имена. Иногда один и тот же объект может иметь два различных имени для одного и того же свойства, одно для свойств интерфейса, другое для свойств схемы. Например, два этих свойства получают основные настройки для пользователя:

```
$len = $userobj->{PasswordMinimumLength};  # свойство интерфейса
$len = $userobj->Get("MinPasswordLength"); # то же самое свойство схемы
```

Наличие двух типов свойств обусловлено тем, что свойства интерфейса существуют в виде части модели COM. Разработчики, определяя интерфейс при создании программы, также определяют свойства интерфейса. Позже, если они хотят расширить набор свойств, им приходится изменять и COM-интерфейс, и любой код, использующий этот интерфейс. В ADSI разработчики могут изменить свойства схемы в провайдере без необходимости изменять лежащий в основе COM интерфейс этого провайдера. Очень важно разобраться с обоими типами свойств, т. к. иногда некоторые данные объекта доступны через свойства только одного типа.

На практике, если вы ищете только названия свойств интерфейса или схемы и не собираетесь писать программы для их поиска, я рекомендую использовать ADSI-броузер Тоби Эверета, о котором я упоминал ранее. Вот пример этого броузера в действии (рис. 6.2).

Как альтернативный вариант упомянем программу *ADSIDump* из каталога *General* примеров SDK, которая может вывести содержимое всего дерева ADSI.

Поиск

Эта последняя сложность, которую следует обсудить, перед тем как двигаться дальше. В разделе «LDAP: сложная служба каталогов» мы провели достаточно времени в разговорах о поиске в LDAP. Но в мире ADSI мы вряд ли услышим хоть слово по этому поводу. Все из-за того, что в Perl (и любом другом языке, в котором используется тот же OLE-интерфейс автоматизации) поиск с ADSI очень сложен; более того, поиск поддеревьев и поиск, в котором используются не самые простые фильтры, мучительно сложен. (Все остальное не так плохо.) Сложный поиск проблематичен, т. к. для его выполнения необходимо выйти за пределы ADSI и использовать совершенно иную методологию для получения данных (не говоря уже о том, что придется выучить новые акронимы от Microsoft).

Но тот, кто занимается системным администрированием, привык смеяться над сложностями, так что начнем с простого поиска, а потом перейдем к более сложным вопросам. Простой поиск, затрагивающий один объект (пространство `base`) или его непосредственных потомков

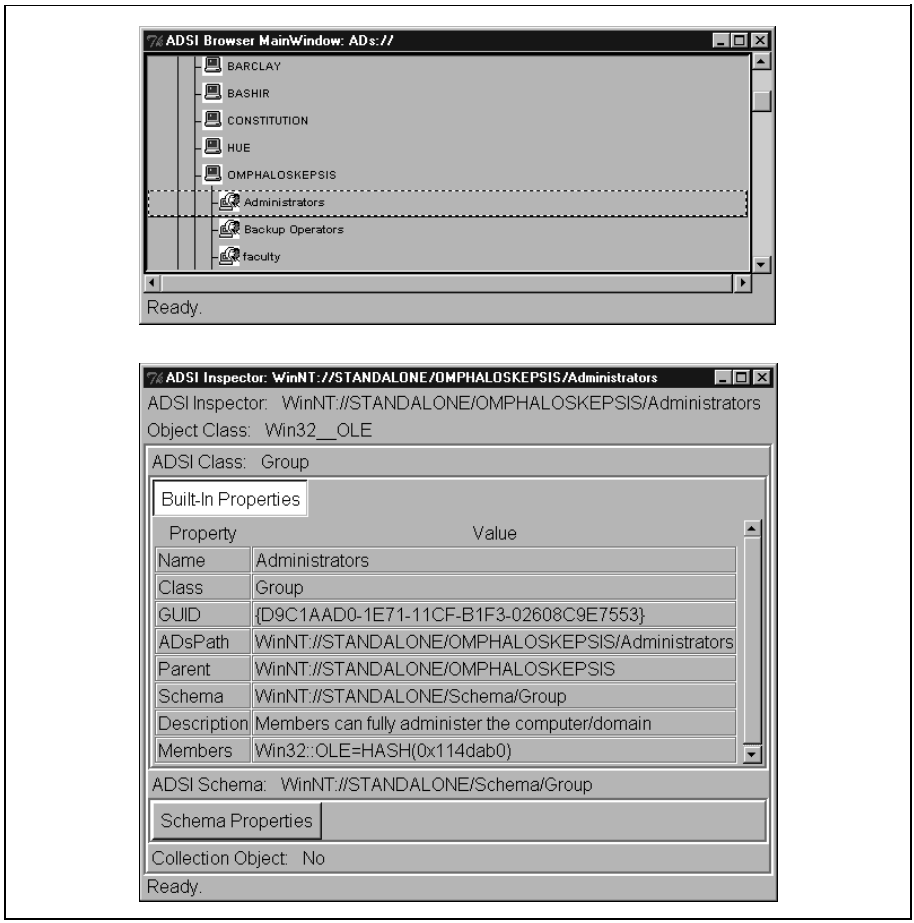

Рис. 6.2. *ADSI-броузер Эверета, отображающий объект Administrators*

(пространство `one`), можно выполнить вручную при помощи Perl. Сделать это можно так:

- Для одного объекта получите нужные свойства и используйте обычные операторы сравнения для определения соответствия:

```
if ($adsobj->{cn} eq "Mark Sausville" and $adsobj->{State} eq "CA"){...}
```

- Для поиска дочерних объектов примените технологии доступа к контейнерам, о которых говорилось раньше, а затем изучите каждый дочерний объект. Несколько примеров поиска такого типа будут рассмотрены очень скоро.

Для того чтобы выполнить более сложный поиск, затрагивающий, скажем, все дерево каталогов или поддерево, вам придется переключиться на использование другой технологии «промежуточного уров-

ня» под названием ADO (ActiveX Data Objects, объекты данных ActiveX). ADO предоставляет языкам сценариев интерфейс к базам уровня Microsoft OLE DB. OLE DB обеспечивает общий интерфейс, ориентированный на базы данных, к источникам данных, подобным реляционным базам данных и службам каталогов. В нашем случае ADO будет применяться для «разговора» с ADSI (который, в свою очередь, общается с самой службой каталогов). Поскольку ADO – это методология, ориентированная на базы данных, рассматриваемая программа предваряет материал об ODBC, о котором речь пойдет в главе 7.

ADO работает только с провайдером LDAP ADSI.
В пространстве имен WinNT она работать не будет.

ADO – это отдельная тема, которая лишь затрагивает службы каталогов, поэтому будет рассмотрен только один пример с короткими пояснениями, остальные примеры работают с ADSI. Дополнительную информацию об ADO можно найти на *http://www.microsoft.com/ado*.

Вот пример программы, выводящей имена всех групп, найденных в данном домене. Детальное обсуждение программы приведено ниже.

```
use Win32::OLE 'in';

# получаем объект ADO, устанавливаем провайдер, открываем соединение
$c = Win32::OLE->new("ADODB.Connection");
$c->{Provider} = "ADsDSOObject";
$c->Open("ADSI Provider");
die Win32::OLE->LastError() if Win32::OLE->LastError();

# подготавливаем и выполняем запрос
$ADsPath = "LDAP://ldapserver/dc=example,dc=com";
$rs = $c->Execute("<$ADsPath>;(objectClass=Group);Name;SubTree");
die Win32::OLE->LastError() if Win32::OLE->LastError();

until ($rs->EOF){
    print $rs->Fields(0)->{Value},"\n";
    $rs->MoveNext;
}

$rs->Close;
$c->Close;
```

Блок кода после загрузки модуля получает экземпляр объекта ADO Connection, устанавливает имя провайдера для этого экземпляра объекта, а затем просит его открыть соединение. Соединение открывается от имени пользователя, запускающего сценарий, хотя можно было установить другие свойства объекта, позволяющие изменить такое поведение.

Затем выполняется собственно поиск при помощи `Execute()`. Поиск можно осуществлять средствами одного из двух «диалектов»: SQL или ADSI.[1] Диалект ADSI, как видно из программы, использует командную строку, состоящую из четырех аргументов, каждый из которых разделен точкой с запятой.[2] Вот эти аргументы:

- ADsPath (в угловых скобках), определяющий сервер и базовое DN-имя для поиска.
- Фильтр поиска (применяется тот же синтаксис LDAP-фильтров, что упоминался раньше).
- Имя или имена (разделенные запятыми) возвращаемых свойств.
- Пространство поиска: либо Base, либо OneLevel, либо SubTree (в соответствии со стандартом LDAP).

`Execute()` возвращает ссылку на первый из объектов ADO `RecordSet`, получаемых в результате запроса. По очереди запрашивается каждый из объектов `RecordSet`, распаковываются объекты, которые в нем содержатся, и выводится свойство `Value`, возвращаемое методом `Fields()` для каждого из этих объектов. Свойство `Value` содержит значение, которое запрашивалось в командной строке (имя объекта `Group`). Вот как выглядит отрывок получаемых данных на машине с Windows 2000:

```
Administrators
Users
Guests
Backup Operators
Replicator
Server Operators
Account Operators
Print Operators
DHCP Users
DHCP Administrators
Domain Computers
Domain Controllers
Schema Admins
Enterprise Admins
Cert Publishers
Domain Admins
Domain Users
```

[1] Тем, кто знает SQL, первый диалект покажется проще. Диалект SQL предлагает несколько интересных возможностей. Например, MS SQL Server 7 можно настроить так, что он будет знать об ADSI-провайдерах, а не только об обычных базах данных. Это означает, что вы можете выполнять SQL-запросы, которые одновременно обращаются к объектам ActiveDirectory через ADSI.

[2] Будьте внимательны при использовании провайдера ADSI ADO: около символов точки с запятой не может быть никаких пробелов, иначе запрос выполняться не будет.

```
Domain Guests
Group Policy Admins
RAS and IAS Servers
DnsAdmins
DnsUpdateProxy
```

Выполнение распространенных задач при помощи пространства имен WinNT и LDAP

Теперь, когда мы разобрались со «списком» сложностей, можно перейти к выполнению некоторых распространенных задач системного администрирования, используя ADSI из Perl. Цель – дать понять, какие задачи можно решать при помощи представленной информации об ADSI. Затем рассмотреть и использовать код, который пригоден для написания собственных программ.

Для этих целей будет использоваться одно из двух пространств имен. Первое пространство – *WinNT*, которое предоставляет доступ к объектам Windows NT 4.0, таким как пользователи, группы, принтеры, службы и т. д.

Второе – это наш старый знакомый – *LDAP*. *LDAP* мы выбираем провайдером при переходе к Windows 2000 и ее службе Active Directory, основанной на LDAP. Большинство объектов *WinNT* также доступны через *LDAP*. Ведь даже в Windows 2000 существуют задачи, которые можно выполнить, только используя пространство имен WinNT (например, создание учетных записей на локальной машине).

Программы, работающие с этими различными пространствами имен, похожи друг на друга (в конце концов, частично в этом и заключается смысл применения ADSI), но необходимо обратить внимание на два важных различия. Во-первых, формат ADsPath немного отличается. В соответствии с ADSI SDK, ADsPath в WinNT может иметь следующий вид:

```
WinNT://DomainName[/ComputerName[/ObjectName[,className]]]]
WinNT://DomainName[/ObjectName[,className]]]
WinNT://ComputerName,computer]
WinNT:
```

ADsPath в LDAP выглядит так:

```
LDAP://HostName[:PortNumber][/DistinguishedName]
```

Обратите внимание, что при работе с NT 4 ADsPath в LDAP требует указывать имя сервера (в Windows 2000 это изменилось). Это означает, что пространство имен LDAP нельзя просмотреть с верхнего уровня, как пространство WinNT, т. к. необходимо указать начальный сервер. В пространстве имен WinNT любой может применить ADsPath или просто `WinNT:` для начала поиска в иерархии доменов.

Также обратите внимание, что свойства объектов в двух пространствах имен похожи, но не идентичны. Например, можно обратиться к одним и тем же объектам из обоих пространств имен WinNT и LDAP, но обратиться к некоторым свойствам Active Directory конкретного объекта пользователя можно только через пространство имен LDAP.

Особенно важно заметить различия между схемами в этих двух пространствах имен. Например, класс `User` для WinNT не имеет обязательных свойств, тогда как класс `User` в LDAP требует наличия свойств `cn` и `samAccountName` в каждом объекте пользователя.

Не забывая об этих различиях, посмотрим на сам код. В целях экономии места пропустим бо́льшую часть проверок ошибок, но рекомендуется запустить сценарий с ключом *-w* и добавить в текст программы примерно такие строки:

```
die "Ошибка OLE:".Win32::OLE->LastError() if Win32::OLE->LastError();
```

Работа с пользователями через ADSI

Для получения списка пользователей домена применяется следующее:

```
use Win32::OLE 'in';

$ADsPath = "WinNT://DomainName/PDCName,computer";
$c = Win32::OLE->GetObject($ADsPath) or die "Невозможно получить $ADsPath\n";
foreach $adsobj (in $c){
    print $adsobj->{Name},"\n" if ($adsobj->{Class} eq "User");
}
```

Для создания нового пользователя и установки его полного имени (свойство Full Name):

```
use Win32::OLE;

$ADsPath="WinNT://DomainName/ComputerName,computer";
$c = Win32::OLE->GetObject($ADsPath) or die "Невозможно получить $ADsPath\n";

# создаем и возвращаем объект User
$u = $c->Create("user",$username);
$u->SetInfo();  # нужно создать пользователя, перед тем как менять значения

# в пространстве имен WinNT: пробел между "Full" и "Name" недопустим
$u->{FullName} = $fullname;
$su->SetInfo();
```

Если `ComputerName` – это первичный контроллер домена (Primary Domain Controller), то мы создали пользователя домена. Если нет, этот пользователь будет локальным для данной машины.

Эквивалентная программа создания глобального пользователя (при помощи LDAP нельзя создавать локальных пользователей) в Active Directory выглядит так:

```
use Win32::OLE;

$ADsPath = "LDAP://ldapserver,CN=Users,dc=example,dc=com";

$c = Win32::OLE->GetObject($ADsPath) or die "Невозможно получить $ADsPath\n";

# создаем и возвращаем объект User
$u=$c->Create("user","cn=".$commonname);
$u->{samAccountName} = $username;
# нужно сначала создать пользователя в каталоге, а потом менять значения
$u->SetInfo();

# пробел между "Full" и "Name" требуется при работе с пространством имен
LDAP:
$u->{'Full Name'} = $fullname;
$u->SetInfo();
```

Для удаления пользователя нужно внести лишь небольшие изменения:

```
use Win32::OLE;

$ADsPath = "WinNT://DomainName/ComputerName,computer";
$c = Win32::OLE->GetObject($ADsPath) or die "Невозможно получить $ADsPath\n";

# удаляем объект User, заметьте, мы в границах контейнера
$c->Delete("user",$username);
$u->SetInfo();
```

Изменить пароль пользователя можно при помощи единственного метода:

```
use Win32::OLE;

$ADsPath = "WinNT://DomainName/ComputerName/".$username;
$u = Win32::OLE->GetObject($ADsPath) or die "Невозможно получить $ADsPath\n";

$u->ChangePasssword($oldpassword,$newpassword);
$u->SetInfo();
```

Работа с группами через ADSI

Для перечисления доступных групп достаточно лишь немного подправить программу, выводящую список пользователей. Меняется только такая строка:

```
print $adsobj->{Name},"\n" if ($adsobj->{Class} eq "Group");
```

Создание и удаление групп выполняется при помощи тех же методов `Create()` и `Delete()`, которые применялись для создания и удаления учетных записей. Единственное различие – первый аргумент нужно изменить на «group». Вот так:

```
$g = $c->Create("group",$groupname);
```

Для добавления пользователя в группу (определяемую при помощи `GroupName`) после ее создания используется следующее:

```
use Win32::OLE;

$ADsPath = "WinNT://DomainName/GroupName,group";

$g = Win32::OLE->GetObject($ADsPath) or die "Невозможно получить $ADsPath\n";

# используется ADsPath для указанного объекта пользователя
$g->Add($userADsPath);
```

Здесь действуют те же правила относительно локальных пользователей и пользователей домена (глобальных), которые мы рассмотрели выше. Для того чтобы добавить пользователя домена в группу, `$user-ADsPath` должна указывать на пользователя на PDC для этого домена.

Для удаления пользователя из группы применяйте:

```
$c->Remove($userADsPath);
```

Работа с разделяемыми ресурсами через ADSI

Теперь займемся более интересными задачами ADSI, адресованными посвященным. Можно применять ADSI, чтобы предоставить в совместное пользование часть локального дискового пространства на машине:

```
use Win32::OLE;

$ADsPath = "WinNT://ComputerName/lanmanserver";

$c = Win32::OLE->GetObject($ADsPath) or die "Невозможно получить $ADsPath\n";

$s = $c->Create("fileshare",$sharename);
$s->{path}        = 'C:\directory';
$s->{description} = "This is a Perl created share";
$s->SetInfo();
```

Разделяемые ресурсы можно удалять при помощи метода `Delete()`.

Перед тем как перейти к другим задачам, хочу воспользоваться случаем и напомнить вам о необходимости обратиться к документации SDK перед работой с каким-либо из этих ADSI-объектов. Кое-какие неожиданности могут оказаться полезными. Если вы заглянете в раздел *Acti-*

ve Directory Service Interfaces 2.5→ADSI Reference→ ADSI Interfaces→ Persistent Object Interfaces→ IADsFileShare файла помощи ADSI 2.5, то увидите, что объект `fileshare` имеет свойство `CurrentUserCount`, которое соответствует количеству пользователей, подсоединенных в настоящее время к разделяемому ресурсу. Этот нюанс может очень сильно пригодиться.

Работа с очередями и заданиями печати через ADSI

Вот как можно определить названия очередей на определенном сервере и модели принтеров, используемых для обслуживания этих очередей:

```
use Win32::OLE 'in';

$ADsPath="WinNT://DomainName/PrintServerName,computer";

$c = Win32::OLE->GetObject($ADsPath) or die "Невозможно получить $ADsPath\n";

foreach $adsobj (in $c){
    print $adsobj->{Name}.":".$adsobj->{Model}."\n"
      if ($adsobj->{Class} eq "PrintQueue");
}
```

После того как стало известно название очереди печати, можно напрямую связаться с ней для запросов и управления:

```
use Win32::OLE 'in';

# таблица получена из раздела
# 'Active Directory Service Interfaces 2.5->ADSI Reference->
# ADSI Interfaces->Dynamic Object Interfaces->IADsPrintQueueOperations->
# IADsPrintQueueOperations Property Methods' (уф!) из ADSI 2.5 SDK

%status =
  (0x00000001 => 'PAUSED',             0x00000002 => 'PENDING_DELETION',
   0x00000003 => 'ERROR' ,             0x00000004 => 'PAPER_JAM',
   0x00000005 => 'PAPER_OUT',          0x00000006 => 'MANUAL_FEED',
   0x00000007 => 'PAPER_PROBLEM',      0x00000008 => 'OFFLINE',
   0x00000100 => 'IO_ACTIVE',          0x00000200 => 'BUSY',
   0x00000400 => 'PRINTING',           0x00000800 => 'OUTPUT_BIN_FULL',
   0x00001000 => 'NOT_AVAILABLE',      0x00002000 => 'WAITING',
   0x00004000 => 'PROCESSING',         0x00008000 => 'INITIALIZING',
   0x00010000 => 'WARMING_UP',         0x00020000 => 'TONER_LOW',
   0x00040000 => 'NO_TONER',           0x00080000 => 'PAGE_PUNT',
   0x00100000 => 'USER_INTERVENTION', 0x00200000 => 'OUT_OF_MEMORY',
   0x00400000 => 'DOOR_OPEN',          0x00800000 => 'SERVER_UNKNOWN',
   0x01000000 => 'POWER_SAVE');

$ADsPath = "WinNT://PrintServerName/PrintQueueName";

$p = Win32::OLE->GetObject($ADsPath) or die "Невозможно получить $ADsPath\n";
```

```
print "Состояние принтера " . $c->{Name} . " -- " .
   ((exists $p->{status}) ? $status{$c->{status}} : "NOT ACTIVE") . "\n";
```

Объект PrintQueue имеет несколько методов для контроля очереди печати: Pause(), Resume() и Purge(). Это позволяет управлять действиями самой очереди. А что если мы захотим изучить или обработать конкретные задачи из очереди?

Для того чтобы добраться до самих заданий, необходимо вызвать метод PrintJobs() объекта PrintQueue. PrintJobs() возвращает коллекцию, состоящую из объектов PrintJob, каждый из которых имеет ряд свойств и методов. Например, вот как можно показать список заданий из определенной очереди:

```
use Win32::OLE 'in';

# таблица получена из раздела
# 'Active Directory Service Interfaces 2.5->ADSI Reference->
# ADSI Interfaces->Dynamic Object Interfaces->IADsPrintJobOperations->
# IADsPrintJobOperations Property Methods' (двойное уф) в ADSI 2.5 SDK

%status = (0x00000001 => 'PAUSED',  0x00000002 => 'ERROR',
           0x00000004 => 'DELETING',0x00000010 => 'PRINTING',
           0x00000020 => 'OFFLINE', 0x00000040 => 'PAPEROUT',
           0x00000080 => 'PRINTED', 0x00000100 => 'DELETED');

$ADsPath = "WinNT://PrintServerName/PrintQueueName";

$p = Win32::OLE->GetObject($ADsPath) or die "Невозможно получить $ADsPath\n";

$jobs = $p->PrintJobs();
foreach $job (in $jobs){
  print $job->{User} . "\t" . $job->{Description} . "\t" .
        $status{$job->{status}} . "\n";
}
```

Каждое задание можно приостановить (Pause()) и продолжить (Resume()).

Работа со службами NT/2000 через ADSI

В последнем наборе примеров рассмотрим, как находить, запускать и останавливать службы на машине с NT/2000. Как и другие примеры из этой главы, эти короткие программки необходимо запускать с достаточными привилегиями для осуществления выполняемых действий.

Для получения списка служб на машине и их состояний можно использовать такую программу:

```
use Win32::OLE 'in';
```

```
# эта таблица получена из раздела
# 'Active Directory Service Interfaces 2.5->ADSI Reference->
# ADSI Interfaces->Dynamic Object Interfaces->IADsServiceOperations->
# IADsServiceOperations Property Methods' ADSI 2.5 SDK

%status =
  (0x00000001 => 'STOPPED',          0x00000002 => 'START_PENDING',
   0x00000003 => 'STOP_PENDING',     0x00000004 => 'RUNNING',
   0x00000005 => 'CONTINUE_PENDING',0x00000006 => 'PAUSE_PENDING',
   0x00000007 => 'PAUSED',           0x00000008 => 'ERROR');

$ADsPath = "WinNT://DomainName/ComputerName,computer";

$c = Win32::OLE->GetObject($ADsPath) or die "Невозможно получить $ADsPath\n";

foreach $adsobj (in $c){
  print $adsobj->{DisplayName} . ":" . $status{$adsobj->{status}} . "\n"
      if ($adsobj->{Class} eq "Service");
}
```

Для запуска, остановки, приостановки или продолжения работы службы вызываются очевидные методы (`Start()`, `Stop()` и т. д.). Вот как можно запустить службу Network Time на машине с Windows 2000, если ранее она была остановлена:

```
use Win32::OLE;

$ADsPath = "WinNT://DomainName/ComputerName/W32Time,service";

$s = Win32::OLE->GetObject($ADsPath) or die "Невозможно получить $ADsPath\n";

$s->Start();
# можно в этом месте проверять в цикле состояние до тех пор,
# пока служба не будет запущена
```

Во избежание потенциальных конфликтов имен пользователей и компьютеров, можно переписать предыдущий пример:

```
use Win32::OLE;

$d = Win32::OLE->GetObject("WinNT://Domain");
$c = $d->GetObject("Computer", $computername);
$s = $c->GetObject("Service", "W32Time");

$s->Start();
```

Для остановки службы нужно всего лишь изменить последнюю строчку на:

```
$s->Stop();
# можно в этом месте проверять в цикле состояние до тех пор,
# пока служба не будет остановлена
```

Эти примеры должны подсказать вам, какой контроль над системой можно получить при помощи ADSI из Perl. Службы каталогов и их интерфейсы могут быть весьма могущественной частью вашей компьютерной инфраструктуры.

Информация о модулях из этой главы

Название	Идентификатор на CPAN	Версия
`Net::Telnet`	JROGERS	3.01
`Net::Finger`	FIMM	1.05
`Net::Whois`	DHUDES	1.9
`Net::LDAP`	GBARR	0.20
`Mozilla::LDAP`	LEIFHED	1.4
`Sys::Hostname` (входит в состав Perl)		
`Win32::OLE` (входит в состав ActiveState Perl)	JDB	1.11

Рекомендуемая дополнительная литература

Finger

«RFC1288:The Finger User Information Protocol», D. Zimmerman, 1991.

WHOIS

ftp://sipb.mit.edu/pub/whois/whois-servers.list – это список наиболее крупных WHOIS-серверов.

«RFC954:NICNAME/WHOIS», K. Harrenstien, M. Stahl, and E. Feinler, 1985.

LDAP

«An Internet Approach to Directories», Netscape, 1997 – отличное введение в LDAP (*http://developer.netscape.com/docs/manuals/ldap/ldap.html*).

«An LDAP Roadmap & FAQ», Jeff Hodges, 1999 (*http://www.kingsmountain.com/ldapRoadmap.shtml*).

http://www.ogre.com/ldap/ и *http://www.linc-dev.com/* – домашние страницы соавторов PerLDAP.

http://www.openldap.org/ – свободно распространяемый LDAP-сервер, находится в стадии активной разработки.

http://www.umich.edu/~dirsvcs/ldap/index.html – домашняя страница «прародителя» служб каталогов OpenLDAP и Netscape. Некоторая документация представляет интерес до сих пор.

«Implementing LDAP», Mark Wilcox (Wrox Press, 1999).

«LDAP-HOWTO», Mark Grennan, 1999 (*http://www.grennan.com/ldap-HOWTO.html*).

«LDAP Overview Presentation», Bruce Greenblatt, 1999 (*http://www.directory-applications.com/presentation/*).

«LDAP:Programming Directory-Enabled Applications With Lightweight Directory Access Protocol», Tim Howes and Mark Smith (Macmillan Technical Publishing, 1997).

Netscape Directory Server Administrator's/Installation/Deployment Guides and SDK documentation (*http://developer.netscape.com/docs/manuals/directory.html*).

«RFC1823:The LDAP Application Program Interface», T. Howes, M. Smith, 1995.

«RFC2222:Simple Authentication and Security Layer (SASL)», J. Myers, 1997.

«RFC2251:Lightweight Directory Access Protocol (v3)», M. Wahl, T. Howes, S.Kille, 1997.

«RFC2252:Lightweight Directory Access Protocol (v3):Attribute Syntax Definitions», M.Wahl, A. Coulbeck, T. Howes, S. Kille, 1997.

«RFC2254:The String Representation of LDAP Search Filters», T. Howes, 1997.

«RFC2255:The LDAP URL Format», T. Howes, M. Smith, 1997.

«RFC2256:A Summary of the X.500(96) User Schema for use with LDAPv3», M. Wahl, 1997.

«The LDAP Data Interchange Format (LDIF)–Technical Specification» (в состоянии разработки), Gordon Good, 1999 (можно найти на *http://search.ietf.org/internet-drafts/draft-good-ldap-ldif-0X.txt*, где X – это номер текущей версии).

«Understanding and Deploying Ldap Directory Services», Tim Howes, Mark Smith, Gordon Good (Macmillan Technical Publishing, 1998).

«Understanding LDAP», Heinz Jonner, Larry Brown, Franz-Stefan Hinner, Wolfgang Reis, Johan Westman, 1998. Превосходное введение в LDAP (*http://www.redbooks.ibm.com/ abstracts/sg244986.html*).

ADSI

http://cwashington.netreach.net/ – еще один хороший сайт (посвящен не только Perl) по созданию сценариев для ADSI и других технологий от Microsoft.

http://www.microsoft.com/adsi – канонический источник информации по ADSI; обязательно загрузите отсюда ADSI SDK.

http://opensource.activestate.com/authors/tobyeverett – содержит коллекцию документации по использованию ADSI из Perl Тоби Эверета.

http://www.15seconds.com – еще один хороший сайт (посвящен не только Perl) по созданию сценариев для ADSI и других технологий от Microsoft.

«Windows 2000 Active Directory», by Alistair G. Lowe-Norris (O'Reilly, 1999).

7

- *Взаимодействие с SQL-сервером из Perl*
- *Использование DBI*
- *Использование ODBC*
- *Документирование сервера*
- *Учетные записи баз данных*
- *Мониторинг состояния сервера*
- *Информация о модулях из этой главы*
- *Рекомендуемая дополнительная литература*

Администрирование баз данных SQL

С какой стати глава, посвященная администрированию баз данных, находится в книге о системном администрировании? Существует по крайней мере три веские причины, по которым люди, интересующиеся Perl и системным администрированием, должны быть лояльнее к базам данных:

1. Через несколько глав этой книги красной нитью проходит мысль о растущей важности баз данных в современном системном администрировании. Мы создавали (пусть и простые) базы данных, чтобы хранить информацию о пользователях и машинах; но это была лишь «верхушка айсберга». Списки рассылки, файлы паролей и даже реестр Windows NT/2000 – все являлось примерами баз данных, наблюдаемыми практически каждый день. Все крупные пакеты для системного администрирования (например, предлагаемые CA, Tivoli, HP и Microsoft) зависят от баз данных. Если вы собираетесь всерьез заняться системным администрированием, рано или поздно вам придется с ними столкнуться.
2. Управление и работа с базами данных – это «игра в игре» для системного администратора. Берясь за базы данных, помимо прочего, необходимо заниматься:

   Учетными записями/пользователями

   Файлами журналов

   Управлением хранением (пространство на диске и т. д.)

Управлением процессами

Вопросами взаимодействия

Резервными копиями

Безопасностью

Звучит знакомо? Мы можем и даже должны знать обо всем этом.

3. Perl – язык для склейки, и можно доказать, что один из лучших. Много усилий потрачено на интеграцию Perl с базами данных, в основном, благодаря колоссальной энергии, окружающей разработку для Интернета. Подобные достижения следует использовать в собственных интересах. Хотя Perl можно интегрировать с базами данных разных форматов, скажем, Unix DBM, Berkeley DB и т. д., в этой главе внимание будет уделено интерфейсу с масштабными базами данных. Об остальных форматах речь пойдет в других главах.

Системный администратор, разбирающийся в базах данных, должен хоть немного «говорить» на структурированном языке запросов (SQL), «официальном» для большинства коммерческих и нескольких некоммерческих систем. Создание сценариев на Perl для администрирования баз данных тоже требует некоторого знания SQL, т. к. в подобных сценариях встречаются простые встроенные операторы SQL. Достаточную для начала информацию о SQL можно найти в приложении D «Пятнадцатиминутное руководство по SQL». Чтобы не уходить в сторону от вопросов системного администрирования, в примерах из этой главы используются те же базы данных, что и ранее.

Взаимодействие с SQL–сервером из Perl

Существует два стандартных способа взаимодействия с SQL-сервером: DBI (DataBase Interface) и ODBC (Open DataBase Connectivity). Когда-то DBI был стандартом Unix, а ODBC – стандартом Win32, но эти границы начали расплываться после того, как ODBC стал доступным в мире Unix, а DBI был перенесен на Win32. Еще сильнее стирает эти границы пакет `DBD::ODBC` – DBD-модуль, «говорящий» на ODBC из DBI.[1]

DBI и ODBC очень похожи как в своем предназначении, так и в использовании, поэтому рассматриваться они будут одновременно. И DBI, и ODBC можно считать «промежуточным программным обеспече-

[1] Помимо стандартов, обсуждаемых здесь, существует несколько отличных серверов и механизмов, зависящих от операционной системы. Одним из примеров является *Sybperl* Майкла Пеплера (Michael Peppler), предназначенный для взаимодействия Perl и Sybase. Многие из этих нестандартных механизмов вызываются как DBI-модули. Например, большая часть функций *Sybperl* доступна теперь в `DBD::Sybase`. На платформе Win32 все большее распространение получают объекты данных ActiveX (ADO), о которых говорилось в главе 6.

нием» (middleware). Они создают уровень абстракции, позволяющий программисту писать программы, применяя вызовы DBI/ODBC, не имея представления об API какой-либо конкретной базы данных. Передать эти вызовы на уровень, зависящий от баз данных, дело DBI/ODBC. Модуль DBI обращается для этого к драйверу DBD; менеджер ODBC вызывает зависящий от источника данных ODBC драйвер, который заботится обо всех частностях, необходимых для соединения с сервером. На рис. 7.1 показана архитектура DBI и ODBC. В обоих случаях получается по меньшей мере трехъярусная модель:

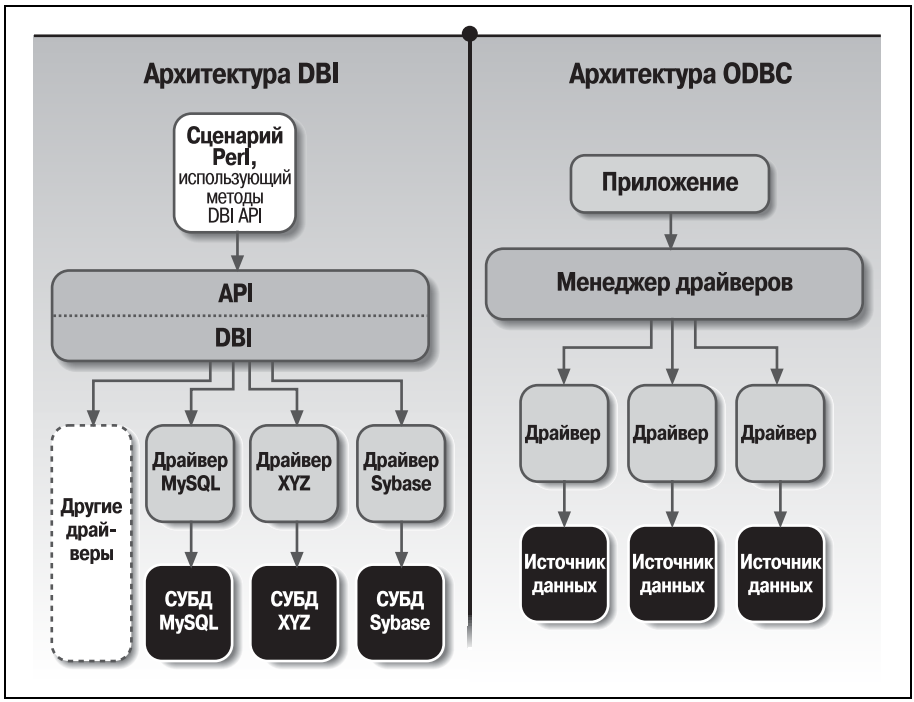

Рис. 7.1. *Архитектура DBI и ODBC*

1. Лежащая в основе система управления базами данных (Oracle, MySQL, Sybase, Microsoft SQL Server и т. д.).

2. Уровень, зависящий от базы данных, который выполняет созданные программистом запросы к серверу. Программисты не работают с этим уровнем напрямую; они используют третий ярус. В DBI с этим уровнем имеет дело специальный модуль DBD. Для взаимодействия с базой данных Oracle будет применяться модуль `DBD::Oracle`. В процессе компилирования модули DBD обычно связываются с клиентской библиотекой данного сервера, поставляемой создателем сервера. В ODBC с этим уровнем работает ODBC-драйвер, зависящий от источника данных и устанавливаемый поставщиком.

Мост над пропастью между базами данных Unix и NT/2000

Очень часто системные администраторы, работающие в многоплатформенном окружении, задают вопрос: «Как я могу использовать Microsoft SQL Server из Unix?» Если центральная система администрирования или наблюдения построена на Unix, то установка нового MS-SQL-сервера представляет собой трудную задачу. Я знаю три способа справиться с этой ситуацией. Второй и третий способы не зависят от SQL-сервера, поэтому даже если вы применяете не Microsoft SQL-сервер, однажды они могут вам пригодиться.

1. Скомпилируйте и используйте `DBD::Sybase`. Модуль `DBD::Sybase` потребует некоторых библиотек для соединения с базой данных. Существует два набора библиотек, а этого более чем достаточно. Первый набор – библиотеки Sybase OpenClient – может быть доступен для вашей платформы (так, они бесплатно распространяются с некоторыми дистрибутивами Linux как часть пакета Sybase Adaptive Server Enterprise). Если вы используете MS-SQL-сервер версии 6.5 или ниже, то `DBD::Sybase`, собранный с этими библиотеками, будет работать. Если это сервер версии 7.0 или выше, для совместимости может понадобиться «файл-заплата» от Microsoft. Информацию о нем следует искать на *http://support.Microsoft.com/support/kb/articles/q239/8/83.asp* (KB статья Q239883). Второй вариант – установить библиотеки FreeTDS, которые можно найти на *http://www.freetds.org*. Изучите инструкции на этом сайте, чтобы собрать нужную версию для своего сервера.
2. Используйте `DBD::Proxy`. Этот модуль DBD входит в состав DBI. Он позволяет на машине с MS-SQL-сервером запустить дополнительный маленький, который будет служить прозрачным прокси-сервером для запросов от Unix-клиентов.
3. Получите и применяйте Unix ODBC из `DBD::ODBC`. Некоторые разработчики, включая MERANT (*http://www.merant.com*) и OpenLink Software (*http://www.openlinksw.com*), продают такое программное обеспечение; но стоит попытаться использовать то, что было создано разработчиками из проекта Open Source. Подробную информацию можно найти на странице *freeODBC* Брайана Джепсона (Brian Jepson) на *http://users.ids.net/~bjepson/freeODBC*. Вам понадобится и драйвер ODBC для Unix-платформы (разработанный производителем базы данных) и менеджер ODBC (подобный `unixODBC` или `iODBC`).

3. Уровень независимого от базы данных интерфейса прикладного программирования (API). Очень скоро мы будем писать сценарии на Perl, взаимодействующие с этим уровнем. В DBI он известен как уровень DBI (т. е. будут выполняться DBI-вызовы). В ODBC обычно происходит взаимодействие с менеджером ODBC-драйверов через вызовы ODBC API.

Прелесть ситуации состоит в том, что код, написанный для DBI или ODBC, можно без осложнений переносить с одного сервера на другой. API-вызовы остаются прежними, поскольку не зависят от используемой базы данных. Эта идея остается справедливой практически во всех случаях программирования баз данных. К сожалению, программы, которые мы будем писать (т. е. предназначенные для администрирования баз данных), обречены зависеть от сервера, т. к. практически нет двух серверов, которые администрировались бы хоть отдаленно похожим образом.[1] Опытные системные администраторы любят переносимые решения, но, увы, их не ожидают.

Впрочем, это грустные размышления, лучше посмотрим, как использовать DBI и ODBC. Обе технологии выполняют одни и те же действия, поэтому может показаться, что в объяснениях или по крайней мере в заголовках присутствует избыточность.

В следующем разделе мы будем исходить из того, что сервер баз данных и необходимые модули Perl уже установлены. В некоторых примерах кода для DBI будет использоваться сервер MySQL; а для ODBC – Microsoft SQL Server.

Использование DBI

Вот что следует делать для использования DBI. Информацию о самом DBI можно найти в книге «Programming the Perl DBI»[2] Аллигатора Декарта (Alligator Descartes) и Тима Банса (Tim Bunce) (O'Reilly).

Шаг 1: Загрузите нужный модуль

Здесь нет ничего особенного, нужно лишь написать:

```
use DBI;
```

Шаг 2: Соединитесь с базой данных и получите дескриптор соединения

Код на Perl, устанавливающий DBI-соединение с базой данных MySQL и возвращающий дескриптор базы данных, выглядит примерно так:

---

[1] MS-SQL первоначально был написан на основе исходного кода Sybase, так что это один из редких примеров обратного.

[2] Аллигатор Декарт, Тим Банс «Программирование на Perl DBI», издательство «Символ-Плюс», 2000 г.

```
# соединиться с базой данных $database, используя заданное имя
# пользователя и пароль, вернуть дескриптор базы данных
$database = "sysadm";
$dbh = DBI->connect("DBI:mysql:$database",$username,$pw);
die "Ошибка! Невозможно соединиться: $DBI::errstr\n" unless (defined
$dbh);
```

До соединения с сервером DBI загрузит низкоуровневый DBD-драйвер (`DBD::mysql`). Перед тем как двигаться дальше, проверим, что соединение (при помощи `connect()`) установлено. DBI предоставляет два параметра для `connect()` — `RaiseError` и `PrintError`, определяющие, будет ли DBI выполнять эту проверку или сообщит об ошибках, когда они возникнут. В частности, если применить:

```
$dbh = DBI->connect("DBI:mysql:$database",
                    $username,$pw,{RaiseError => 1});
```

то DBI вызовет `die`, при условии, что соединение с базой данных не произошло.

Шаг 3: Отправьте команды SQL-серверу

Когда модуль загружен и соединение с сервером баз данных установлено, начинается самое интересное. Пошлем серверу несколько SQL-команд. Мы будем использовать запросы, приведенные в качестве примеров из приложения D. В этих запросах для заключения в кавычки применяется оператор `q` (т. е. `something` записывается как `q{something}`), следовательно, не нужно беспокоиться о кавычках внутри запросов. Вот первый из двух DBI-методов, существующих для отправки команд:

```
$results=$dbh->do(q{UPDATE hosts
                    SET bldg = 'Main'
                    WHERE name = 'bendir'});
die "Невозможно выполнить обновление:$DBI::errstr\n" unless (defined
$results);
```

Переменная `$results` равна либо количеству обновленных записей, либо `undef`, если произошла ошибка. И хотя важно знать, сколько записей было обработано, такой метод абсолютно не подходит для операторов, подобных `SELECT`, где необходимо увидеть сами данные. Следовательно, нужно применить второй метод.

Для использования второго метода необходимо сначала подготовить оператор SQL (с помощью команды `prepare`), а затем выполнить (`execute`) его на сервере. Вот пример:

```
$sth = $dbh->prepare(q{SELECT * from hosts}) or
  die "Ошибка! Невозможно подготовить запрос:".$dbh->errstr."\n";
$rc = $sth->execute or
  die "Ошибка! Невозможно выполнить запрос:".$dbh->errstr."\n";
```

Команда prepare() возвращает нечто новое, чего мы раньше никогда не встречали: дескриптор команды. Так же, как дескриптор базы данных ссылается на открытое соединение с базой данных, дескриптор команды ссылается на конкретный SQL-оператор, который был подготовлен с помощью prepare(). Получив дескриптор команды, можно использовать execute для отправки запроса на сервер. Позже подобный дескриптор команды будет применяться для получения результатов запроса.

Вероятно, вас удивляет, зачем сначала подготавливать оператор, вместо того чтобы напрямую выполнить его. Подобная операция позволяет драйверу DBD (или, что более вероятно, вызываемой библиотеке) выполнить синтаксический разбор SQL-запроса. После того как оператор подготовлен, можно повторно выполнить его через дескриптор команды, не проводя больше синтаксический анализ. Зачастую это основной фактор повышения эффективности. В действительности, DBI-метод do(), используемый по умолчанию, для каждой выполняемой команды применяет сначала prepare(), а затем execute().

Как и вызов do, рассмотренный ранее, метод execute() возвращает количество обработанных записей. Если результатом запроса является ноль записей, возвращается строка 0E0, эта строка при логической проверке воспримется как «истина». Если драйверу неизвестно, сколько записей было обработано, возвращается -1.

Перед тем как перейти к ODBC, стоит упомянуть еще об одной особенности prepare(), поддерживаемой большинством DBD-модулей, а именно о заполнителях (placeholders). Заполнители, также называемые позиционными маркерами, позволяют подготовить команду с оставленными в ней «дырами», в которые можно внести значения во время выполнения команды (execute()). Подобное свойство помогает конструировать запросы на лету, не затрачивая лишнего времени на их синтаксический разбор. Знак вопроса используется как заполнитель для одного скалярного значения. Вот пример кода на Perl, демонстрирующий такое применение заполнителей:

```
@machines = qw(bendir shimmer sander);
$sth = $dbh->prepare(q{SELECT name, ipaddr FROM hosts WHERE name = ?});
foreach $name (@machines){
  $sth->execute($name);
  сделать-что-то-с-результатами
}
```

Каждый раз, проходя через цикл foreach, запрос SELECT выполняется с разными условиями WHERE. Можно применять и несколько заполнителей:

```
$sth->prepare(
   q{SELECT name, ipaddr FROM hosts
```

```
        WHERE (name = ? AND bldg = ? AND dept = ?)});
$sth->execute($name,$bldg,$dept);
```

Теперь, когда известно, как получить количество записей, обрабатываемых запросом, отличным от SELECT, посмотрим, как получить результаты запросов SELECT.

Шаг 4: Получите результаты запроса SELECT

Механизм, рассмотренный здесь, похож на историю о курсорах из приложения D. Когда серверу при помощи execute() посылается запрос SELECT, применяется способ, позволяющий возвращать результаты построчно.

Для получения данных в DBI используется один из методов, перечисленных в табл. 7.1.

Таблица 7.1. DBI-методы получения данных

Имя	Возвращает	Возвращает, если больше нет записей
fetchrow_arrayref()	Ссылка на анонимный массив со значениями, являющимися полями следующей записи	undef
fetchrow_array()	Массив со значениями, являющимися полями следующей записи	Пустой список
fetchrow_hashref()	Ссылка на анонимный хэш, ключами которого являются имена полей, а значениями – значения полей следующей записи	undef
fetchall_arrayref()	Ссылка на массив массивов	Ссылка на пустой массив

Посмотрим, как работают эти методы в нашем случае. Для каждого из примеров будем считать, что перед вызовом метода выполнено следующее:

```
$sth = $dbh->prepare(q{SELECT name,ipaddr,dept from hosts}) or
  die "Невозможно подготовить запрос: ".$dbh->errstr."\n";
$sth->execute or die "Невозможно выполнить запрос: ".$dbh->errstr."\n";
```

Вот метод fetchrow_arrayref() в действии:

```
while ($aref = $sth->fetchrow_arrayref){
  print "name: "    .  $aref->[0] . "\n";
  print "ipaddr: " .  $aref->[1] . "\n";
  print "dept: "    .  $aref->[2] . "\n";
}
```

В документации по DBI говорится, что fetchrow_hashref() менее эффективен, чем fetchrow_arrayref(), из-за дополнительной обработ-

ки, но данный метод позволяет получить более читаемый код. Вот пример:

```
while ($href = $sth->fetchrow_hashref){
  print "name: "    .  $href->{name}   . "\n";
  print "ipaddr: " .  $href->{ipaddr}. "\n";
  print "dept: "    .  $href->{dept}   . "\n";
}
```

А теперь рассмотрим «удобный» метод `fetchall_arrayref()`. Он загружает весь полученный результат в одну структуру данных, возвращая ссылку на массив ссылок. При использовании данного метода будьте осторожны и учитывайте размер запросов, поскольку результаты целиком считываются в память. Если размер результата равен 100 Гбайт, это может вызвать проблемы.

Каждая ссылка выглядит точно так же, как и результат вызова метода `fetchrow_arrayref()` (рис. 7.2).

Вот какой код выводит результат запроса целиком:

```
$aref_aref = $sth->fetchall_arrayref;
foreach $rowref (@$aref_aref){
  print "name: "    .  $rowref->[0] . "\n";
  print "ipaddr: " .  $rowref->[1] . "\n";
  print "dept: "    .  $rowref->[2] . "\n";
  print '-'x30,"\n";
}
```

Этот пример применим только к конкретному набору данных, поскольку предполагается, что количество полей фиксировано и они следуют в определенном порядке. Например, считается, что имя машины возвращается в первом поле запроса (`$rowref->[0]`).

Можно переписать предыдущий пример и сделать его более общим, если использовать атрибуты (часто называемые метаданными) дескриптора команды. В частности, если после запроса посмореть на `$sth->{NUM_OF_FIELDS}`, то можно узнать количество полей в получен-

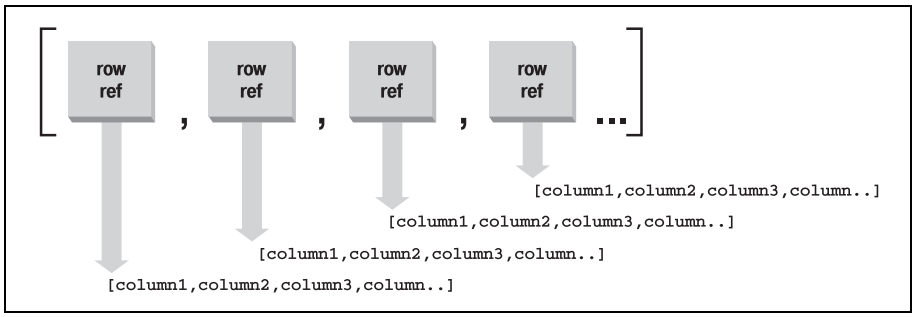

***Рис. 7.2.** Структура данных, возвращаемая fetchrow_arrayref*

ных данных. $sth->{NAME} содержит ссылку на массив названий полей. Последний пример можно переписать так:

```
$aref_aref = $sth->fetchall_arrayref;
foreach $rowref (@$aref_aref){
  for ($i=0; $i < $sth->{NUM_OF_FIELDS};$i++){
    print $sth->{NAME}->[$i].": ".$rowref->[$i]."\n";
  }
  print '-'x30,"\n";
}
```

Обязательно изучите документацию по DBI, чтобы подробно узнать об остальных метаданных.

Шаг 5: Закройте соединение с сервером

С DBI это очень просто сделать:

```
# сообщаем серверу, что данные из дескриптора команды больше
# не нужны (необязательно, т. к. мы собираемся завершить
# соединение)
$sth->finish;
# разорвать соединение дескриптора с базой данных
$dbh->disconnect;
```

Что еще надо сказать о DBI

Осталось еще две темы, которые стоит обсудить, прежде чем переходить к ODBC. Во-первых, это набор методов, которые я называю «удобными» (shortcut methods). Методы, приведенные в табл. 7.2, объединяют перечисленные шаги 3 и 4.

Таблица 7.2. Удобные методы DBI

Название	Объединяет в себе следующие методы
selectrow_arrayref($stmnt)	prepare($stmnt), execute(), fetchrow_arrayref()
selectcol_arrayref($stmnt)	prepare($stmnt), execute(), (@{fetchrow_arrayref()})[0] (т. е. возвращает первое поле для каждой записи)
selectrow_array($stmnt)	prepare($stmnt), execute(), fetchrow_array()

Во-вторых, заслуживает внимания способность DBI связывать переменные с результатами запроса. Методы bind_col() и bind_columns() используются для автоматического помещения результатов запроса в указанную переменную или список переменных. Обычно это заменяет дополнительный шаг, а то и два при написании программы. Ниже приведен пример, включающий bind_columns():

```
$sth = $dbh->prepare(q{SELECT name,ipaddr,dept from hosts}) or
  die "Невозможно подготовить запрос:".$dbh->errstr".\n";
$rc = $sth->execute or
```

```
  die "Невозможно выполнить запрос:".$dbh->errstr"."\n";

# эти переменные получат 1-й, 2-й и 3-й столбцы из SELECT
$rc = $sth->bind_columns(\$name,\$ipaddr,\$dept);

while ($sth->fetchrow_arrayref){
    # $name, $ipaddr и $dept автоматически получают значения из
    # результатов запроса
    сделать-что-то-с-результатами
}
```

Использование ODBC

Основные шаги при использовании ODBC похожи на только что рассмотренные действия с DBI.

Шаг 1: Загрузите нужный модуль Perl

```
use Win32::ODBC;
```

Шаг 2: Соединитесь с базой данных и получите дескриптор соединения

Перед тем как установить соединение в ODBC, следует выполнить один дополнительный шаг. Нужно создать *имя источника данных* (*Data Source Name, DSN*). DSN – это именованная ссылка, в которой хранится конфигурационная информация (например, имя сервера и базы данных), необходимая для доступа к источнику информации, такому как SQL-сервер. Имена источников данных бывают двух типов: *пользовательские* (*user*) и *системные* (*system*), различающиеся тем, будет ли соединение доступно одному пользователю на машине либо любому пользователю или службе.[1]

DSN можно создать либо из панели управления ODBC в Windows NT/2000, либо программным образом из Perl. Мы пойдем вторым путем, хотя бы для того, чтобы не вызывать насмешек со стороны пользователей Unix. Вот как можно поступить для создания пользовательского имени источника данных на сервере MS-SQL:

```
# создаем пользовательское имя источника данных на Microsoft-SQL-сервере
# Замечание: чтобы создать системное имя источника данных,
# замените ODBC_ADD_DSN на ODBC_ADD_SYS_DSN
if (Win32::ODBC::ConfigDSN(
                    ODBC_ADD_DSN,
                    "SQL Server",
                    ("DSN=PerlSysAdm",
                     "DESCRIPTION=DSN for PerlSysAdm",
```

[1] Существует еще и третий тип – *файловый* (*file*), который записывает конфигурационную информацию DSN в файл, который могут использовать несколько машин. Рассматриваемый модуль `Win32::ODBC` не содержит методов, позволяющих создать DSN такого типа.

```
                    "SERVER=mssql.happy.edu",   # имя сервера
                    "ADDRESS=192.168.1.4",      # IP-адрес сервера
                    "DATABASE=sysadm",          # наша база данных
                   "NETWORK=DBMSSOCN",         # Библиотека сокетов TCP/IP
                   ))){
                              print "DSN создан\n";
                            }
else {
   die "Невозможно создать DSN:" . Win32::ODBC::Error() . "\n";
}
```

После того как имя источника данных создано, его можно использовать для открытия соединения с базой данных:

```
# соединяемся с DSN, возвращаем дескриптор базы данных
$dbh=new Win32::ODBC("DSN=PerlSysAdm;UID=$username;PWD=$pw;");
die "Невозможно соединиться с DSN PerlSysAdm:" . Win32::ODBC::Error() .
"\n"
  unless (defined $dbh);
```

Шаг 3: Отправьте команды SQL на сервер

ODBC-эквивалент DBI-командам `do()`, `prepare()` и `execute()` немного проще, потому что модуль `Win32::ODBC` имеет один метод `Sql()` для отправки команды на сервер. Хотя в ODBC теоретически упоминается о подготовке команды и заполнителях, это не реализовано в текущей версии модуля `Win32::ODBC`.[1] В `Win32::ODBC` также не используются дескрипторы команд; взаимодействие происходит через дескриптор базы данных, открытый ранее при помощи метода `new`. Так что нам остается команда с простейшей структурой:

```
$rc = $dbh->Sql(q{SELECT * from hosts});
```

Есть разница между методами ODBC и DBI: в отличие от `do()` из DBI, `Sql()` возвращает `undef`, если запрос был завершен *успешно*, и некоторое ненулевое значение, если запрос не был выполнен.

Если необходимо узнать, сколько записей было обработано в результате запроса `INSERT`, `DELETE` или `UPDATE`, следует использовать метод `RowCount()`. В документации по `Win32::ODBC` сказано, что этот вызов реализован не во всех драйверах ODBC (либо реализован не для всех операторов SQL), поэтому не стоит слепо полагаться на драйвер, лучше сначала все проверить. Как и в случае с методом `execute()`

[1] В то время, когда писалась эта книга, Дейв Рот тестировал новую версию `Win32::ODBC`, позволяющую связывать параметры. Применяемый в этом случае синтаксис похож на DBI (т. е. сначала выполняется `Prepare()`, а затем `Sql()`), но включает некоторые особенности ODBC. Подробную информацию можно найти на *http://www.roth.net*.

из DBI, RowCount() вернет -1, если драйверу недоступна информация о количестве полученных записей.

Вот как выглядит пример использования do() из предыдущего раздела для ODBC:

```
if (defined $dbh->Sql(q{UPDATE hosts
                        SET bldg = 'Main'
                        WHERE name = 'bendir'})){
  die "Невозможно выполнить обновление: ".Win32::ODBC::Error()."\n"
}
else {
  $results = $dbh->RowCount();
}
```

Шаг 4: Получите результаты запроса SELECT

Получение результатов запроса SELECT для ODBC выполняется подобно тому, как это было сделано для DBI, но с одним отличием. Во-первых, *получение* данных с сервера и *обращение* к ним являются двумя разными шагами в Win32::ODBC. Метод FetchRow() получает следующую запись, возвращает 1, если все прошло успешно, и undef, если что-то было не так. Когда запись получена, можно выбрать один из двух методов, чтобы обратиться к ней.

Метод Data() возвращает список полученных полей, если вызывается в списочном контексте, и все поля, склеенные вместе, если вызывается в скалярном контексте. Data() может принимать в качестве необязательного аргумента список, определяющий, какие поля возвращать и в каком порядке (иначе, в соответствии с документацией, они возвращаются в «неопределенном» порядке).

Метод DataHash() возвращает хэш, ключами которого являются имена полей. Это очень похоже на DBI-метод fetchrow_hashref(), с тем исключением, что он возвращает хэш, а не ссылку на хэш. Как и Data(), DataHash() тоже может принимать список в качестве необязательного аргумента для определения того, какие поля возвращать.

В нашем случае эти методы выглядят так:

```
if ($dbh->FetchRow()){
  @ar = $dbh->Data();
  сделать-что-то-со-значениями @ar
}
и:
if ($dbh->FetchRow()){
   %ha = $dbh->DataHash('name','ipaddr');
 сделать-что-то-со-значениями-$ha{name}-и-$ha{ipaddr}
}
```

Справедливости ради надо отметить, что информацию, передаваемую через атрибут дескриптора команды {NAME} в DBI, в мире Win32::ODBC можно получить при помощи метода FieldNames(). Для то-

го чтобы узнать количество полей (как в {NUM_OF_FIELDS}), придется пересчитать элементы в списке, возвращенном методом FieldNames().

Шаг 5: Закройте соединение с сервером

```
$dbh->close();
```

Если вы создали имя источника данных и хотите удалить его, чтобы убрать за собой, используйте оператор, похожий на тот, который применялся для его создания:

```
# замените ODBC_REMOVE_DSN на ODBC_REMOVE_SYS_DSN, если вы
# создавали системное имя источника данных
if (Win32::ODBC::ConfigDSN(ODBC_REMOVE_DSN,
                           "SQL Server","DSN=PerlSysAdm")){
  print "DSN удален\n";
}
else {
  die "Невозможно удалить DSN:".Win32::ODBC::Error()."\n";
}
```

Теперь известно, как работать с базами данных из Perl, используя DBI и ODBC. Применим эти знания на практике и поработаем с более сложными примерами из области администрирования баз данных.

Документирование сервера

Очень много времени и энергии уходит на настройку SQL-сервера и объектов, расположенных на нем. Способ документирования подобной информации может пригодиться в ряде ситуаций. Если база данных будет повреждена, а резервной копии не окажется, вам придется восстанавливать все ее таблицы. При необходимости перенести данные с одного сервера на другой важно знать конфигурацию источника и приемника данных. Даже для собственных баз данных может пригодиться возможность просмотреть карту таблиц.

Чтобы передать всю «прелесть» непереносимой (nonportable) природы администрирования баз данных, приведу пример реализации одной простой задачи для трех различных SQL-серверов с использованием как DBI, так и ODBC. Каждая из этих программ делает одно и то же: выводит список всех баз данных на сервере, их таблицы и структуру каждой таблицы. Эти сценарии можно очень легко расширить для предоставления более подробной информации о каждом объекте. Например, бывает полезно узнать, в каких полях есть значения NULL или NOT NULL. Вывод всех трех программ выглядит одинаково:

```
---sysadm---
        hosts
                name [char(30)]
                ipaddr [char(15)]
```

```
                aliases [char(50)]
                owner [char(40)]
                dept [char(15)]
                bldg [char(10)]
                room [char(4)]
                manuf [char(10)]
                model [char(10)]
---hpotter---
        customers
                cid [char(4)]
                cname [varchar(13)]
                city [varchar(20)]
                discnt [real(7)]
        agents
                aid [char(3)]
                aname [varchar(13)]
                city [varchar(20)]
                percent [int(10)]
        products
                pid [char(3)]
                pname [varchar(13)]
                city [varchar(20)]
                quantity [int(10)]
                price [real(7)]
        orders
                ordno [int(10)]
                month [char(3)]
                cid [char(4)]
                aid [char(3)]
                pid [char(3)]
                qty [int(10)]
                dollars [real(7)]
...
```

Сервер MySQL и DBI

Вот как выглядит способ получить эту информацию с сервера MySQL с использованием DBI. Существующее в MySQL дополнение команды *SHOW* очень упрощает эту задачу:

```
use DBI;

print "Введите имя пользователя: ";
chomp($user = <STDIN>);
print "Введите пароль для $user: ";
chomp($pw = <STDIN>);

$start = "mysql"; # первоначально будем подсоединяться к этой базе данных

# соединяемся с базой данных
$dbh = DBI->connect("DBI:mysql:$start",$user,$pw);
die "Невозможно соединиться: ".$DBI::errstr."\n" unless (defined $dbh);
```

```
# ищем базы данных на сервере
$sth=$dbh->prepare(q{SHOW DATABASES}) or
  die "Невозможно подготовить запрос show databases: ". $dbh->errstr."\n";
$sth->execute or
  die "Невозможно выполнить запрос show databases: ". $dbh->errstr."\n";
while ($aref = $sth->fetchrow_arrayref) {
    push(@dbs,$aref->[0]);
}
$sth->finish;

# ищем таблицы в каждой базе данных
foreach $db (@dbs) {
    print "---$db---\n";

    $sth=$dbh->prepare(qq{SHOW TABLES FROM $db}) or
     die "Невозможно подготовить запрос show tables: ". $dbh->errstr."\n";
    $sth->execute or
      die "Невозможно выполнить запрос show tables: ". $dbh->errstr."\n";

    @tables=();
    while ($aref = $sth->fetchrow_arrayref) {
       push(@tables,$aref->[0]);
    }

    $sth->finish;

   # ищем информацию о полях для каждой таблицы
   foreach $table (@tables) {
       print "\t$table\n";

       $sth=$dbh->prepare(qq{SHOW COLUMNS FROM $table FROM $db}) or
          die "Невозможно подготовить запрос show columns: ". $dbh-
>errstr."\n";
       $sth->execute or
         die "Невозможно выполнить запрос show columns: ". $dbh->errstr."\n";

       while ($aref = $sth->fetchrow_arrayref) {
           print "\t\t",$aref->[0]," [",$aref->[1],"]\n";
       }

       $sth->finish;
   }
}
$dbh->disconnect;
```

Добавим несколько комментариев к приведенной программе:

- Мы соединяемся с начальной базой данных только для того, чтобы удовлетворить принятой в DBI семантике соединения, но такой контекст возникает не обязательно благодаря командам SHOW. В следующих двух примерах этого не будет.

- Если читатель думает, что команды подготовки и выполнения запросов `SHOW TABLES` и `SHOW COLUMNS` являются отличными кандидатами на использование заполнителей, то он совершенно прав. К сожалению, именно эта комбинация DBD драйвера/сервера не поддерживает заполнители в таком контексте (по крайней мере, это было так в период написания данной книги). В следующем примере мы столкнемся с подобной ситуацией.
- Имя пользователя базы данных и его пароль запрашиваются интерактивно, поскольку альтернативы (прописывание их в коде или передача в командной строке, при которых любой, кто просматривает список процессов, сможет их увидеть) еще хуже. В данном случае символы пароля будут отображаться при вводе. Для большей осторожности стоит применять что-то подобное `Term::Readkey`, чтобы подавить отображение символов.

Сервер Sybase и DBI

В этом подразделе представлен аналог для Sybase. Внимательно просмотрите программу, а после этого поговорим о некоторых существенных моментах:

```
use DBI;

print "Введите имя пользователя: ";
chomp($user = <STDIN>);
print "Введите пароль для $user: ";
chomp($pw = <STDIN>);

$dbh = DBI->connect('dbi:Sybase:',$user,$pw);
die "Невозможно соединиться: $DBI::errstr\n"
  unless (defined $dbh);

# ищем базы данных на сервере
$sth = $dbh->prepare(q{SELECT name from master.dbo.sysdatabases}) or
  die "Невозможно подготовить запрос к sysdatabases: ".$dbh->errstr."\n";
$sth->execute or
  die "Невозможно выполнить запрос к sysdatabases: ".$dbh->errstr."\n";

while ($aref = $sth->fetchrow_arrayref) {
    push(@dbs, $aref->[0]);
}
$sth->finish;

foreach $db (@dbs) {
    $dbh->do("USE $db") or
      die "Невозможно использовать $db: ".$dbh->errstr."\n";
    print "---$db---\n";

    # найти таблицы в каждой базе данных
    $sth=$dbh->prepare(q{SELECT name FROM sysobjects WHERE type="U"}) or
      die "Невозможно подготовить запрос к sysobjects: ".$dbh->errstr."\n";
```

```
    $sth->execute or
      die "Невозможно выполнить запрос к sysobjects: ".$dbh->errstr."\n";

    @tables=();
    while ($aref = $sth->fetchrow_arrayref) {
        push(@tables,$aref->[0]);
    }
    $sth->finish;

    # мы должны быть "внутри" базы данных для следующего шага
    $dbh->do("use $db") or
          die "Невозможно изменить $db: ".$dbh->errstr."\n";

    # ищем поля для каждой таблицы
    foreach $table (@tables) {
       print "\t$table\n";

       $sth=$dbh->prepare(qq{EXEC sp_columns $table}) or
          die "Невозможно подготовить запрос sp_columns: ".$dbh->errstr."\n";
       $sth->execute or
          die "Невозможно выполнить запрос sp_columns: ".$dbh->errstr."\n";

       while ($aref = $sth->fetchrow_arrayref) {
           print "\t\t",$aref->[3]," [",$aref->[5],"(",
                  $aref->[6],")]\n";
       }
       $sth->finish;
    }
}
$dbh->disconnect or
  warn "Невозможно отсоединиться: ".$dbh->errstr."\n";
```

Вот обещанные заметные моменты:

- Sybase хранит информацию о базах данных и таблицах в специальных системных таблицах *sysdatabases* и *sysobjects*. Каждая база данных содержит таблицу *sysobjects*, в то время как сервер хранит обобщенную информацию о них в одной таблице *sysdatabases*, расположенной в основной базе данных. Мы используем более явный синтаксис *databases.owner.table* в первом запросе SELECT, чтобы недвусмысленно обратиться именно к этой таблице. Для перехода к *sysobjects* каждой базы данных можно применять этот же синтаксис, вместо того чтобы явно переключаться между ними при помощи USE. Более того, как и при переходе в каталог средствами *cd*, такой контекст упрощает написание других запросов.
- Запрос SELECT к *sysobjects* применяет ключевое слово WHERE, чтобы вернуть информацию только о пользовательских таблицах. Это было сделано для ограничения размера вывода. Если бы мы хотели включить также и все системные таблицы, то могли бы изменить запрос на:

```
WHERE type="U" AND type="S"
```

- Складывается впечатление, что заполнители в `DBD::Sybase` реализованы так для того, чтобы препятствовать их употреблению с хранимыми процедурами. Будь реализация другой, следовало бы использовать заполнители в `EXEC sp_columns`.

Сервер MS-SQL и ODBC

Наконец, вот код для получения той же информации с сервера MS-SQL через ODBC. Заметьте, что применяемый синтаксис SQL практически идентичен предыдущему примеру благодаря связи Sybase/MS-SQL. Интересны отличия между этим примером и предыдущим:

- Использование DSN, которое предоставляет нам контекст базы данных по умолчанию, так что нет необходимости указывать, где искать таблицу *sysdatabases*.
- Употребление `$dbh->DropCursor()` в качестве грубой аналогии `$sth->finish`.
- Неудобный синтаксис, который приходится применять для запуска хранимых процедур. Информацию о хранимых процедурах и других подобных аномалиях ищите на веб-страницах по `Win32::ODBC`.

Вот как выглядит программа:

```
use Win32::ODBC;

print "Введите имя пользователя: ";
chomp($user = <STDIN>);
print "Введите пароль для $user: ";
chomp($pw = <STDIN>);

$dsn="sysadm"; # имя источника данных, которое мы используем

# ищем доступные DSN, создаем переменную $dsn, если она еще не существует
die "Невозможно запросить доступные DSN".Win32::ODBC::Error()."\n"
  unless (%dsnavail = Win32::ODBC::DataSources());
if (!defined $dsnavail{$dsn}) {
    die "невозможно создать DSN:".Win32::ODBC::Error()."\n"
      unless (Win32::ODBC::ConfigDSN(ODBC_ADD_DSN,
              "SQL Server",
              ("DSN=$dsn",
               "DESCRIPTION=DSN for PerlSysAdm",
               "SERVER=mssql.happy.edu",
               "DATABASE=master",
               "NETWORK=DBMSSOCN", # библиотека сокетов TCP/IP
              )));
}

# соединение с основной базой данных
$dbh = new Win32::ODBC("DSN=$dsn;UID=$user;PWD=$pw;");
die "Невозможно соединиться с DSN $dsn:".Win32::ODBC::Error()."\n"
```

```
    unless (defined $dbh);

# ищем базы данных на сервере
if (defined $dbh->Sql(q{SELECT name from sysdatabases})){
    die "Невозможно послать запрос к базе данных:".Win32::ODBC::Error()."\n";
}

while ($dbh->FetchRow()){
    push(@dbs, $dbh->Data("name"));
}
$dbh->DropCursor();

# ищем пользовательские таблицы в каждой базе данных
foreach $db (@dbs) {
    if (defined $dbh->Sql("use $db")){
        die "Невозможно изменить базу данных на $db:" .
            Win32::ODBC::Error() . "\n";
    }
    print "---$db---\n";
    @tables=();
    if (defined $dbh->Sql(q{SELECT name from sysobjects
                            WHERE type="U"})){
       die "Невозможно запросить таблицы из $db:" .
            Win32::ODBC::Error() . "\n";
    }
    while ($dbh->FetchRow()) {
       push(@tables,$dbh->Data("name"));
    }
    $dbh->DropCursor();

    # ищем информацию о полях для каждой таблицы
    foreach $table (@tables) {
       print "\t$table\n";
       if (defined $dbh->Sql(" {call sp_columns (\'$table\')} ")){
           die "Невозможно запросить поля из таблицы
                $table:".Win32::ODBC::Error() . "\n";
       }
       while ($dbh->FetchRow()) {
           @cols=();
           @cols=$dbh->Data("COLUMN_NAME","TYPE_NAME","PRECISION");
           print "\t\t",$cols[0]," [",$cols[1],"(",$cols[2],")]\n";
       }
       $dbh->DropCursor();
    }
}
$dbh->Close();

die "Невозможно удалить DSN:".Win32::ODBC::Error()."\n"
    unless (Win32::ODBC::ConfigDSN(ODBC_REMOVE_DSN,
                                   "SQL Server","DSN=$dsn"));
```

Учетные записи баз данных

Как упоминалось раньше, администраторы баз данных вынуждены иметь дело с рядом тех же вопросов, с которыми борются системные администраторы, в частности, с поддержкой регистрационных имен и учетных записей. Например, днем на работе мы ведем занятия по программированию баз данных. Каждый студент, посещающий эти занятия, получает регистрационное имя на сервере Sybase и свою собственную (пусть и маленькую) базу данных, с которой он может работать. Вот упрощенная версия программы, которую мы используем для создания таких баз данных и регистрационных имен:

```
use DBI;

# ИСПОЛЬЗОВАНИЕ: syaccreate <username>

$admin = 'sa';
print "Введите пароль для $admin: ";
chomp($pw = <STDIN>);
$user=$ARGV[0];

# генерируем *фиктивный* пароль, основываясь на имени
# пользователя, записанном в обратном порядке, и дополняем его
# дефисами, чтобы его длина составляла 6 символов
$genpass = reverse join('',reverse split(//,$user));
$genpass .= "-" x (6-length($genpass));

# вот перечень SQL-команд, которые используем
# мы: 1) создаем базу данных на устройстве USER_DISK
#        с журналом регистрации в USER_LOG
#     2) добавляем регистрационное имя для пользователя
#        на сервере, делая новую базу базой по умолчанию
#     3) переключаемся на вновь созданную базу данных
#     4) изменяем владельца базы данных на пользователя
@commands = ("create database $user on USER_DISK=5 log on USER_LOG=5",
             "sp_addlogin $user,\"$genpass\",$user",
             "use $user",
             "sp_changedbowner $user");

# соединяемся с сервером
$dbh = DBI->connect('dbi:Sybase:',$admin,$pw);
die "Невозможно соединиться: $DBI::errstr\n"
  unless (defined $dbh);

# обходим в цикле массив команд и выполняем последовательно все команды
for (@commands) {
    $dbh->do($_) or die "Не могу $_: " . $dbh->errstr . "\n";
}

$dbh->disconnect;
```

Поскольку эта задача заключается в выполнении ряда команд, которые не возвращают данных, можно записать их в компактном цикле, в котором вызывается повторно `$dbh->do()`. Можно было бы использовать полностью идентичный сценарий для удаления этих учетных записей и баз данных, когда занятия завершатся:

```
use DBI;

# ИСПОЛЬЗОВАНИЕ: syacdelete <username>

$admin = 'sa';
print "Введите пароль для $admin: ";
chomp($pw = <STDIN>);
$user=$ARGV[0];

# перечень SQL-команд, которые мы будем использовать;
# мы: удаляем базу данных пользователя
#     удаляем регистрационное имя с сервера
@commands = ("drop database $user",
             "sp_droplogin $user");

# соединяемся с сервером
$dbh = DBI->connect('dbi:Sybase:',$admin,$pw);
die "Невозможно соединиться: $DBI::errstr\n"
  unless(defined $dbh);

# обходим в цикле массив команд, выполняя их последовательно
for (@commands) {
    $dbh->do($_) or die "Невозможно $_: " . $dbh->errstr . "\n";
}

$dbh->disconnect or
  warn "Невозможно рассоединиться: " . $dbh->errstr . "\n";
```

Существует много функций, связанных с учетными записями, которые можно добавить в эту программу. Вот лишь некоторые идеи:

Проверка паролей

Утилита соединяется с сервером и получает список баз данных и учетных записей; затем пытается установить связь, применяя ненадежные пароли (регистрационные имена, пустые пароли, пароли по умолчанию).

Карта пользователей

Создаем список регистрационных имен и баз данных, доступных пользователям с этими регистрационными именами.

Управление паролями

Система ограничения срока действия паролей.

Мониторинг состояния сервера

В качестве последнего примера рассмотрим несколько способов наблюдения за состоянием SQL-сервера. Программы такого рода по своей природе похожи на службы наблюдения за сетью, уже рассмотренные в главе 5 «Службы имен TCP/IP».

Наблюдение за свободным пространством

Если на мгновение вникнуть в технические тонкости, то сервер баз данных – это место для хранения разного добра. И если места для его хранения не остается, то это либо плохо, либо очень плохо. Так что программы, помогающие следить за свободным и занятым пространством на сервере, действительно очень полезны. Посмотрим на DBI-программу, созданную для наблюдения за дисковым пространством на сервере Sybase.

Вот отрывок вывода программы, которая в графическом виде показывает, как любая база данных использует место на сервере. В каждом разделе отражено, сколько процентов пространства занято данными и журналом. В следующей диаграмме `d` соответствует пространству, занятому данными, а `l` – журналам. Для каждой диаграммы указан объем занятого и доступного пространства (в процентах):

```
                |ddddddd                              |15.23%/5MB
hpotter---------|                                     |
                |                                     |0.90%/5MB

                |ddddddd                              |15.23%/5MB
dumbledore------|                                     |
                |                                     |1.52%/5MB

                |dddddddd                             |16.48%/5MB
hgranger--------|                                     |
                |                                     |1.52%/5MB

                |ddddddd                              |15.23%/5MB
rweasley--------|                                     |
                |l                                    |3.40%/5MB

                |ddddddddddddddddddddddddddd          |54.39%/2MB
hagrid----------|                                     |
                |- no log                             |
```

Вот как генерировался этот вывод:

```
use DBI;

$admin = 'sa';
print "Введите пароль для $admin: ";
chomp($pw = <STDIN>);
$pages = 2; # данные хранятся в 2-килобайтных страницах
```

```
# соединяемся с сервером
$dbh = DBI->connect('dbi:Sybase:',$admin,$pw);
die "Невозможно соединиться: $DBI::errstr\n"
  unless (defined $dbh);

# получаем имена баз данных на сервере
$sth = $dbh->prepare(q{SELECT name from sysdatabases}) or
  die "Невозможно подготовить запрос к sysdatabases: ".$dbh->errstr."\n";
$sth->execute or
  die "Невозможно выполнить запрос к sysdatabases: ".$dbh->errstr."\n";

while ($aref = $sth->fetchrow_arrayref) {
    push(@dbs, $aref->[0]);
}
$sth->finish;

# получаем состояние для каждой из баз данных
foreach $db (@dbs) {

    # получаем и суммируем значения из поля size для всех
    # сегментов, не относящихся к журналам регистрации
    $size    = &querysum(qq{SELECT size FROM master.dbo.sysusages
                            WHERE  dbid = db_id(\'$db\')
                            AND    segmap != 4});
    # получаем и суммируем значения из поля size для сегмента,
    # соответствующего журналам регистрации
    $logsize = &querysum(qq{SELECT size FROM master.dbo.sysusages
                            WHERE  dbid = db_id(\'$db\')
                            AND    segmap = 4});

    # переходим к другой базе данных и получаем информацию об
    # используемом пространстве
    $dbh->do(q{use $db}) or
      die "Невозможно изменить базу данных на $db: ".$dbh->errstr."\n";

    # мы использовали функцию reserved_pgs, чтобы вернуть
    # количество страниц, используемых под данные (doampg) и
    # индекс (ioampg)
   $used=&querysum(q{SELECT reserved_pgs(id,doampg)+reserved_pgs(id,ioampg)
                     FROM sysindexes
                     WHERE id != 8});

    # то же самое, только на этот раз получаем информацию по
    # журналам регистрации
    $logused=&querysum(q{SELECT reserved_pgs(id, doampg)
                         FROM sysindexes
                         WHERE id=8});

    # выводим информацию в графическом виде
    &graph($db,$size,$logsize,$used,$logused);
}
$dbh->disconnect;

# готовим/выполняем запрос SELECT, получаем сумму результатов
```

```
sub querysum {
    my($query) = shift;
    my($sth,$aref,$sum);

    $sth = $dbh->prepare($query) or
      die "Невозможно подготовить запрос $query: ".$dbh->errstr."\n";
    $sth->execute or
      die "Невозможно выполнить запрос $query: ".$dbh->errstr."\n";

    while ($aref=$sth->fetchrow_arrayref) {
        $sum += $aref->[0];
    }
    $sth->finish;

    $sum;
}

# выводим в виде диаграммы имя базы данных, ее размер, размер журнала
# регистрации и информацию об использовании пространства
sub graph {
    my($dbname,$size,$logsize,$used,$logused) = @_;

    # строка для информации об использовании пространства данными
    print ' 'x15 . '|'.'d'x (50 *($used/$size)) .
          ' 'x (50-(50*($used/$size))) . '|';

    # использованное пространство и общий объем, отведенный под данные
    printf("%.2f",($used/$size*100));
    print "%/". (($size * $pages)/1024)."MB\n";
    print $dbname.'-'x(14-length($dbname)).'-|'.(' 'x 49)."|\n";

    if (defined $logsize) { # строка для информации об
         # использовании пространства под журналы регистрации
        print ' 'x15 . '|' . 'l'x (50 *($logused/$logsize)) .
              ' 'x (50-(50*($logused/$logsize))) . '|';
        # использованное пространство и общий объем, отведенный
        # под журналы регистрации
        printf("%.2f",($logused/$logsize*100));
        print "%/". (($logsize * $pages)/1024)."MB\n";
    }
    else { # у некоторых баз данных нет отдельного места для
           # журналов регистрации
         print ' 'x15 . "|- no log".(' 'x 41)."|\n";
    }
    print "\n";
}
```

Читатель, разбирающийся в SQL, вероятно, удивится, зачем использовать специальную подпрограмму (`querysum`) для суммирования данных из одного столбца вместо того, чтобы применить отличный оператор `SUM` из SQL. `querysum()` придумана только в качестве примера того, что можно сделать на лету из Perl. Подпрограмма на Perl подходит, скорее, для более сложных задач. Например, если нужно отдельно просуммировать данные, выбирая их по регулярному выражению, лучше

это сделать из Perl, чем обращаться к серверу и просить его составить таблицы (даже если это можно сделать).

Где выполняется вся работа?

Вопрос, который может возникнуть при написании SQL-программ из Perl, звучит так: «Нужно ли обрабатывать данные на сервере при помощи SQL или на клиенте при помощи Perl?» Часто SQL-функции на сервере (например, `SUM()`) и операторы Perl пересекаются.

Так, было бы эффективнее использовать ключевое слово `DISTINCT`, чтобы удалить повторяющиеся записи из возвращаемых данных, перед тем как передавать их программе на Perl, даже если эту операцию легко можно выполнить в самом Perl.

К сожалению, существует слишком много переменных, чтобы можно было быстро и точно решить, какой метод применять. Вот несколько факторов, которые следует учитывать:

- Насколько эффективно сервер обрабатывает определенный запрос?
- Сколько данных обрабатывается?
- Сколько нужно обрабатывать данные и насколько сложна эта обработка?
- Каковы скорость сервера, клиента и сети (если она используется)?
- Хотите ли вы, чтобы код можно было перенести на другой сервер баз данных?

Зачастую приходится испытать оба способа, прежде чем сделать выбор.

Наблюдение за использованием процессорного времени на SQL-сервере

В последнем примере этой главы DBI будет выводить обновляемую раз в минуту строку состояния, содержащую информацию об использовании процессорного времени на SQL-сервере. Чтобы сделать задачу более интересной, можно из одного и того же сценария одновременно наблюдать за двумя серверами. Комментарий к сценарию последует позже:

```
use DBI;

$syadmin = "sa";
print "Пароль администратора базы данных Sybase: ";
chomp($sypw = <STDIN>);
```

```
$msadmin = "sa";
print "Пароль администратора базы данных MS-SQL: ";
chomp($mspw = <STDIN>);

# соединяемся с сервером Sybase
$sydbh = DBI->connect("dbi:Sybase:server=SYBASE",$syadmin,$sypw);
die "Невозможно соединиться с сервером sybase: $DBI::errstr\n"
  unless (defined $sydbh);
# включаем параметр ChopBlanks, чтобы удалить концевые пробелы из столбцов
$sydbh->{ChopBlanks} = 1;

# соединяемся с сервером MS-SQL (очень здорово, что мы можем
# использовать для этого DBD::Sybase!)
$msdbh = DBI->connect("dbi:Sybase:server=MSSQL",$msadmin,$mspw);
die "Невозможно соединиться с сервером mssql: $DBI::errstr\n"
  unless (defined $msdbh);
# включаем параметр ChopBlanks, чтобы удалить концевые пробелы
$msdbh->{ChopBlanks} = 1;

$|=1; # выключаем буферизацию вывода STDOUT

# инициализируем обработчик сигнала с тем, чтобы можно было
# корректно завершиться
$SIG{INT} = sub {$byebye = 1;};

# бесконечный цикл, который завершится при установке
# нашего флага прерывания
while (1) {
    last if ($byebye);

    # запускаем хранимую процедуру sp_monitor
    $systh = $sydbh->prepare(q{sp_monitor}) or
       die "Невозможно подготовить sy sp_monitor:".$sydbh->errstr."\n";
    $systh->execute or
       die "Невозможно выполнить sy sp_monitor:".$sydbh->errstr."\n";
    # цикл для получения нужных нам строк.
    # мы знаем, что у нас есть все, что нужно, когда мы
    # получаем информацию cpu_busy
    while($href = $systh->fetchrow_hashref or
          $systh->{syb_more_results}) {
             # есть то, что нужно, перестаем спрашивать
             last if (defined $href->{cpu_busy});
    }
    $systh->finish;

    # заменяем все, кроме %, значениями, которые мы
    # получили
    for (keys %{$href}) {
       $href->{$_} =~ s/.*-(\d+%)/\1/;
    }

    # собираем все нужные нам данные в одну строку
    $info = "Sybase: (".$href->{cpu_busy}." CPU), ".
```

```
                    "(".$href->{io_busy}." IO), ".
                    "(".$href->{idle}." idle)   ";

    # отлично, теперь сделаем то же самое и для другого сервера
    # (MS-SQL)
    $mssth = $msdbh->prepare(q{sp_monitor}) or
       die "Невозможно подготовить ms sp_monitor:".$msdbh->errstr."\n";
    $mssth->execute or
       die "Невозможно выполнить ms sp_monitor:".$msdbh->errstr."\n";
    while($href = $mssth->fetchrow_hashref or
            $mssth->{syb_more_results}) {
              # есть то, что нужно, перестаем спрашивать
             last if (defined $href->{cpu_busy});
    }
    $mssth->finish;

    # заменяем все, кроме %
    for (keys %{$href}) {
       $href->{$_} =~ s/.*-(\d+%)/\1/;
    }

    $info .= "MSSQL: (" . $href->{'cpu_busy'}." CPU), ".
                    "(".$href->{'io_busy'}." IO), ".
                    "(".$href->{'idle'}." idle)";
    print " "x78,"\r";
    print $info,"\r";

    sleep(5) unless ($byebye);
}

# попадаем сюда, только если мы прервали цикл
$sydbh->disconnect;
$msdbh->disconnect;
```

Сценарий выводит эту строку на экран и обновляет ее каждые пять секунд:

```
Sybase: (33% CPU), (33% IO), (0% idle)   MSSQL: (0% CPU), (0% IO), (100% idle)
```

Основу данной программы составляет хранимая процедура `sp_monitor`, существующая как на Sybase-, так и на MS-SQL-сервере. Вывод `sp_monitor` выглядит примерно так:

```
last_run                    current_run                 seconds
--------------------------- --------------------------- -----------
Aug 3 1998 12:05AM          Aug 3 1998 12:05AM          1

cpu_busy                  io_busy                   idle
------------------------- ------------------------- -----------------------
0(0)-0%                   0(0)-0%                    40335(0)-0%
```

```
packets_received          packets_sent              packet_errors
------------------------- ------------------------- -------------------------
1648(0)                   1635(0)                   0(0)

total_read          total_write         total_errors        connections
------------------- ------------------- ------------------- ---------------
391(0)              180(0)              0(0)                11(0)
```

К сожалению, `sp_monitor` показывает непереносимую особенность Sybase, которая прекочевала к MS-SQL: множественные наборы результатов. Каждая из строк возвращается в виде отдельного результата. Модуль `DBD::Sybase` справляется с этим, устанавливая специальный атрибут команды. Вот как возникла эта проверка:

```
while($href = $systh->fetchrow_hashref or
      $systh->{syb_more_results}) {
```

и вот почему следовало выйти из цикла до того, как были замечены нужные поля:

```
        # есть то, что нужно, перестаем спрашивать
        last if (defined $href->{cpu_busy});
```

Сама программа будет выполняться в вечном цикле до тех пор, пока не получит сигнал прерывания (наиболее вероятно, что это будет нажатие клавиш <Ctrl>+<C> пользователем). Получив такой сигнал, мы делаем самую безопасную вещь из тех, что можно сделать с обработчиком сигнала, и устанавливаем флаг возврата. Подобную технологию рекомендуют использовать на страницах *perlipc* для безопасной обработки сигналов. После получения сигнала `INT` будет установлен соответствующий флаг, который выбросит нас из цикла на следующей итерации. Получение этого сигнала позволяет программе «деликатно» закрыть дескрипторы баз данных, перед тем как сбросить «этот бренный шум».[1]

Эта небольшая программа всего лишь затронула возможности наблюдения за состоянием сервера, доступные нам. Было бы несложно, взяв полученные от `sp_monitor` результаты, построить график, чтобы получить более наглядное представление о том, как используется сервер. Но... оставим заботы об украшательстве читателю.

---

[1] Текст оригинала «before shuffling off this mortal coil» – почти цитата из монолога Гамлета: «When we have shuffled off this mortal coil...» – в переводе М. Лозинского эта строка выглядит так: «Когда мы сбросим этот бренный шум...». – *Примеч. ред.*

Информация о модулях из этой главы

Модуль	Идентификатор на CPAN	Версия
`DBI`	TIMB	1.13
Msql-Mysql-модули (`DBD::mysql`)	JWIED	1.2210
`DBD::Sybase`	MEWP	0.21
`Win32::ODBC` (с *http://www.roth.net*)	GBARR	970208

Рекомендуемая дополнительная литература

SQL

http://w3.one.net/~jhoffman/sqltut.htm – содержит отличное руководство по SQL Джеймса Хоффмана (James Hoffman); в конце руководства можно найти ссылки на сайты, посвященные SQL.

DBI

«Advanced Perl Programming», Sriram Srinivasan (O'Reilly, 1997).

http://www.symbolstone.org/technology/perl/DBI/index.html – официальная домашняя страница DBI; это должно быть вашей первой остановкой.

«Programming the Perl DBI», Alligator Descartes, Tim Bunce (O'Reilly, 2000).

ODBC

http://www.microsoft.com/odbc – информация об ODBC от Microsoft. Можно также поискать информацию об ODBC и о библиотеках ODBC в MDAC SDK на сайте *http://msdn.microsoft.com*.

http://www.roth.net/perl/odbc/ – официальная страница `Win32::ODBC`.

«Win32 Perl Programming: The Standard Extensions», Dave Roth (Macmillan Technical Publishing, 1999). Книга автора `Win32::ODBC`, на настоящий момент это лучший справочник по программированию модулей для Win32 Perl.

Прочее

http://sybooks.sybase.com – вся документация от Sybase с удобным интерфейсом поиска и простой системой навигации. Иногда бывает полезна не только для решения вопросов по Sybase/MS-SQL, но и для общих вопросов по SQL.

http://www.mbay.net/~mpeppler/ – домашняя страница Майкла Пепплера (автора *SybPerl* и `DBD::Sybase`). Содержит информацию не только по Sybase, но и по программированию баз данных в целом.

8

- *Отправка почты*
- *Распространенные ошибки при отправке почты*
- *Получение почты*
- *Информация о модулях из этой главы*
- *Рекомендуемая дополнительная литература*

Электронная почта

В отличие от других глав книги, здесь не обсуждается администрирование какой-либо службы, технологии или предметной области. Вместо этого мы собираемся рассмотреть, как использовать электронную почту из Perl в качестве инструмента системного администрирования.

В контексте системного администрирования Perl может быть полезен, как посылая, так и отправляя почту. Электронная почта – это отличный способ сообщить о чем-либо: например о том, что в программе происходит что-то не то; или о результатах выполнения автоматического процесса (скажем, службы планировщика заданий или *cron*), или об изменениях чего-то, за чем необходимо следить. Мы выясним, как посылать такую почту из Perl, а также узнаем о некоторых ловушках, связанных с отправкой почты самим себе.

Также мы рассмотрим, применение Perl для обработки входящей почты, повышая ее эффективность. Perl поможет бороться со спамом и разбираться с вопросами пользователей.

Будем впредь считать, что у вас уже есть надежная, работоспособная почтовая система. Кроме того, будем исходить из того, что в ней применяются протоколы, соответствующие спецификации IETF для отправки и получения почты. В примерах этой главы используется протокол SMTP (простой протокол передачи почты, RFC821), а сообщения соответствуют RFC822. Но всему свое время.

Отправка почты

Начнем с рассмотрения механизмов отправки почты, а затем перейдем к более сложным вопросам. Традиционный (для Unix) Perl-код для

отправки почты бывает похож на пример, включенный в список часто задаваемых вопросов:

```
# считаем, что sendmail установлен
open(SENDMAIL, "|/usr/lib/sendmail -oi -t -odq") or
  die "Невозможно запустить процесс для sendmail: $!\n";
print SENDMAIL <<"EOF";
From: от кого <me\@host>
To: кому <you\@otherhost>
Subject: Тема сообщения

Тело сообщения следует после пустой строки и может состоять из любого
количества строк.
EOF
close(SENDMAIL) or warn "Невозможно закрыть sendmail ";
```

Когда в Perl 5 изменились правила интерполяции массивов (по сравнению с Perl 4), очень много сценариев, отправляющих почту, перестали работать. Даже сейчас будьте осторожны и следите за таким кодом:

```
$address = "fred@example.com";
```

Чтобы все работало верно, его надо заменить на одну из следующих строк:

```
$address="fred\@example.com";
$address='fred@example.com';
$address= join('@', 'fred', 'example.com');
```

Код, вызывающий *sendmail*, как было в нашем примере, во многих случаях будет работать отлично, но если в операционной системе не установлен агент передачи почты с именем «sendmail» (например, в NT или MacOS), он не будет работать никогда. В таких операционных системах выбор действий невелик.

Получение sendmail (или иного агента передачи почты)

Если вы работаете на Win32, то вам повезло, т. к. я знаю по крайней мере о трех версиях *sendmail*, перенесенных под Win32:

о трех версиях *sendmail*, перенесенных под Win32:

- Перенесенная версия *sendmail* от Cygwin (*http://dome/weeg.uiowa.edu/pub/domestic/sos/ports*)
- Коммерческая версия *sendmail* от Mercury Systems (*http://www.demobuilder.com/sendmail.htm*)
- Коммерческая версия *Sendmail for NT* от Sendmail, Inc. (*http://www.sendmail.com*)

Тем, кому нужно что-то менее тяжеловесное и кто хочет внести некоторые изменения в программу на Perl, чтобы поддержать различные аргументы командной строки, возможно, помогут достичь цели другие программы для Win32:

- *blat* (*http://www.interlog.com/~tcharron/blat.html*)
- *netmail95* (*http://www.geocities.com/SiliconValley/Lakes/2382/netmail.html*)
- *wmailto* (*http://www.impaqcomp.com/jgaa/wmailto.html*)

Преимущества такого подхода состоят в том, что можно выбросить из сценария все сложности отправки почты. Хороший агент передачи почты (MTA) пытается повторно соединиться с почтовым сервером, если тот в данный момент недоступен, выбирает нужный целевой сервер (ищет записи Mail eXchanger в DNS и осуществляет переходы между ними), при необходимости переписывает заголовки, справляется с внезапными коллизиями и т. д. Если можно избежать необходимости заботиться обо всем этом в Perl, то это просто замечательно.

Использование IPC, специфичных для операционной системы

В MacOS или Windows NT можно управлять почтовым клиентом, используя IPC (Interprocess Communication, межпроцессные взаимодействия).

Я не знаю о существовании версий *sendmail* для MacOS, но в нем для управления почтовым клиентом можно применять AppleScript:

```
$to="someone\@example.com";
$from="me\@example.com";
$subject="Hi there";
$body="message body\n";

MacPerl::DoAppleScript(<<EOC);
tell application "Eudora"

    make message at end of mailbox "out"

    -- 0 is the current message
    set field \"from\" of message 0 to \"$from\"
    set field \"to\" of message 0 to \"$to\"
    set field \"subject\" of message 0 to \"$subject\"
    set body of message 0 to \"$body\"
    queue message 0
    connect with sending without checking
    quit
end tell
EOC
```

В этом примере запускается очень простая программа AppleScript, которая общается с почтовым клиентом *Eudora*. Сценарий создает новое сообщение, помещает его в очередь для отправки, а затем отдает инструкции почтовому клиенту об отправке сообщения из очереди перед выходом.

Еще один, более эффективный способ написать подобный сценарий состоит в том, чтобы использовать модуль `Mac::Glue`, уже рассмотренный в главе 2 «Файловые системы».

```
use Mac::Glue ':glue';

$e=new Mac::Glue 'Eudora';
$to="someone\@example.com";
$from="me\@example.com";
$subject="Hi there";
$body="message body";

$e->make(
   new => 'message',
   at => location(end => $e->obj(mailbox => 'Out'))
);

$e->set($e->obj(field => from    => message => 0), to => $from);
$e->set($e->obj(field => to      => message => 0), to => $to);
$e->set($e->obj(field => subject => message => 0), to => $subject);
$e->set($e->prop(body => message => 0), to => $body);

$e->queue($e->obj(message => 0));
$e->connect(sending => 1, checking => 0);
$e->quit;
```

В NT можно обратиться к библиотеке Collaborative Data Objects Library (раньше она называлась Active Messaging), простой в использовании надстройке на архитектуре MAPI (интерфейс прикладного программирования систем передачи сообщений). Вызвать эту библиотеку для управления таким почтовым клиентом, как Outlook можно, применив модуль `Win32::OLE` следующим образом:

```
$to="me\@example.com";
$subject="Hi there";
$body="message body\n";

use Win32::OLE;

# инициализируем OLE и COINIT_OLEINITIALIZE, необходимые при
# использовании объектов MAPI.Session
Win32::OLE->Initialize(Win32::OLE::COINIT_OLEINITIALIZE);
die Win32::OLE->LastError(),"\n" if Win32::OLE->LastError();

# создаем объект сессии, который вызовет Logoff при уничтожении
```

```
my $session = Win32::OLE->new('MAPI.Session','Logoff');
die Win32::OLE->LastError(),"\n" if Win32::OLE->LastError();

# регистрируемся в этой сессии, используя OL98 Internet Profile по умолчанию
$session->Logon('Microsoft Outlook Internet Settings');
die Win32::OLE->LastError(),"\n" if Win32::OLE->LastError();

# создаем объект message
my $message = $session->Outbox->Messages->Add;
die Win32::OLE->LastError(),"\n" if Win32::OLE->LastError();

# создаем объект recipient
my $recipient = $message->Recipients->Add;
die Win32::OLE->LastError(),"\n" if Win32::OLE->LastError();

# заполняем данными объект recipient
$recipient->{Name} = $to;
$recipient->{Type} = 1; # 1 = "To:", 2 = "Cc:", 3 = "Bcc:"

# все адреса должны быть расшифрованы по справочнику
# (в этом случае, скорее всего, по вашей адресной книге)
# Полные адреса расшифровываются сами в себя, так что эта
# строка в большинстве случаев не изменит объект recipient
$recipient->Resolve();
die Win32::OLE->LastError(),"\n" if Win32::OLE->LastError();

# заполняем строку Subject: и тело сообщения
$message->{Subject} = $subject;
$message->{Text} = $body;

# помещаем сообщение в очередь для отправки
# 1-й аргумент = сохранить копию сообщения
# 2-й аргумент = позволить пользователю изменить сообщение
# перед отправкой в диалоговом окне
# 3-й аргумент = родительское окно диалога, если 2-й аргумент True
$message->Send(0, 0, 0);
die Win32::OLE->LastError(),"\n" if Win32::OLE->LastError();

# явно уничтожить объект $session, вызвав $session->Logoff
undef $session;
```

В отличие от предыдущего примера, программа всего лишь помещает письмо в очередь. Это уже дело почтового клиента (такого как Outlook) или транспортного агента (например Exchange) периодически инициировать отправку почты. Существует CDO/AM 1.1 – метод для объекта `Session` **под названием** `DeliverNow()`**, обращающийся к MAPI с заданием очистить все очереди входящих и исходящих сообщений. К сожалению, в некоторых ситуациях он недоступен или не работает, поэтому его нет и в предыдущем примере.**

В упомянутом примере управление MAPI производится «вручную» при помощи вызовов OLE. Если вы хотите использовать MAPI, «не пачкая рук», можно применить модуль `Win32::MAPI`, который берет на себя все функции (модуль находится на *http://www.generation.net/~aminer/Perl/*).

Программы, полагающиеся на AppleScript/Apple Events или MAPI, так же непереносимы, как и вызов программы *sendmail*. Они берут на себя часть работы, но относительно неэффективны. К этим методам нужно прибегать в последнюю очередь.

Общение напрямую по почтовым протоколам

Последний выбор – написать программу, общающуюся с почтовым сервером на его родном языке. Большая часть этого языка документирована в RFC821. Вот как выглядит основной обмен данными по SMTP (Simple Mail Transfer Protocol, простой протокол передачи почты). Данные, которые мы посылаем, выделены жирным шрифтом:

```
% telnet example.com 25              -- соединяемся с SMTP-портом на
                                        example.com
Trying 192.168.1.10 ...
Connected to example.com.
Escape character is '^]'.
220 mailhub.example.com ESMTP Sendmail 8.9.1a/8.9.1; Sun, 11 Apr 1999
15:32:16 -0400 (EDT)
HELO client.example.com              -- идентифицируем машину, с которой мы
                                        пришли (можно использовать EHLO)

250 mailhub.example.com Hello dnb@client.example.com [192.168.1.11], pleased
to meet you
MAIL FROM: <dnb@example.com>         -- определяем отправителя
250 <dnb@example.com>... Sender ok
RCPT TO: <dnb@example.com>           -- определяем получателя
250 <dnb@example.com>... Recipient ok
DATA                                 -- начинаем отправлять данные, не забы-
                                        вая о некоторых ключевых заголовках
354 Enter mail, end with "." on a line by itself
From: David N. Blank-Edelman (David N. Blank-Edelman)
To: dnb@example.com
Subject: SMTP - хороший протокол

Просто хочу напомнить себе о том, насколько я люблю SMTP.
     С миром,
       dNb
.                                    -- завершаем сообщение
250 PAA26624 Message accepted for delivery
QUIT                                 -- конец сессии
221 mailhub.example.com closing connection
Connection closed by foreign host.
```

Несложно записать в сценарий подобную беседу. Можно было бы использовать модуль Socket или что-нибудь вроде Net::Telnet, как в главе 6 «Службы каталогов». Но существует несколько хороших модулей для отправки почты, которые упрощают эту задачу. Среди них модуль Женды Крыницки (Jenda Krynicky) Mail::Sender, Mail::Sendmail Миливожа Ивковича (Milivoj Ivkovic) и Mail::Mailer из пакета *MailTools* Грэхема Бара (Graham Barr). Все эти модули не зависят от операционной системы и будут работать практически везде, где доступен современный дистрибутив Perl. Мы рассмотрим Mail::Mailer, поскольку он предлагает единый интерфейс к двум способам отправки почты, которые обсуждались до сих пор. Как и в случае с большинством модулей, написанных в объектно-ориентированном стиле, первый шаг заключается в создании экземпляра нового объекта:

```
use Mail::Mailer;

$from="me\@example.com";
$to="you\@example.com";
$subject="Hi there";
$body="message body\n";

$type="smtp";
$server="mail.example.com";

my $mailer = Mail::Mailer->new($type, Server => $server) or
  die "Невозможно создать новый объект mailer:$!\n";
```

Переменная $type позволяет выбрать один из следующих типов поведения:

smtp

Посылает почту, обращаясь к модулю Net::SMTP (часть пакета *libnet*), доступному и для большинства не-Unix версий Perl. Если используется *MailTools* версии 1.13 или выше, можно задать имя SMTP-сервера, применяя приведенную выше символику =>. В противном случае, придется устанавливать имя сервера во время процедуры установки *libnet*.

mail

Отправка почты при помощи почтового агента *mail* (или любого другого, который задан вторым необязательным аргументом). Это напоминает недавнее использование AppleScript и MAPI.

sendmail

Отправка почты с помощью программы *sendmail*, как и в первом случае из данного раздела.

Кроме того, можно установить переменную окружения PERL_MAILERS, чтобы изменить путь, установленный по умолчанию для поиска программ (например, *sendmail*) в системе.

Вызов метода `open()` для нашего объекта `Mail::Mailer` заставляет последний выполнять роль дескриптора для исходящего сообщения. В этом вызове передаются заголовки сообщения ссылке на анонимный хэш:

```
$mailer->open({From => $from,
               To => $to,
               Subject => $subject}) or
  die "Невозможно заполнить объект mailer:$!\n";
```

Тело сообщения выводится в этот псевдодескриптор, который потом закрывается для отправки сообщения:

```
print $mailer $body;
$mailer->close;
```

Этого вполне достаточно, чтобы отправка почты из Perl не зависела от системы.

В зависимости от того, какой тип поведения `$type` был выбран при работе с модулем, могут оказаться скрытыми (а могут и не оказаться) более сложные вопросы, относящиеся к MTA, о которых уже говорилось. В предыдущем примере использовалось поведение `smtp`, а это означает, что программа должна быть достаточно умна, чтобы обрабатывать такие сбои как недоступность сервера. Приведенный пример не настолько «сообразителен». Обязательно позаботьтесь о таких моментах, когда будете писать программы.

Распространенные ошибки при отправке почты

Что ж, можно приступить к использованию электронной почты для отправки извещений. Однако когда мы начнем писать программы для выполнения этой функции, то быстро обнаружим, что вопрос *как* посылать почту, отнюдь не так интересен, как вопросы *когда* и *что* посылать.

В данном разделе мы исследуем эти вопросы, действуя «от противного». Если рассмотреть, что и как *не надо* посылать, то сами вопросы можно изучить глубже. Так что поговорим об ошибках, допускаемых наиболее часто при написании программ системного администрирования, отправляющих почту.

Слишком частая отправка сообщений

Самая распространенная ошибка – это отправка слишком большого количества сообщений. Иметь сценарии, отправляющие почту, вообще говоря, отличная идея. Если с какой-либо службой случится что-то неприятное, то отправка обычного электронного сообщения или сообщения на пейджер – очень хороший способ привлечь внимание чело-

века к случившейся проблеме. Но в большинстве случаев, раз в пять минут отправлять по сообщению о неприятностях – это очень *плохое* решение. Слишком усердные почтовые генераторы очень быстро попадают в почтовые фильтры тех, кто должен был их читать. В результате оказывается, что важная почта просто игнорируется.

Контроль над частотой отправки почты

Самый простой способ избежать лишней почты – добавить в программу меры предосторожности, чтобы устанавливать задержку между сообщениями. Если сценарий запущен постоянно, то очень просто запомнить время отправки последнего сообщения:

```
$last_sent = time;
```

Если программа запускается один раз в N минут или часов через *cron* в Unix или механизмы планирования задач NT, эту информацию можно переписать в файл, состоящий из одной строки, и считывать его при следующем запуске программы. В подобном случае обязательно обратите внимание на меры предосторожности, перечисленные в главе 1 «Введение».

В зависимости от ситуации можно поэкспериментировать с временем задержки. В этом примере показана экспоненциальная задержка (exponential backoff):

```
$max  = 24*60*60; # максимальная задержка в секундах (1 день)
$unit = 60;       # увеличиваем задержку относительно этого значения (1
минута)

# интервал времени, прошедший с момента отправки предыдущего
# сообщения и последняя степень 2, которая использовалась для
# расчета интервала задержки. Созданная нами подпрограмма
# возвращает ссылку на анонимный массив с этой информацией
sub time_closure {
    my($stored_sent,$stored_power)=(0,-1);
    return sub {
       (($stored_sent,$stored_power) = @_) if @_;
       [$stored_sent,$stored_power];
    }
};

$last_data=&time_closure; # создаем замыкание

# возвращаем значение "истина" при первом вызове и затем после
# задержки
sub expbackoff {
    my($last_sent,$last_power) = @{&$last_data};

    # возвращаем true, если это первое наше обращение или если
    # текущая задержка истекла с тех пор, как мы спрашивали
```

```
    # последний раз. Если мы возвращаем значение true, мы
    # запоминаем время последнего утвердительного ответа и
    # увеличиваем степень двойки, чтобы вычислить задержку.
    if (!$last_sent or
       ($last_sent +
         (($unit * 2**$last_power >= $max) ?
             $max : $unit * 2**$last_power) <= time())){
            &$last_data(time(),++$last_power);
           return 1;
    }
    else {
      return 0;
    }
}
```

Подпрограмма `expbackoff()` возвращает значение `true` (1), если нужно отправить сообщение, и `false` (0), если нет. При первом вызове она возвращает `true`, а затем быстро увеличивает время задержки до тех пор, пока значение `true` не станет появляться лишь раз в день.

Чтобы сделать программу более интересной, я применил необычную конструкцию под названием *замыкание* (*closure*) для хранения времени последней отправки сообщения и последней степени двойки, используемой для расчета задержки. Замыкание используется как способ скрытия важных переменных от остальной программы. В данной маленькой программе это было сделано из любопытства, но польза от такой технологии очень быстро становится очевидной в программах большего размера, где более вероятно, что другой код может случайно перезаписать значения этих переменных. Вот, вкратце, как работают замыкания.

Подпрограмма `&time_closure()` возвращает ссылку на анонимную подпрограмму, по существу, на небольшой отрывок кода без имени. Позже данная ссылка будет вызывать этот код, используя стандартный синтаксис символических ссылок: `&$last_data`. Код из анонимной подпрограммы возвращает ссылку на массив, поэтому и используется такая масса знаков пунктуации, чтобы получить доступ к возвращаемым данным:

```
my($last_sent,$last_power) = @{&$last_data};
```

Вот и вся тайна, которая скрывается за замыканиями: поскольку ссылка создается в том же блоке, что и переменные `$stored_sent` и `$stored_power` (посредством `my()`), то они схватываются в уникальном контексте. Переменные `$stored_sent` и `$stored_power` можно прочитать и изменить только при выполнении кода из этой ссылки. Кроме того, они сохраняют свои значения между вызовами. Например:

```
# создаем замыкание
$last_data=&time_closure;
```

```
# вызываем подпрограмму, устанавливающую значения переменных
&$last_data(1,1);

# пытаемся изменить их за пределами подпрограммы
$stored_sent = $stored_power = 2;

# выводим их текущие значения, используя подпрограмму
print "@{&$last_data}\n";
```

Результатом выполнения этого кода будет "1 1", хотя и создается впечатление, что в третьей строке были изменены значения переменных `$stored_sent` и `$stored_power`. Да, значения глобальных переменных с теми же именами были изменены, но невозможно затронуть копии, защищенные замыканиями.

Можно говорить о переменной из замыкания как о спутнике на орбите планеты. Спутник удерживается гравитацией планеты, так что куда движется планета, туда перемещается и спутник. Позицию спутника можно описать только относительно планеты: чтобы найти спутник, сначала нужно отыскать планету. Каждый раз, когда вы находите планету, там же будет и спутник, на том же месте, где был и прошлый раз. Можно считать, что переменные из замыкания находятся на орбите вокруг ссылки на анонимную подпрограмму, отдельно от вселенной остальной программы.

Но оставим астрофизику в покое и вернемся к рассуждениям об отправке почты. Иногда лучше, чтобы программа вела себя как двухлетний ребенок, жалующийся с течением времени все чаще. Вот еще одна программа, похожая на предыдущий пример. На этот раз со временем увеличивается количество дополнительно посылаемых сообщений. Начинаем мы с отправки сообщений один раз в день, а затем уменьшаем время задержки до тех пор, пока минимальная задержка не станет равной пяти минутам:

```
$max  = 60*60*24; # максимальная задержка в секундах (1 день)
$min  = 60*5;     # минимальная задержка в секундах (5 минут)
$unit = 60;       # уменьшаем задержку относительно этого значения (1 минута)

$start_power = int log($max/$unit)/log(2); # ищем ближайшую степень двойки

sub time_closure {
    my($last_sent,$last_power)=(0,$start_power+1);
    return sub {
      (($last_sent,$last_power) = @_) if @_;
      # keep exponent positive
      $last_power = ($last_power > 0) ? $last_power : 0;
      [$last_sent,$last_power];
    }
};

$last_data=&time_closure; # создаем замыкание
```

```
# возвращаем true при первом вызове и затем после роста
# экспоненты
sub exprampup {
    my($last_sent,$last_power) = @{&$last_data};

    # возвращаем true, если это первое обращение или если
    # текущая задержка истекла с момента последнего обращения.
    # Если сообщение отправляется, то мы запоминаем время
    # последнего ответа и увеличиваем степень 2, используемую
    # для расчета задержки
    if (!$last_sent or
       ($last_sent +
         (($unit * 2**$last_power <= $min) ?
              $min : $unit * 2**$last_power) <= time())){
            &$last_data(time(),++$last_power[1]);
            return 1;
    }
    else {
       return 0;
    }
}
```

В обоих примерах вызывалась дополнительная подпрограмма (`&$last_data`), которая позволяла выяснить, когда было отправлено последнее сообщение и как вычислялась задержка. Позже, при необходимости изменить программу, такое деление позволит изменить способ хранения состояния. Например, если переписать программу так, чтобы она выполнялась периодически, а не постоянно, то замыкание совсем нетрудно заменить обычной подпрограммой, сохраняющей нужные данные в текстовом файле и потом считывающей их оттуда.

Контролируем количество сообщений

Другая разновидность синдрома «чрезмерной отправки почты» – это проблема «каждый в сети за себя». Если все машины из сети решат послать вам чуточку почты, вы вполне можете пропустить что-то действительно важное в этом потоке сообщений. Было бы лучше, если бы все сообщения отправлялись в центральный репозиторий. А затем в собранном виде почта поступала бы в одном сообщении.

Давайте рассмотрим несколько надуманный пример. Предположим, что каждая машина в сети записывает в разделяемый каталог файл, состоящий из одной строки.[2] Имя каждого файла совпадает с именем машины и в каждом из них хранятся результаты научных вычисле-

[1] В тексте оригинала «--$last_power». Изменено в соответствии с логикой изложения. – *Примеч. науч. ред.*

[2] Другое подходящее место для подобной информации – база данных.

ний, сделанных прошлой ночью. В файле будет одна строка следующего формата:

```
имя-узла удача-или-неудача количество-завершенных-вычислений
```

Программа, проверяющая эту информацию и отсылающая результаты, может выглядеть так:

```
use Mail::Mailer;
use Text::Wrap;

# список машин, отправляющих сообщения
$repolist = "/project/machinelist";
# каталог, куда они записывают файлы
$repodir  = "/project/reportddir";
# разделитель файловой системы, используется для переносимости.
# Можно было бы использовать модуль File::Spec
$separator= "/";
# отправляем почту "с" этого адреса
$reportfromaddr  = "project\@example.com";
# отправляем почту на этот адрес
$reporttoaddr    = "project\@example.com";
# считываем список машин в хэш. Позже будем вынимать из этого
# хэша по мере доклада машин, оставив только те машины, которые
# не принимали участие в действии
open(LIST,$repolist) or die "Невозможно открыть список $repolist:$!\n";
while(<LIST>){
    chomp;
    $missing{$_}=1;
    $machines++;
}

# считываем все файлы из центрального каталога
# замечание: этот каталог должен автоматически очищаться другим
# сценарием
opendir(REPO,$repodir) or die "Невозможно открыть каталог $repodir:$!\n";

while(defined($statfile=readdir(REPO))){
    next unless -f $repodir.$separator.$statfile;

    # открываем каждый файл и считываем информацию о состоянии
    open(STAT,$repodir.$separator.$statfile)
      or die "Невозможно открыть $statfile:$!\n";

    chomp($report = <STAT>);

    ($hostname,$result,$details)=split(' ',$report,3);

    warn "В файле $statfile утверждается, что он был сгенерирован машиной
    $hostname!\n"
      if($hostname ne $statfile);
```

```
    # имя узла больше не считается пропущенным
    delete $missing{$hostname};
    # заполняем значениями хэши
    if ($result eq "success"){
        $success{$hostname}=$details;
        $succeeded++;
    }
    else {
        $fail{$hostname}=$details;
        $failed++;
    }
    close(STAT);
}
closedir(REPO);

# создаем информативный заголовок для сообщения
if ($succeeded == $machines){
    $subject = "[report] Success: $machines";
}
elsif ($failed == $machines or scalar keys %missing >= $machines) {
    $subject = "[report] Fail: $machines";
}
else {
    $subject = "[report] Partial: $succeeded ACK, $failed NACK".
      ((%missing) ? ", ".(scalar keys %missing)[1]." MIA" : "");
}

# создаем объект mailer и заполняем заголовки
$type="sendmail";
my $mailer = Mail::Mailer->new($type) or
  die "Невозможно создать новый объект:$!\n";

$mailer->open({From=>$reportfromaddr, To=>$reporttoaddr, Subject=>$subject})
or
  die "Невозможно заполнить объект mailer:$!\n";

# создаем тело сообщения
print $mailer "Run report from $0 on " . scalar localtime(time) . "\n";

if (keys %success){
    print $mailer "\n==Succeeded==\n";
    foreach $hostname (sort keys %success){
      print $mailer "$hostname: $success{$hostname}\n";
    }
}
```

[1] **Наличие скобок обязательно из-за приоритета операций, выявлено при компиляции. –** ***Примеч. науч. ред.***

```
if (keys %fail){
    print $mailer "\n==Failed==\n";
    foreach $hostname (sort keys %fail){
      print $mailer "$hostname: $fail{$hostname}\n";
    }
}

if (keys %missing){
    print $mailer "\n==Missing==\n";
    print $mailer wrap("","",join(" ",sort keys %missing)),"\n";
}

# отправляем сообщение
$mailer->close;
```

Сначала программа считывает список имен машин, участвующих в данном предприятии. Затем, чтобы проверить, встречались ли машины, не поместившие файл в общий каталог, используется хэш, созданный на основе этого списка. Мы открываем каждый файл из данного каталога и выделяем из него информацию о состоянии. Получив результаты, создаем сообщение и отправляем его.

Получается такой отчет:

```
Date: Wed, 14 Apr 1999 13:06:09 -0400 (EDT)
Message-Id: <199904141706.NAA08780@example.com>
Subject: [report] Partial: 3 ACK, 4 NACK, 1 MIA
To: project@example.com
From: project@example.com

Run report from reportscript on Wed Apr 14 13:06:08 1999

==Succeeded==
barney: computed 23123 oogatrons
betty: computed 6745634 oogatrons
fred: computed 56344 oogatrons

==Failed==
bambam: computed 0 oogatrons
dino: computed 0 oogatrons
pebbles: computed 0 oogatrons
wilma: computed 0 oogatrons

==Missing==
mrslate
```

Другой способ изучить подобные результаты состоит в том, чтобы создать демон журналов регистрации и посылать отчет от каждой машины через сокет. Сначала взгляните на код для сервера. Он совпадает с кодом из предыдущего примера. Рассмотрим новую программу и обсудим ее важные особенности:

```
use IO::Socket;
use Text::Wrap; # используется для создания аккуратного вывода

# список машин, посылающих отчеты
$repolist = "/project/machinelist";
# номер порта для соединения с клиентами
$serverport = "9967";

&loadmachines; # загружаем список машин

# настраиваем нашу сторону сокета
$reserver = IO::Socket::INET->new(LocalPort => $serverport,
                                  Proto     => "tcp",
                                  Type      => SOCK_STREAM,
                                  Listen    => 5,
                                  Reuse     => 1)
  or die "Невозможно настроить сокет на нашей стороне: $!\n";

# начинаем слушать порт в ожидании соединений
while(($connectsock,$connectaddr) = $reserver->accept()){

    # имя подсоединившегося клиента
    $connectname = gethostbyaddr((sockaddr_in($connectaddr))[1],AF_INET);

    chomp($report=$connectsock->getline);

    ($hostname,$result,$details)=split(' ',$report,3);

    # если нужно сбросить информацию, выводим готовое к
    # отправке сообщение и заново инициализируем все
    # хэши/счетчики
    if ($hostname eq "DUMPNOW"){
      &printmail($connectsock);
      close($connectsock);
      undef %success;
      undef %fail;
      $succeeded = $failed = 0;
      &loadmachines;
      next;
    }

    warn "$connectname говорит, что был сгенерирован $hostname!\n"
      if($hostname ne $connectname);
    delete $missing{$hostname};
    if ($result eq "success"){
      $success{$hostname}=$details;
      $succeeded++;
    }
    else {
      $fail{$hostname}=$details;
      $failed++;
```

```
    }
    close($connectsock);
}
close($reserver);

# загружаем список машин из заданного файла
sub loadmachines {
    undef %missing;
    undef $machines;
    open(LIST,$repolist) or die "Невозможно открыть список $repolist:$!\n";
    while(<LIST>){
      chomp;
      $missing{$_}=1;
      $machines++;
    }
}

# выводим готовое к отправке сообщение. Первая строка - тема,
# последующие строки - тело сообщения
sub printmail{
    ($socket) = $_[0];

    if ($succeded == $machines){
      $subject = "[report] Success: $machines";
    }
    elsif ($failed == $machines or scalar keys %missing >= $machines) {
      $subject = "[report] Fail: $machines";
    }
    else {
      $subject = "[report] Partial: $succeeded ACK, $failed NACK".
        ((%missing) ? ", ".(scalar keys %missing)[1]." MIA" : "");
    }

    print $socket "$subject\n";

    print $socket "Run report from $0 on ".scalar localtime(time)."\n";

    if (keys %success){
      print $socket "\n==Succeeded==\n";
      foreach $hostname (sort keys %success){
       print $socket "$hostname: $success{$hostname}\n";
      }
    }

    if (keys %fail){
      print $socket "\n==Failed==\n";
      foreach $hostname (sort keys %fail){
```

[1] **Наличие скобок обязательно из-за приоритета операций, выявлено при компиляции. –** ***Примеч. науч. ред.***

```
        print $socket "$hostname: $fail{$hostname}\n";
      }
    }

    if (keys %missing){
      print $socket "\n==Missing==\n";
      print $socket wrap("","",join(" ",sort keys %missing)),"\n";
    }
}
```

Кроме переноса части кода в отдельные подпрограммы, главное изменение заключается в том, что добавлен код для работы с сетью. Модуль `IO::Socket` позволяет без труда открывать и использовать сокеты, которые можно сравнить с телефоном. Сначала нужно установить свою сторону сокета (`IO::Socket->new()`), как бы включая свой телефон, а затем ждать «звонка» от клиента (`IO::Socket->accept()`). Программа приостановлена (или «заблокирована») до тех пор, пока не установлено соединение. Когда соединение установлено, запоминается имя подсоединившегося клиента. Затем из сокета считывается строка ввода.

Мы предполагаем, что строка ввода выглядит точно так же, как и строки из отдельных файлов в предыдущем примере. Единственное различие – это загадочное имя узла *DUMPNOW*. Если это имя встречается, подсоединившемуся клиенту выводится тема и тело готового к отправке сообщения, при этом сбрасываются все счетчики и хэш-таблицы. За отправку сообщения, полученного от сервера, ответственен клиент. Теперь посмотрим на пример клиента и узнаем, что он может сделать с этим сообщением:

```
use IO::Socket;

# номер порта для соединения с клиентом
$serverport = "9967";
# и имя сервера
$servername = "reportserver";
# преобразуем имя в IP-адрес
$serveraddr = inet_ntoa(scalar gethostbyname($servername));
$reporttoaddr  = "project\@example.com";
$reportfromaddr  = "project\@example.com";

$reserver = IO::Socket::INET->new(PeerAddr => $serveraddr,
                                  PeerPort => $serverport,
                                  Proto    => "tcp",
                                  Type     => SOCK_STREAM)
  or die "Невозможно создать сокет на нашей стороне:: $!\n";

if ($ARGV[0] ne "-m"){
    print $reserver $ARGV[0];
}
```

```
else {
    use Mail::Mailer;

    print $reserver "DUMPNOW\n";
    chomp($subject = <$reserver>);
    $body = join("",<$reserver>);

    $type="sendmail";
    my $mailer = Mail::Mailer->new($type) or
      die "Невозможно создать новый объект mailer:$!\n";

    $mailer->open({
       From => $reportfromaddr,
         To => $reporttoaddr,
         Subject => $subject
        }) or
          die "Невозможно заполнить объект mailer:$!\n";

    print $mailer $body;
    $mailer->close;
}

close($reserver);
```

Эта программа проще. Сначала открывается сокет с сервером. В большинстве случаев ему передается информация о состоянии (полученная в командной строке как `$ARGV[0]`) и соединение закрывается. При желании создать клиент-серверную систему регистрации, подобную этой, вероятно, нам пришлось бы перенести данный клиентский код в подпрограмму и вызывать ее из другой, гораздо более крупной.

Если передать сценарию ключ $-m$, он отправит серверу «DUMPNOW» и прочитает полученную от него строку темы сообщения и тело сообщения. Затем этот вывод передается модулю `Mail::Mailer` и отправляется в виде почтового сообщения при помощи той же программы, которую мы видели раньше.

Для ограничения размера примера и для того, чтобы не уходить в сторону от дискуссии, здесь представлен лишь костяк кода для клиента и сервера. В нем нет ни проверки ошибок или ввода, ни управления доступом, ни авторизации (в сети любой, получивший доступ к серверу, может взять с него данные), ни постоянного хранилища данных (а что, если машина не работает?), ни даже мало-мальских мер предосторожности. Мало того, в каждый момент времени можно обрабатывать только один запрос. Если клиент остановится в середине транзакции, мы «влипли». Более изощренные примеры можно найти в книгах «Advanced Perl Programming» (Углубленное программирование на Perl) Шрирама Шринивасана (Sriram Srinivasan) и «Perl Cookbook» («Perl: Библиотека программиста») Тома Кристиансена (Tom Christiansen) и Натана Торкингтона (Nathan Torkington), обе выпущены изда-

тельством O'Reilly. Модуль `Net::Daemon` Джошена Вьедмана (Jochen Wiedmann) также поможет создавать более сложные программы-демоны.

Однако пора вернуться к рассмотрению других ошибок, допускаемых в программах для системного администрирования, отправляющих почтовые сообщения.

Пропуск темы сообщения

Строка *Subject:* – это такая «вещица», которую не стоит пропускать. При автоматической отправке почты можно на лету создавать информативную строку *Subject:* для каждого сообщения. Так что практически нет прощения тому, кто отправляет в почтовый ящик сообщения с заголовками:

```
Super-User     File history database merge report
Super-User     File history database merge report
Super-User     File history database merge report
Super-User     File history database merge report
Super-User     File history database merge report
Super-User     File history database merge report
Super-User     File history database merge report
```

в то время как они могли бы выглядеть так:

```
Super-User     Backup OK, 1 tape, 1.400 GB written.
Super-User     Backup OK, 1 tape, 1.768 GB written.
Super-User     Backup OK, 1 tape, 2.294 GB written.
Super-User     Backup OK, 1 tape, 2.817 GB written.
Super-User     Backup OK, 1 tape, 3.438 GB written.
Super-User     Backup OK, 3 tapes, 75.40 GB written.
```

Строка *Subject*: должна содержать краткую и четкую информацию, описывающую происходящее. Из темы сообщения должно быть очевидно, докладывает ли программа об успехе, неудаче или о чем-то среднем. Незначительные добавочные усилия с лихвой окупаются экономией времени при чтении почты.

Недостаточная информация в теле сообщения

Эта ошибка попадает в ту же категорию, что и предыдущая. Если сценарий собирается сообщать о проблемах или ошибках в почтовых сообщениях, значит сгенерировать какую-то информацию, которая должна там присутствовать. Они сводятся к каноническим вопросам журналистики:

Кто?

Какой сценарий сообщает об ошибке? Добавьте содержимое переменной `$0` (если не устанавливали ее явно), чтобы показать полный

путь к текущему сценарию. Сообщите о версии сценария, если таковая у него имеется.

Где?

Сообщите что-то о том месте в сценарии, где возникает проблема. Функция `caller()` из Perl возвращает всю нужную для этого информацию:

```
# замечание: то, что возвращает caller(), может зависеть от
# версии Perl, так что обязательно загляните в документацию по
# perlfunc
($package, $filename, $line, $subroutine, $hasargs, $wantarray,
 $evaltext, $is_require) = caller($frames);
```

Где `$frames` – это количество нужных фреймов на стеке (если вызывались подпрограммы из подпрограмм). Чаще всего вы будете устанавливать `$frames` в `1`. Вот пример списка, возвращаемого функцией `caller()` в середине кода для сервера из последнего полного примера:

```
('main','repserver',32,'main::printmail',1,undef)
```

Подобная запись указывает, что сценарий, запущенный из файла *repserver* в строке 32, находился в пакете `main`. В этот момент выполнялся код из подпрограммы `main::printmail` (у нее есть аргументы, кроме того, она не вызывается в списочном контексте).

Если вы не хотите вручную применять `caller()`, можете воспользоваться отчетом о проблемах, предоставляемым модулем `Carp`.

Когда?

Опишите состояние программы в момент возникновения ошибки. К примеру, какой была последняя строка прочтенных данных?

Почему?

Если сумете, ответьте на незаданный читателем вопрос: «Зачем беспокоить меня этим почтовым сообщением?» Ответ может быть очень простым, например: «данные об учетных записях не были полностью обработаны», «DNS-сервер сейчас недоступен» или «в серверной пожар». Это даст читающему представление о предмете разговора (и, возможно, побудит к изучению).

Что?

Ну и наконец, надо сказать о том, что же пошло не так раньше всего.

Вот небольшой пример на Perl, который охватывает все эти пункты:

```
use Text::Wrap;

sub problemreport {
# $shortcontext - описание проблемы в одной строке
# $usercontext - подробное описание проблемы
```

```
# $nextstep - лучшее предположение о том, что делать, чтобы исправить
проблему
    my($shortcontext,$usercontext,$nextstep) = @_;
    my($filename, $line, $subroutine) = (caller(1))[1,2,3];

    push(@return,"Проблема с $filename: $shortcontext\n");

    push(@return,"*** Сообщение о проблеме с $filename ***\n\n");
    push(@return,fill("","","- Проблема: $usercontext")."\n\n");
    push(@return,"- Место: строка $line файла $filename в
                $subroutine\n\n");
    push(@return,"- Произошла: ".scalar localtime(time)."\n\n");

    push(@return,"- Дальнейшие действия: $nextstep\n");

    \@return;
}

sub fireperson {
    $report = &problemreport("компьютер горит ",<<EOR,<<EON);
При составлении отчета загорелась задняя часть компьютера. Случилось это
сразу же после обработки пенсионного плана для ORA.
EOR
Пожалуйста, потушите пожар, а потом продолжайте работу.
EON

  print @{$report};

}

&fireperson;
```

Обращение к `&problemreport` выведет, начиная с темы сообщения, отчет о проблеме, согласующийся с `Mail::Mailer`, как и в предыдущих примерах. Ну а `&fireperson` является примером тестирования этой подпрограммы.

Теперь, разобравшись с отправкой почты, перейдем к другой стороне медали.

Получение почты

Обсуждая в этом разделе получение почты, мы не будем говорить о ее *сборе* (*fetching*). Передача почты с одной машины на другую не представляет особого интереса. Модули `Mail::POP3Client` Сина Дауда (Sean Dowd) и `Mail::Cclient` Малколма Битти (Malcolm Beattie) легко могут передать почту по протоколам POP (Post Office Protocol) или IMAP (Internet Message Access Protocol). Гораздо интереснее посмотреть, что с этой почтой делать после ее получения, и именно на этом мы и остановимся.

Начать следует с основ, поэтому рассмотрим инструменты, позволяющие разбить как отдельные сообщения, так и почтовые ящики. Чтобы рассмотреть первое, вновь обратимся к пакету *MailTools* Грэма Бара (Graham Barr), на этот раз прибегнем к модулям `Mail::Internet` и `Mail::Header`.

Разбиение отдельных сообщений

Модули `Mail::Internet` и `Mail::Header` предлагают удобный способ разбить заголовки почтового сообщения, соответствующего RFC822. RFC822 определяет формат почтового сообщения, включая имена допустимых заголовков и их форматов.

Модулю `Mail::Internet` необходимо передать либо файловый дескриптор файла с сообщением, либо ссылку на массив, содержащий его строки:

```
use Mail::Internet;

$messagefile = "mail";

open(MESSAGE,"$messagefile") or die "Невозможно открыть $messagefile:$!\n";
$message = new Mail::Internet \*MESSAGE;
close(MESSAGE);
Если мы хотим анализировать сообщение, поступающее в стандартный поток ввода
(т. е. переданное через конвейер на стандартный ввод), можно поступить так:
use Mail::Internet;

$message = new Mail::Internet \*STDIN;
```

`Mail::Internet` возвращает экземпляр объекта сообщения. Чаще всего с этим экземпляром объекта будет применяться один из двух методов: `body()` и `head()`. Метод `body()` возвращает ссылку на анонимный массив, содержащий строки тела сообщения. `head()` более интересен и предлагает плавное продолжение модуля `Mail::Header`.

При загрузке `Mail::Internet` неявно загружается `Mail::Header`. Если вызвать метод `head()` модуля `Mail::Internet`, он вернет экземпляр объекта заголовка `Mail::Header`. Это будет тот же экземпляр объекта, который можно получить, если использовать не `Mail::Internet`, а напрямую `Mail::Header`:

```
use Mail::Header;

$messagefile = "mail";

open(MESSAGE,"$messagefile") or die "Невозможно открыть $messagefile:$!\n";
$header = new Mail::Header \*MESSAGE;
close(MESSAGE);
```

Объект `$header` содержит заголовки сообщения и предлагает несколько удобных способов для получения данных. Например, чтобы вывести

отсортированный список встречающихся имен заголовков (которые в модуле называются «тегами»), можно добавить такую строчку в конец предыдущего примера:

```
print join("\n",sort $header->tags);
```

В зависимости от сообщения можно увидеть что-то подобное нижеследующему:

```
Cc
Date
From
Message-Id
Organization
Received
Reply-To
Sender
Subject
To
```

Необходимо получить все заголовки *Received:* из сообщения. Вот как это можно сделать:

```
@received = $header->get("Received");
```

Часто методы `Mail::Header` используются вместе с объектом `Mail::Internet`. Если применять `Mail::Internet` для возвращения объекта, содержащего и тело и заголовки сообщения, можно сцепить вместе некоторые методы из этих модулей:

```
@received = $message->head->get("Received");
```

Обратите внимание, что `get()`вызывается в списочном контексте. В скалярном контексте метод вернул бы только первое вхождение этого тега, в случае, если бы мы не задали вторым аргументом порядковый номер тега. Например, `get("Received",2)` вернет вторую строку `Received:` из сообщения. Модуль `Mail::Header` предоставляет и другие методы для удаления и добавления тегов в заголовки; подробную информацию можно найти в документации.

Разбиение почтового ящика

Перейти на следующий уровень, где мы разобьем на части почтовые ящики, довольно просто. Если почта хранится в формате «классического mbox» или *qmail* (еще один агент передачи почты, подобный *sendmail*), можно использовать модуль `Mail::Folder` Кевина Джонсона (Kevin Johnson). Многие из распространенных почтовых агентов (не в Unix), таких как Eudora, тоже хранят почту в классическом формате Unix mbox, так что этот модуль может быть полезным на многих платформах.

Что-то подобное мы уже видели:

```
use Mail::Folder::Mbox; # для классического формата Unix mbox

$folder = new Mail::Folder('mbox',"filename");
```

Конструктор `new()` принимает тип формата почтового ящика и имя файла для анализа. Он возвращает экземпляр объекта `folder`, через который можно запрашивать, добавлять, удалять и изменять сообщения. Чтобы получить шестое сообщение, нужно применить следующее:

```
$message = $folder->get_message(6);
```

Теперь `$message` содержит экземпляр объекта `Mail::Internet`. С этим экземпляром объекта можно применять все только что обсужденные методы. Если вам нужен только заголовок сообщения, можно использовать:

```
$header = $folder->get_header(6);
```

Здесь нет никаких сюрпризов; возвращается ссылка на экземпляр объекта `Mail::Header`. Чтобы узнать о других доступных методах, загляните в документацию по `Mail::Folder`.

Отслеживание спама

Теперь, когда мы знаем, как можно разбить на части сообщение, посмотрим, что делать с этим умением на практике. Первое, что можно рассмотреть, – это непрошеная коммерческая электронная почта, или спам. Большинству пользователей не нравится получать такие сообщения. Системные администраторы не любят спам за то, что он попусту забивает каталоги почтовых очередей и журналы регистрации. Кроме того, каждое непрошеное коммерческое письмо вызывает недовольство пользователей, так что в почтовом ящике системного администратора на каждое письмо «из спама» зачастую приходится более десятка возмущенных писем от пользователей.

Лучший способ сдерживать подобный напор – создать такую «атмосферу», в которой спам недопустим и, помимо этого, его очень сложно посылать. Жалобы интернет-провайдеру спамера (у большинства из которых определены строгие правила на этот счет) часто приводят к тому, что виновнику отказывают в услугах. Если это происходит вновь и вновь, то со временем спамеру все сложнее найти провайдера. А чем сложнее спамеру продолжать свое дело, тем меньше вероятность, что он в нем останется.

Жалобы провайдерам затруднены по следующим причинам:

- Спамеры часто подделывают часть своих сообщений, пытаясь скрыть следы. Сам по себе этот факт является еще одной причиной не любить спамеров. В конце концов, это красноречиво говорит об их намерениях.

- Спамеры часто посылают свою «макулатуру» через посторонние (и совершенно невинные, но неверно настроенные почтовые серверы). Этот процесс называется «ретрансляцией» (relaying), поскольку в большинстве случаев такая почта ни исходит от кого-либо с такого сервера, ни предназначена кому-либо там. Почтовый сервер выступает лишь в качестве передатчика. Системные администраторы серверов, допускающих ретрансляцию, часто оказываются в сложном положении. Их атакованные почтовые серверы начинают страдать от непредвиденных загрузок (вероятно, в ущерб полезным службам), имена их серверов попадают в черные списки и довольно большая часть почты «проникнута» ненавистью тех, кто подвергся «атаке» спамера.

С помощью Perl можно проанализировать сообщение и найти его источник. Начнем мы с малого и затем перейдем к более сложным вопросам, применяя то, что узнали в главе 5 «Службы имен TCP/IP» и главе 6. Если читателю хочется посмотреть на очень сложный Perl-сценарий для борьбы со спамом, советую взглянуть на *adcomplain* Билла Макфаддена (Bill McFadden), который можно найти на *http://www.rdrop.com/users/billmc/adcomplain.html*.

Вот копия настоящего спамерского сообщения, тело которого изменено, чтобы не доставить его автору удовольствия:

```
Received: from isiteinc.com (www.isiteinc.com [206.136.243.2])
   by mailhost.example.com (8.8.6/8.8.6) with ESMTP id NAA14955
   for <webadmin@example.com>; Fri, 7 Aug 1998 13:55:41 -0400 (EDT)
From: responses@example.com
Received: from extreme (host-209-214-9-150.mia.bellsouth.net
  [209.214.9.150])
  by isiteinc.com (8.8.3/8.8.3) with SMTP id KAA19050 for
    webadmin@example.com; Fri, 7 Aug 1998 10:48:09 -0700 (EDT)
Date: Fri, 7 Aug 1998 10:48:09 -0700 (EDT)
Received: from login_0246.whynot.net mx.whynot.net[206.212.231.88])
    by whynot.net (8.8.5/8.7.3) with SMTP id XAA06927 for
    <webadmin@example.com>; Fri, 7 August 1998 13:48:11 -0700 (EDT)
To: <webadmin@example.com>
Subject: ***ADVERTISE VACATION RENTALS - $25/year*** - http://www.example.com
Reply-To: sample@whynot.net
X-PMFLAGS: 10322341.10
X-UIDL: 10293287_192832.222
Comments: Authenticated Sender is <user122@whynot.net>
Message-Id: <77126959_36550609>

Мы рады представить вам новый веб-сайт http://www.example.com от Extreme
Technologies, Inc.

Наш новый сайт, посвященный путешествиям, содержит список некоторых из самых
интересных путешествий, которые только можно отыскать в WWW. На нашем сайте
вы легко найдете информацию об аренде и продаже жилья, о предоставляемых
```

```
услугах и многом другом. Наш список сопровождается цветными фотографиями,
анимированной графикой, точными описаниями и информацией о том, как напрямую
связаться с арендатором/поставщиком. Кроме того, мы изменяем графику на сайте
каждый месяц!
```

Давайте критически оценим это сообщение. Во-первых, большинство его заголовков подозрительны. Как вы видели в предыдущем разделе этой главы, большинство заголовков (*To:*, *From:* и т. д.) передаются почтовой программе в разделе DATA передачи сообщения. Это единственный набор заголовков, которые трудно подделать и которые добавляются агентом передачи почты, когда сообщение проходит через почтовую систему.

В особенности нужно внимательно посмотреть на заголовки *Received:*. Спамер может подделать строки *Received:*, но он не сумеет убрать те из них, которые были добавлены последующими почтовыми системами. Можно вставить фальшивые, но это потребует определенных и довольно серьезных знаний (например, как подделать TCP/IP-пакет или сфабриковать записи в DNS), которыми рядовые спамеры очень редко обладают.

Начнем с того, что выделим заголовки *Received:* из сообщения и представим их на экране в более читаемом виде. Мы выведем их в той последовательности, в которой передавалось сообщение, начиная с первого почтового сервера, получившего сообщение, и заканчивая местом назначения (нашим сайтом):

```
use Mail::Header;

$header = new Mail::Header \*STDIN;

$header->unfold('Received');
@received = $header->get('Received');

for (reverse @received){
    chomp;
    parseline($_);
    if (!defined $ehelo and !defined $validname and !defined $validip){
      print "$_\n";
    }
    else {
      write;
    }
}

format STDOUT =
@<<<<<<<<<<<<<<<<<<<<<<  @<<<<<<<<<<<<<<<<<<<<<<  @<<<<<<<<<<<<<<<<
$ehelo,$validname,$validip
.

sub parseline {
```

```
    my $line = $_;

    # "нормальный" -- от HELO (REAL [IP])
    if (/from\s+(\w\S+)\s*\((\S+)\s*\[(\d+\.\d+\.\d+\.\d+)/){
      ($ehelo,$validname,$validip) = ($1,$2, $3);
    }
    # невозможно выполнить обратное разыменование -- от HELO ([IP])
    elsif (/from\s+(\w\S+)\s+\(\[(\d+\.\d+\.\d+\.\d+)\]/){
      ($ehelo,$validname,$validip) = ($1,undef, $2);
    }
    # exim -- из [IP] (helo=[HELO IP])
    elsif (/
from\s+\[(\d+\.\d+\.\d+\.\d+)\]\s+\(helo=\[(\d+\.\d+\.\d+\.\d+)\]/){
      ($validip,$ehelo,$validname) = ($1,$2, undef);
    }
    # Sun Internet Mail Server -- из [IP] by HELO
    elsif (/from\s+\[(\d+\.\d+\.\d+\.\d+)\]\s+by\s+(\S+)/){
      ($validip,$ehelo,$validname) = ($1,$2, undef);
    }
    # Microsoft SMTPSVC -- из HELO - (IP)
    elsif (/from\s+(\S+)\s+-\s+(\d+\.\d+\.\d+\.\d+)\s+/){
      ($ehelo,$validname,$validip) = ($1,$2, $3);
    }
    else { # punt!
      $ehelo = $validname = $validip = undef;
    }

    return [$ehelo,$validname,$validip];
}
```

Первым делом из сообщения выбираются (при помощи `unfold()`) заголовки *Received:*. Метод `unfold()` удаляет из заданных строк символы новой строки и символы продолжения. Это делается для упрощения анализа.

Все строки просматриваются в обратном порядке (по сравнению с тем, как они были найдены в сообщении). По существу, движение идет от центра сообщения к периферии, т. к. каждая система, через которую оно проходило, добавляла еще один уровень заголовков *Received:*. Большая часть работы выполняется подпрограммой `&parseline`. Используя несколько регулярных выражений, попробуем выделить следующее из заголовков *Received:*:

Имя узла HELO/EHLO

Имя, представленное в обмен на HELO или EHLO при SMTP-«беседе».

«Действительный» IP-адрес

IP-адрес клиента, замеченный агентом передачи почты во время соединения. Он, скорее всего, будет «действительным», поскольку

использует информацию, не зависящую от того, что предоставил клиент во время SMTP-диалога. Это важно, поскольку клиент спамера, скорее всего, будет заядлым лжецом. Слово *действительный* берется в кавычки потому, что существуют способы сфабриковать и эту информацию.

«Действительное» имя

Имя клиента, найденное агентом передачи почты при обратном разыменовании IP-адреса через DNS. Эта информация, как и предыдущая, поступает не от клиента (хотя она тоже может быть фальшивой).

Формат правильной строки *Received:* определяется в RFC821 и RFC822. Однако, если просмотреть доставленную почту (как сделал я, создавая использованные регулярные выражения), можно заметить, что не все агенты передачи почты следуют данному правилу. В нашей программе будут применяться наиболее распространенные форматы, но существуют и другие способы, которые обязательно надо учитывать, если есть намерение в дальнейшем расширять код. Чтобы иметь представление о возможных форматах, посмотрите в сценарий *adcomplain*.

Вот как выглядит вывод программы, выполненной для сообщения, приведенного ранее:

```
login_0246.whynot.net    mx.whynot.net           206.212.231.88
extreme                  host-209-214-9-150.mia  209.214.9.150
isiteinc.com             www.isiteinc.com        206.136.243.2
```

В первом столбце перечислены имена, используемые машинами при идентификации; во втором – имена этих машин, указанные серверу при соединении; а в третьем – IP-адреса инициаторов данного соединения. Как уже отмечалось, в построенном списке последняя строка соответствует машине, передавшей сообщение на наш почтовый сервер.

Спамеры не могут удалить строки *Received:*, но они могут изменить их содержимое, задав фальшивое имя во время приветствия HELO или EHLO. Подтверждение этому можно увидеть во второй строке примера, поскольку имя узла из первого столбца не имеет ничего общего с именем из второго столбца, который более достоверен.

Но предположим, что они совпадают. Как же тогда узнать, что строка *Received:* была подделана? Один из способов – сверять «действительный» IP-адрес из каждой строки *Received:* с «действительным» именем узла и сообщать об отклонениях. Приведенная ниже подпрограмма вернет значение (1), если разыменованное имя не совпадет с IP-адресом, в противном случае – (0). Вскоре этот код будет добавлен к основной программе:

```
use Socket;

sub checkrev{
```

```
    my($ip,$name) = @_;

    return 0 unless ($ip and $name);

    my $namelook = gethostbyaddr(inet_aton($ip),AF_INET);
    my $iplook   = gethostbyname($name);

    $iplook = inet_ntoa($iplook) if $iplook;

    # может быть записан в различных регистрах
    if ($iplook eq $ip and lc $namelook eq lc $name){
        return 0;
    }
    else {
        return 1;
    }
}
```

Такая проверка не совсем надежна, потому что, как ни обидно, у вполне законных узлов может отсутствовать информация для обратного разыменования. Кроме того, серверам имен может быть предписано отвечать неверно, предоставляя фиктивную информацию (т. е. на `gethostbyaddr()` нельзя безоговорочно полагаться).

Много о чем можно догадаться из заголовков *Received:*, прежде чем отследить владельцев каждого хопа (hop). Например, кем разделяется мнение, что любой из хопов является известным источником спама?

Поиск в локальном черном списке

На некоторых сайтах ведутся локальные черные списки узлов, известных как распространители спама. Такая тактика была взята на вооружение, когда спам только начал появляться, и некоторые провайдеры отказывались принимать меры даже против самых закоренелых клиентов-спамеров. В ответ на это в основных агентах передачи почты появились механизмы, отвергающие соединения от узлов и доменов, входящих в список антисоциальных.

Можно использовать такой черный список, чтобы разобраться, проходило ли сообщение через узлы, известные своим спамерством. Ясно, что в черном списке нет сервера, передавшего нам почту (иначе с ним просто не было бы установлено соединение), но любой из остальных почтовых серверов, указанный в заголовках *Received:*, вполне может там быть.

Не существует способа написать одну программу, проверяющую все возможные черные списки агентов передачи почты, поскольку разные агенты хранят эту информацию в различных форматах. Большая часть узлов в Интернете в настоящее время применяет в качестве агента передачи почты *sendmail*, так что в нашем примере будет применяться его формат черного списка. В новых версиях *sendmail* черный спи-

сок хранится в базе данных при помощи библиотек Berkeley DB 2.X, доступных на *http://www.sleepycat.com*.

Поль Маркес (Paul Marquess) написал модуль `BerkeleyDB`, специально предназначенный для работы с библиотеками Berkeley 2.x/3.x. Это может сбить с толку, поскольку в документации по `DB_File`, еще одному известному модулю Маркеса, входящему в состав дистрибутива Perl, также рекомендуется применять библиотеки 2.x/3.x. `DB_File` использует библиотеки Berkeley DB 2.x/3.x в «режиме совместимости» (в частности, библиотека собирается с ключом `--enable-compat185`, так что доступен API версии 1.x API). Модуль `BerkeleyDB` позволяет программисту на Perl применять расширенные возможности из API версии 2.x/3.x.

Агент передачи *sendmail* использует формат Berkeley DB 2.x/3.x, так что нужно включить модуль `BerkeleyDB`. Вот пример, который выводит содержимое локального черного списка:

```
$blacklist = "/etc/mail/blacklist.db";

use BerkeleyDB;

#свяжем хэш %blist с черным списком, используя Berkeley DB
# для получения значений
tie %blist, 'BerkeleyDB::Hash', -Filename  => $blacklist
  or die "Невозможно открыть файл $filename: $! $BerkeleyDB::Error\n" ;

# обходим в цикле каждый ключ и значение из этого файла,
# выводя только записи REJECT
while(($key,$value) = each %blist){
    # в списке также могут быть записи "OK", "RELAY" и др.
    next if ($value ne "REJECT");

    print "$key\n";
}
```

Принимая за основу этот код, можно написать подпрограмму, проверяющую, находится ли данный узел или домен (содержащий этот узел) в черном списке. Если нужно узнать об узле *mailserver.spammer.com*, следует обойти в цикле все записи из черного списка (в котором могут находиться *mailserver.spammer.com*, *spammer.com* или даже просто *spammer*), чтобы проверить, содержатся ли в имени узла какие-либо записи из него.

Существует много способов написать на Perl программу, сравнивающую список значений с какими-либо данными. Но для того чтобы программа была эффективной и интересной, мы будем использовать две продвинутые технологии. Они созданы для уменьшения числа компиляций регулярных выражений, которые применяются в ходе выполнения программы. Каждый раз, когда программа использует «новое» регулярное выражение, Perl должен компилировать его зано-

во. Например, в этом отрывке кода Perl вынужден обрабатывать новое значение на каждой итерации:

```
# вообразите себе внешний цикл, в котором этот код вызывается
# множество раз
foreach $match (qw(alewife davis porter harvard central kendall park)){
   $station =~ /$match/ and print "found our station stop!";
}
```

Этот процесс требует больших вычислительных затрат, так что если бы можно было сократить количество вычислений, программа стала бы намного эффективней. Время, потраченное на компиляцию регулярных выражений, становится значительным в программах, где в цикле рассматривается список различных регулярных выражений.

Вот пример первой технологии, созданной для решения указанной проблемы:

```
use BerkeleyDB;

$blacklist = "/etc/mail/blacklist.db";

&loadblist;

# принимаем имя узла в качестве аргумента командной строки и
# сообщаем, если оно есть в черном списке
if (defined &checkblist($ARGV[0])){
        print "*** $found найден в черном списке \n";
    }

# загружаем черный список в массив анонимных подпрограмм
sub loadblist{
    tie %blist, 'BerkeleyDB::Hash', -Filename  => $blacklist
      or die "Невозможно открыть $filename: $! $BerkeleyDB::Error\n" ;

    while(my($key,$value) = each %blist){
        # в черном списке могут быть "OK", "RELAY" и пр.
        next if ($value ne "REJECT");
        push(@blisttests, eval 'sub {$_[0] =~ /\Q$key/o and $key}');
    }
}

sub checkblist{
    my($line) = shift;

    foreach $subref (@blisttests){
        return $found if ($found = &$subref($line));
    }
    return undef;
}
```

В этом примере используются анонимные подпрограммы – технология, продемонстрированная в книге Джозефа Хола (Joseph Hall) «Effective Perl Programming» (Эффективное программирование на Perl) (Addison Wesley). Для каждой записи из черного списка создается анонимная подпрограмма. Каждая подпрограмма сверяет переданные ей данные с одним из элементов черного списка. Если они совпадают, такая запись возвращается. Ссылки на эти подпрограммы хранятся в списке. Вот строка, в которой создается подпрограмма и ссылка на нее добавляется к списку:

```
push(@blisttests, eval 'sub {$_[0] =~ /\Q$key/o and $key}');
```

Так что, если в черном списке есть запись *spammer*, ссылка на код, добавленная в массив, будет указывать на подпрограмму, по сути эквивалентную следующей:

```
sub {
   $_[0] =~ /\Qspammer/o and "spammer";
}
```

\Q в начале регулярного выражения присутствует для того, чтобы точки (как в *.com*) или другие зарезервированные знаки пунктуации не считались бы метасимволами регулярных выражений.

Позже в программе будет обойден в цикле список ссылок на код и выполнена каждая маленькая подпрограмма для переданных данных. Если результатом каких-либо из этих вычислений окажется значение «истина», мы вернем код возврата подпрограммы:

```
return $found if ($found = &$subref($line));
```

Компиляция регулярного выражения, которая нас так беспокоит, происходит всего один раз – при создании ссылки на подпрограмму. Можно вызывать каждую подпрограмму столько раз, сколько надо, не теряя время на компиляцию регулярного выражения.

Существует и другой, чуть менее продвинутый подход, который можно применять, если у вас Perl версии 5.005 или выше. В Perl 5.005 была введена новая синтаксическая конструкция, названная «прекомпилируемым регулярным выражением», которая делает подобную задачу несколько проще. Переписать код, используя эту новую конструкцию, можно было бы примерно так:

```
sub loadblist{
    tie %blist, 'BerkeleyDB::Hash', -Filename  => $blacklist
      or die "Невозможно открыть файл $filename: $! $BerkeleyDB::Error\n" ;

    while(my($key,$value) = each %blist){
        # в черном списке могут быть записи "OK", "RELAY" и пр.
        next if ($value ne "REJECT");
        push(@blisttests,[qr/\Q$key/,$key]);
```

```
    }
}

sub checkblist{
    my($line) = shift;

    foreach my $test (@blisttests){
        my($re,$key) = @{$test};
        return $key if ($line =~ /$re/);
    }
    return undef;
}
```

На этот раз ссылка была перенесена на анонимный массив в `@blisttest`. Первый элемент этого массива – скомпилированное регулярное выражение, созданное с применением нового синтаксиса `qr//`. Это позволяет сохранить регулярное выражение после его компиляции. Такая форма обработки значительно увеличивает скорость выполнения программы при дальнейшем поиске соответствия. Второй элемент анонимного массива – сама запись из черного списка, которая будет возвращена при найденном соответствии скомпилированному регулярному выражению.

Поиск в черном списке для всего Интернета

В последнем примере программы на вопрос «Спамер ли это?» мы отвечали, руководствуясь собственным мнением об узле или домене, не принимая во внимание опыт остальных пользователей Интернета. Существуют несколько спорные службы[1], предлагающие простой доступ к глобальным черным спискам спамеров или известных узлов, открыто допускающих ретрансляцию почты. Две хорошо известные службы такого типа – Realtime Blackhole List (RBL) от Mail Abuse Prevention System и Open Relay Behaviour-modification System (ORBS). Для получения доступа к этим спискам:

1. Измените на обратный порядок следования элементов проверяемого IP-адреса. Например, `192.168.1.34` станет `34.1.168.192`.
2. Добавьте специальное имя домена к полученному числу. Для проверки адреса в RBL необходимо использовать `34.1.168.192.rbl.maps.vix.com`.
3. Выполните запрос к DNS-серверу для данного адреса.

[1] Споры ведутся о том, нужно ли вести подобные черные списки, кто должен их поддерживать, как их нужно применять, при каких обстоятельствах узлы должны добавляться туда и удаляться оттуда и о многих других «политических» моментах, которые только можно себе представить. Информацию о таких службах можно найти на сайтах *http://www.maps.vix.com* и *http://www.orbs.org*.

Если вы получите положительный ответ (т. е. запись о ресурсах A), это означает, что данный IP-адрес находится в черном списке.

Несколько менее спорным является список Dial-up User List, также поддерживаемый силами специалистов из Mail Abuse Prevention System. Это список диапазонов IP-адресов, динамически присваиваемых модемным пулам. Теоретически, SMTP-соединения не должны исходить от какого-либо из этих узлов. Почта с таких узлов должна отправляться через почтовый сервер провайдера (которого нет в этом списке).

Вот один из способов проверить, находится ли IP-адрес в каком-либо из этих списков:

```
sub checkaddr{
    my($ip,$domain) = @_;

    return undef unless (defined $ip);

    my $lookupip = join('.',reverse split(/\./,$ip));

    if (gethostbyname($lookupip.$domain)){
        return $ip;
    }
    else {
        return undef;
    }
}
```

Очень скоро эта подпрограмма будет добавлена в предпоследний пример этого раздела. А пока, располагая существенно бо́льшим объемом информации о каждом из заголовков *Received:*, попробуем вычислить человека или людей, ответственных за администрирование каждой машины из списка. Модуль Net::Whois, уже рассмотренный в главе 6, вероятно, первым будет использоваться для решения этой проблемы.

К сожалению, этот модуль специализируется только на получении информации о связи имен и доменов (name-to-domain information). Кроме того, он «предполагает», что информация будет представлена в виде, используемом InterNIC. Нам могут понадобиться сведения о связи IP-адресов и доменов (IP address-to-domain information) от WHOIS-серверов на *http://whois.arin.net* (American Registry for Internet Numbers), *http://whois.ripe.net* (European IP Address Allocations) и *http://whois.apnic.net* (Asia Pacific Address Allocations). Отсутствие соответствующего модуля – первое препятствие, которое необходимо преодолеть.

Но даже если бы мы знали, как соединиться со всеми этими реестрами и обработать их различные форматы вывода, было бы неясно, к какому из них нужно обращаться для поиска информации о данном IP-адресе. Нам необходимо определить, к какому серверу нужно обратиться, и это второе препятствие. К счастью, если обратиться к ARIN с за-

просом по адресу, не принадлежащему его базе данных, он направит нас на нужный реестр. Так что, если мы спросим ARIN об адресе из Японии, он отправит нас на APNIC.

Для преодоления первого препятствия можно использовать модуль общего назначения, подобный `Net::Telnet` из главы 6. Другой путь – уже рассмотренный модуль `IO::Socket`. Что выбрать – дело личных предпочтений, ну и, конечно, необходима возможность доступа к нему с вашей платформы.

Служба WHOIS работает на порту 43 TCP, хотя ее имя будет использоваться только в целях предосторожности. С WHOIS-сервером очень просто общаться. Необходимо соединиться, выполнить запрос (в нашем случае это IP-адрес) и получить ответ. Программа, запрашивающая произвольный WHOIS-сервер, очень проста:

```
sub getwhois{
    my($ip) = shift;
    my($info);

    $cn = new Net::Telnet(Host => $whoishost,
                          Port => 'whois',
                          Errmode => "return",
                          Timeout => 30)
          or die "Невозможно установить соединение с $whoishost
connection:$!\n";

    unless ($cn->print($ip."\n")){
        $cn->close;
        die "Невозможно послать $ip на $whoishost: ".$cn->errmsg."\n";
    }

    while ($ret = $cn->get){
        $info .=$ret;
    };

    $cn->close;

    return $info;
}
```

Для преодоления второго препятствия, состоящего в выборе нужного реестра, есть, по крайней мере, две возможности. Можно послать запрос к *http://whois.arin.net* и проанализировать ответ. Например, вот запись диалога с ARIN по поводу IP-адреса японской машины. Жирный шрифт используется для выделения текста, введенного человеком:

```
% telnet whois.arin.net 43
Trying 192.149.252.22 ...
Connected to whois.arin.net.
Escape character is '^]'.
```

```
210.161.92.226
Asia Pacific Network Information Center (NETBLK-APNIC-CIDR-BLK)
   Level 1 - 33 Park Road
   Milton, 4064
   AU

   Netname: APNIC-CIDR-BLK2
   Netblock: 210.0.0.0 - 211.255.255.0

   Coordinator:
      Administrator, System  (SA90-ARIN)  sysadm@APNIC.NET
      +61-7-3367-0490

   Domain System inverse mapping provided by:

   SVC01.APNIC.NET              202.12.28.131
   NS.TELSTRA.NET               203.50.0.137
   NS.KRNIC.NET                 202.30.64.21
   NS.RIPE.NET                  193.0.0.193

   *** please refer to whois.apnic.net for more information ***
   *** before contacting APNIC                              ***
   *** use whois -h whois.apnic.net <object>                ***

   Record last updated on 04-Mar-99.
   Database last updated on 19-Apr-99 16:14:16 EDT.
```

Подобные результаты означают, что запрос нужно послать к *http://whois.apnic.net*.

Другой способ – послать запрос к «умному» WHOIS-серверу, который сделает всю работу сам. Из них мне больше всего нравится сервер – *http://whois.geektools.com*.[1] «Умник» проанализирует ваш запрос, отправит его на нужный WHOIS-сервер и вернет результаты. Тому, кто пользуется этой службой, не нужно беспокоиться о том, на каком именно сервере хранится информация.

Чтобы код программы сильно не разрастался, а мы не отвлекались от обсуждаемой темы, будем пользоваться вторым (более простым) способом.

Поместим все эти маленькие запросы в одну большую программу и запустим ее. Итак, если нужно выполнить все приведенные выше подпрограммы с нашим примером сообщения:

```
use Mail::Header;
use Socket;
```

[1] Кстати, WHOIS прокси-сервер GeekTools написан на Perl. Детальную информацию об этой службе и копию кода можно найти на *http://www.geektools.com*.

```
use BerkeleyDB;
use Net::Telnet;

$header = new Mail::Header \*STDIN;

$header ->unfold('Received');
@received = $header->get('Received');

$rbldomain  = ".rbl.maps.vix.com";
$orbsdomain = ".relays.orbs.org";
$duldomain  = ".dul.maps.vix.com";
$blacklist  = "/etc/mail/blacklist.db";
$whoishost  = "whois.geektools.com";

&loadblist;

for (reverse @received){
    chomp;


    parseline($_);
    if (!defined $ehelo and !defined $validname and !defined $validip){
       print "$_\n";
    }
    else {
     $flags  = (&checkaddr($validip,$rbldomain)  ? "R" : ""); # в RBL?
     $flags .= (&checkaddr($validip,$orbsdomain) ? "O" : ""); # в ORBS?
     $flags .= (&checkaddr($validip,$duldomain)  ? "D" : ""); # в DUL?
     $flags .= (&checkblist($_)                  ? "B" : ""); # в нашем списке?
     $flags .= (&checkrev($validip,$validname)   ? "L" : ""); # rev-lookup?
      push(@iplist,$validip);

      write;
    }
}

for (@iplist){
    print "\nWHOIS info for $_:\n";
    print &getwhois($_);
}

format STDOUT =
@<<<<<<<<<<<<<<<<<<<<  @<<<<<<<<<<<<<<<<<<<<  @<<<<<<<<<<<<<<< @<<<<
$ehelo,$validname,$validip,$flags
.
```

то будут получены такие результаты (слегка сокращенные):

```
login_0246.whynot.net   mx.whynot.net           206.212.231.88   L
extreme                 host-209-214-9-150.mia  209.214.9.150    DB
isiteinc.com            www.isiteinc.com        206.136.243.2    OB
```

```
WHOIS info for 206.212.231.88:

WHOIS info for 209.214.9.150:
BellSouth.net Inc. (NETBLK-BELLSNET-BLK4)
   1100 Ashwood Parkway
   Atlanta, GA 30338

   Netname: BELLSNET-BLK4
   Netblock: 209.214.0.0 - 209.215.255.255
   Maintainer: BELL

   Coordinator:...

WHOIS info for 206.136.243.2:
Brainsell Incorporated (NET-ISITEINC)
   4105-R Laguna St.
   Coral Gables, FL 33146
   US

   Netname: ISITEINC
   Netnumber: 206.136.243.0

   Coordinator:...
```

Гораздо лучше! Теперь нам известно:

- Спамер дал неверные ответы HELO/EHLO.
- Первый узел, по всей вероятности, фальшивый (не удалась попытка разыменования, и по нему нет информации WHOIS).
- Сообщение, скорее всего, попало в сеть через соединение по телефонным линиям.
- Два из этих адресов уже находятся в нашем черном списке.
- ORBS они тоже не нравятся.
- Кроме того, нам известна контактная информация для связи с провайдером.

Perl помог разобраться с непрошеной коммерческой почтой.

Впрочем, спам это очень неприятная вещь. Лучше перейдем к более приятной теме, например, к взаимодействию пользователей средствами электронной почты.

Увеличение почты в службу поддержки

Даже если у вас нет официальной службы поддержки, наверняка существует несколько адресов, куда пользователи могут посылать свои вопросы и сообщения о трудностях. У электронной почты, как средства для связи по вопросам поддержки, есть несколько преимуществ:

- Все можно хранить и отслеживать, в отличие от разговоров в коридорах.
- Электронная почта асинхронна; системный администратор может читать почту и отвечать на нее в более спокойные вечерние часы.
- По желанию, это может быть индивидуальная, групповая или широковещательная рассылка. Если 14 человек пишут об одном и том же (скажем, об одних и тех же сообщениях спамеров), есть возможность ответить им всем одновременно, когда проблема будет решена.
- Почту легко переслать тому, кто разбирается в обсуждаемом деле или ответственен за конкретные службы.

Все это веские причины использовать электронную почту для связи по вопросам поддержки. Однако есть у электронной почты и недостатки:

- Если проблема касается самой почтовой системы или у пользователя что-то не ладится с электронной почтой, то для связи нужно применять что-то другое.
- Пользователи могут и будут писать в сообщениях все что угодно. Нет никакой гарантии, что в письме будет информация, которая пригодится для решения проблемы или помощи пользователю. Возможно, вы даже не поймете, зачем нужно было такое сообщение. Это приводит к головоломке, о которой стоит поговорить в данном разделе.

Мое любимое письмо в службу поддержки приведу точно в таком виде, в каком оно было получено, за исключением имени автора:

```
Date: Sat, 28 Sep 1996 12:27:35 -0400 (EDT)
From: Special User <user@example.com>
To: systems@example.com
Subject: [Req. #9531] printer help

something is wrong and I have know idea what
(что-то случилось, и я не имею понятия, что именно)
```

Если бы пользователь не упомянул слово «принтер» в теме сообщения, не было бы никаких указаний на то, с чего начать, и нам, вероятно, пришлось бы думать, что и впрямь случилось нечто ужасное. Конечно, это самый крайний случай. Чаще вы будете получать примерно такую почту:

```
From: Another user <user2@example.com>
Subject: [Req #14563] broken macine
To: systems@example.com
Date: Wed, 11 Mar 1998 10:59:42 -0500 (EST)

С машиной krakatoa.example.com  происходит что-то не то
```

Пользователи посылают подобные письма, лишенные содержания, не со зла. Мне кажется, что корень всех бед в полном несоответствии представлений о компьютерной среде у пользователей и системных администраторов.

Для большинства пользователей видимая структура компьютерной среды ограничена клиентской машиной, на которой они работают, соседним принтером и их хранилищем данных (т. е. домашним каталогом). Для системного администратора структура совсем иная. Он видит ряд серверов, предоставляющих услуги клиентам, у каждого из которых может быть множество периферийных устройств. На каждой машине может быть установлено различное программное обеспечение и все они могут находиться в различном состоянии (системная загрузка, конфигурация и т. д.).

Для пользователя вопрос «С какой машиной проблемы?» кажется странным. Они говорят об *одном* компьютере, о том, на котором работают *сейчас*. Неужели это не очевидно? Системному администратору столь же странной кажется просьба «помогите с принтером»; в конце концов, он следит за многими принтерами.

Также обстоит дело и со спецификой проблемы. Системные администраторы всего мира каждый день скрежещут зубами, получая почту, в которой сказано: «Мой компьютер не работает, не могли бы вы помочь мне?». Они знают, что «не работает» может относиться к множеству симптомов, у каждого из которых есть свои причины. Для пользователя, столкнувшегося с тремя зависаниями компьютера за неделю, слова «не работает» выглядят вполне конкретными.

Один из способов разобраться с таким расхождением – четко определить, какие данные посылать в сообщениях. На некоторых сайтах пользователь должен послать сообщение о проблеме, употребляя определенную форму или приложение. Беда такого подхода в том, что пользователи редко испытывают удовольствие от лишних движений мышью и нажатий клавиш, необходимых только для того, чтобы сообщить о проблеме или задать вопрос. Чем больше усилий нужно приложить, тем меньше вероятность, что кто-то воспользуется таким механизмом. Неважно, насколько хорошо продумана форма и какой у нее дизайн, если никто не захочет ею пользоваться. Вопросы в коридорах снова станут нормой. Снова вернулись назад?

Что ж, если применить Perl, может быть и нет. Perl наверняка поможет увеличить количество нормальной почты и поучаствовать в процессе поддержки. Один из первых шагов системного администратора – выяснить местоположение: «Где случилась проблема? С каким принтером? С каким компьютером?». И так далее.

Вот костяк программы, которую я назвал *suss,* представляющая собой «скелет» для сбора информации. Программа изучает сообщение и пытается выяснить, с какой машиной оно связано. В результате часто

можно определить имя узла для писем из категории «С моим компьютером проблемы», не вступая ради этого в дальнейшую беседу с рассеянным пользователем. Имя узла – хорошая отправная точка в процессе поиска возникших проблем.

Программа *suss* применяет очень простой алгоритм для разгадывания имени узла (обычно, поиск в хэше для каждого слова из сообщения). Сначала изучается тема сообщения, затем его тело и, наконец, выполняется поиск по заголовкам *Received:*. Вот упрощенная версия, считывающая файл */etc/hosts*, чтобы определить имена узлов:

```
use Mail::Internet;
$localdomain = ".example.com";

# считываем файл /etc/hosts
open(HOSTS,"/etc/hosts") or die "Не могу открыть файл узлов\n";
while(defined($_ = <HOSTS>)){
    next if /^#/;         # пропускаем комментарии
    next if /^$/;         # пропускаем пустые строки
    next if /monitor/i;   # пример вводящего в заблуждение узла

    # выделяем первое имя узла и переводим его в нижний регистр
    $machine = lc((split)[1]);
    $machine =~ s/\Q$localdomain\E$//oi; # удаляем имя домена
    $machines{$machine}++ unless $machines{$machine};
}

# анализируем сообщение
$message = new Mail::Internet \*STDIN;
$message->head->unfold();

# проверяем тему сообщения
my $subject = $message->head->get('Subject');
$subject  =~ s/[.,;?]//g;
for (split(/\s+/,$subject)) {
    if (exists $machines{lc $_}) {
      print "subject: $_\n";
      $found++;
    }
}
exit if $found;

# проверяем тело сообщения
chomp(my @body = @{$message->body()});
my $body = join(" ",@body);
$body =~ s/[^\w\s]/ /g;              # удаляем знаки пунктуации
@body{split(' ', lc $body)} = ();
for (keys %body) {
    if (exists $machines{lc $_}) {
      print "body: $_\n";
      $found++;
```

```
    }
}
exit if $found;

# последняя надежда: проверяем последнюю строку Received: $received =
(reverse $message->head->get('Received'))[0];
$received =~ s/\Q$localdomain\E//g;
for (split(/\s+/,$received)) {
    if (exists $machines{lc $_}) {
      print "received: $_\n";
    }
}
```

Несколько комментариев к программе:

- Простота проверки становится проблемой, когда мы сталкиваемся с вполне приемлемыми именами узлов, подобных *monitor*. Если имена узлов, являющиеся обычными словами, могут появиться в сообщениях, вам придется либо специально их обработать, как было сделано с `next if /monitor/i`, либо придумать более сложную схему анализа, что предпочтительнее.
- Мы используем срез хэша (`@body{…}`), чтобы ускорить поиск по телу сообщения. За один шаг из сообщения выделяются все уникальные слова. Чтобы разобраться с этой конструкцией, можно прочитать ее изнутри. Во-первых, `split()` возвращает из сообщения список всех «слов» (в нижнем регистре). Эти слова используются как ключи для хэша `%body`. Поскольку имена ключей в хэше повторяться не могут, он будет содержать только уникальные слова из тела сообщения. Именно подобные возможности делают программирование на Perl приятным.

Теперь применим эту программу. Вот два настоящих сообщения в службу поддержки:

```
Received: from strontium.example.com (strontium.example.com [192.168.1.114])
        by mailhub.example.com (8.8.4/8.7.3) with ESMTP id RAA27043
        for <systems>; Thu, 27 Mar 1997 17:07:44 -0500 (EST)
From: User Person <user@example.com>
Received: (user@localhost)
        by strontium.example.com (8.8.4/8.6.4) id RAA10500
        for systems; Thu, 27 Mar 1997 17:07:41 -0500 (EST)
Message-Id: <199703272207.RAA10500@strontium.example.com>
Subject: [Req #11509] Monitor
To: systems@example.com
Date: Thu, 27 Mar 1997 17:07:40 -0500 (EST)

Hi,
My monitor is flickering a little bit and it is tiresome
when working with it to much.
Is it possible to fix it or changing the monitor?
```

```
Thanks.

User.

-------------------------------------
Received: from example.com (user2@example.com [192.168.1.7])
        by mailhost.example.com (8.8.4/8.7.3) with SMTP id SAA00732
        for <systems@example.com>; Thu, 27 Mar 1997 18:34:54 -0500 (EST)
Date: Thu, 27 Mar 1997 18:34:54 -0500 (EST)
From: Another User <user2@example.com>
To: systems@example.com
Subject: [Req #11510] problems with two computers
Message-Id: <Pine.SUN.3.95.970327183117.23440A-100000@example.com>

In Jenolen (in room 292), there is a piece of a disk stuck in it. In
intrepid, there is a disk with no cover (or whatever you call that silver
thing) stuck in it. We tried to turn off intrepid, but it wouldn't work. We
(the proctor on duty and I) tried to get the disk piece out, but it didn't
work. The proctor in charge decided to put signs on them saying 'out of
order'

AnotherUser
```

После запуска программы для этих двух сообщений мы получили:

```
received: strontium
```

и:

```
body: jenolen
body: intrepid
```

Оба узла были найдены верно и для этого понадобился лишь небольшой отрывок простого кода. Шагнем дальше и предположим, что поступило такое письмо:

```
Received: from [192.168.1.118] (buggypeak.example.com [192.168.1.118])
        by mailhost.example.com (8.8.6/8.8.6) with SMTP id JAA16638
        for <systems>; Tue, 4 Aug 1998 09:07:15 -0400 (EDT)
Message-Id: <v02130502b1ecb78576a9@[192.168.1.118]>
Date: Tue, 4 Aug 1998 09:07:16 -0400
To: systems@example.com
From: user@example.com (Nice User)
Subject: [Req #15746] printer

Could someone please persuade my printer to behave and print like a nice
printer should?  Thanks much :)

-Nice User.
```

Пользователь, должно быть, не знает, что вы «пасете стадо» из 30 принтеров. Но можно применить Perl и чуть-чуть наблюдательности,

чтобы сделать умные догадки. Пользователи стараются печатать на принтерах, расположенных ближе всего к тому компьютеру, за которым в данный момент работают. Если бы можно было определить машину, с которой отправлена почта, вероятно, удалось бы вычислить и принтер. Существует много способов получить информацию о связи компьютер-принтер, например, из отдельного файла, из поля в базе данных узлов, о которой упоминалось в главе 5, или даже из службы каталогов LDAP. Вот простой пример, в котором используется простая база данных компьютеров и связанных с ними принтеров:

```
use Mail::Internet;
use DB_File;

$localdomain = ".example.com";

# printdb - это файл Berkeley DB. Ключи - имена узлов, значения - принтеры
$printdb     = "printdb";

# анализируем сообщение
$message = new Mail::Internet \*STDIN;
$message->head->unfold();

# проверяем тему сообщения
my $subject = $message->head->get('Subject');
if ($subject =~ /print(er|ing)?/i){
    # ищем машину-отправителя (ситаем, что используется формат заголовков
Sendmail)
    $received = (reverse $message->head->get('Received'))[0];
    ($host) =
         $received =~ /^from \S+ \((?:\S+@)?(\S+)\Q$localdomain\E \[/;
}

tie %printdb, "DB_File",$printdb  or die "Невозможно подключиться к базе
данных $printdb:$!\n";

print "Проблема на машине $host может быть связана с принтером " .
$printdb{$host} . ".\n";

untie %printdb;
```

Если в теме сообщения упоминаются слова «печать», «принтер» или «напечатать», мы выделяем имя узла из заголовка *Received:*. Для получения этой информации можно применить одно регулярное выражение, т. к. известен формат, используемый для заголовков *Received:* в нашей сети. Зная имя узла, нетрудно найти связанный с ним принтер в базе данных Berkeley DB. Конечный результат выглядит так:

```
Проблема на машине buggypeak может быть связана с принтером hiroshige.
```

Потратив время на изучение структуры своего окружения, вы найдете разные способы получать больше пользы от почты, доставленной в службу поддержки. Приведенные в этом разделе примеры невелики и созданы для того, чтобы заставить вас задуматься о возможностях. Как еще могут помочь программы, читающие почту (возможно, это почта, отправленная другими программами)? Perl предоставляет много способов проанализировать почту, рассмотреть ее в широком контексте и затем использовать найденную информацию.

Информация о модулях из этой главы

Модуль	Идентификатор на CPAN	Версия
`Mac::Glue`	CNANDOR	0.58
`Win32::OLE` (входит в состав ActiveState Perl)	JDB	1.11
`Mail::Mailer` (можно найти в *MailTools*)	GBARR	1.13
`Text::Wrap` (можно найти в *Text-Tabs+Wrap*, также распространяется с Perl)	MUIR	98.112902
`IO::Socket` (можно найти в `IO`, кроме того, распространяется с Perl)	GBARR	1.20
`Mail::Internet` (можно найти в *MailTools*)	GBARR	1.13
`Mail::Header` (можно найти в *MailTools*)	GBARR	1.13
`Mail::Folder::Mbox` (можно найти в `Mail::Folder`)	KJOHNSON	0.07
`Socket` (распространяется с Perl)		
`BerkeleyDB`	PMQS	0.10
`Net::Telnet`	JROGERS	3.01
`DB_File` (распространяется с Perl)	PMQS	1.72

Рекомендуемая дополнительная литература

«Advanced Perl Programming», Sriram Srinivasan (O'Reilly, 1997) – в книге есть хороший раздел о программировании серверов.

«Effective Perl Programming», Joseph Hall, Randal Schwartz (Addison Wesley, 1998) – полезная книга, в которой можно найти множество идиом Perl.

http://www.cauce.org/ – сайт от Coalition Against Unsolicited Email (коалиция против непрошеной почты). Существует много сайтов, посвященных борьбе со спамом; этот сайт неплох для начала. Здесь можно найти ссылки на множество других ресурсов, включая те, на которых описан подробный анализ почтовых заголовков.

http://www.eudora.com/developers/scripting.html – содержит информацию по Eudora и ссылки на другие источники по AppleScript.

http://www.microsoft.com и *http://msdn.microsoft.com* – содержат информацию по «MAPI», «active messaging» и «CDO». Названия этой технологии уже менялись дважды, так что я не решаюсь привести точную ссылку. На сайтах Microsoft содержится масса полезной информации (особенно в разделе про библиотеки MSDN) по этим темам, но она постоянно перемещается с места на место.

«Perl Cookbook», Tom Christiansen, Nathan Torkington (O'Reilly, 1998). В этой книге также рассматриваются вопросы программирования серверов.

«RFC821:Simple Mail Transfer Protocol», J. Postel, 1982.

«RFC822:Standard for the format of ARPA Internet text messages», D. Crocker, 1982.

«RFC954:NICNAME/WHOIS», K. Harrenstien, M. Stahl, E. Feinler, 1985.

- *Текстовые журналы*
- *Двоичные журналы*
- *Данные с состоянием и без*
- *Проблемы с пространством на диске*
- *Анализ журналов*
- *Информация о модулях из этой главы*
- *Рекомендуемая дополнительная информация*

9

Журналы

Если бы эта книга не была посвящена системному администрированию, то было бы странно уделять файлам журналов целую главу. Но у системных администраторов особые отношения с журналами. Как доктор Дулитл, который мог говорить с животными, системные администраторы должны уметь общаться с огромным «зоопарком» программного и аппаратного обеспечения. Большая часть этого общения проходит через журналы, так что необходимо быть «лингвистом», разбирающимся в журналах. В этом очень сильно может помочь Perl.

Невозможно рассмотреть все мыслимые способы обработки и анализа журналов. Целые книги были посвящены лишь статистическому анализу подобных данных. Цель этой главы в том, чтобы дать читателю представление о некоторых общих подходах и инструментах Perl, пробудив в нем интерес к дальнейшему их изучению.

Текстовые журналы

Журналы бывают разных типов, следовательно, нам нужно использовать различные подходы к их обработке. Самые распространенные журналы – полностью состоящие из строк текста. Популярные серверные пакеты, такие как Apache (веб), INN (новости Usenet) и Sendmail (электронная почта) записывают в журналы огромное количество текста. Большая часть журналов на Unix-машинах выглядит одинаково, потому что все они создаются одной и той же программой, известной под именем *syslog*. Файлы, созданные *syslog*, можно считать обычными текстовыми файлами.

Вот простая программа на Perl, ищущая слово «error» в текстовом файле журнала:

```
open(LOG,"logfile") or die "Невозможно открыть журнал:$!\n";
while(<LOG>){
print if /\berror\b/i;
}
close(LOG);
```

Тем, кто хорошо знает Perl, вероятно не терпится сократить ее до одной строки. Пожалуйста:

```
perl -ne 'print if /\berror\b/i' logfile
```

Двоичные журналы

Иногда не просто писать программы, имеющие дело с журналами. Вместо приятных на вид, легко анализируемых текстовых строк, некоторые средства ведения журналов создают двоичные файлы патентованного формата, которые нельзя проанализировать при помощи одной строки кода на Perl. К счастью, Perl не боится таких напастей. Рассмотрим несколько подходов к работе с такими файлами. Ниже приведены два различных примера двоичных журналов: файл *wtmp* в Unix и журналы событий NT/2000.

В главе 3 «Учетные записи пользователей» мы упоминали о регистрации пользователей для работы на машине с Unix. В большинстве Unix-систем регистрация в системе и завершение работы с ней регистрируются в файле *wtmp*. Если нужно узнать о «привычках» пользователя относительно регистрации (например, на какой машине он обычно регистрируется?), то необходимо обратиться к этому файлу.

В NT/2000 журналы событий играют более обобщенную роль. Они используются для регистрации практически всех событий, происходящих на машине, включая регистрацию работы пользователей, сообщения операционной системы, события системы безопасности и т. д. Их роль аналогична роли службы *syslog* в Unix.

Использование unpack()

В Perl существует функция `unpack()`, специально созданная для анализа двоичных данных и структур. Давайте посмотрим, как ее можно использовать для работы с файлами *wtmp*. Формат *wtmp* отличается в различных системах Unix. В следующем примере мы будем иметь дело с файлами *wtmp* из SunOS 4.1.4 и Digital Unix 4.0, поскольку они достаточно просты. Вот как выглядит текстовое представление первых трех записей в файле *wtmp* из SunOS 4.1.4:

```
0000000    ~  \0  \0  \0  \0  \0  \0  \0   r   e   b   o   o   t  \0  \0
0000020   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000040    ,   /   ;   4   c   o   n   s   o   l   e  \0   r   o   o   t
0000060   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
```

```
0000100   \0  \0  \0  \0   ,   /   ; 203   c   o   n   s   o   l   e  \0
0000120   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000140   \0  \0  \0  \0  \0  \0  \0  \0   ,   /   < 230
```

Если вы еще не знакомы со структурой этого файла, подобный «ASCII-дамп» (так он называется) данных выглядит как некий полуслучайный мусор. Как же нам разобраться со структурой этого файла?

Самый простой способ понять формат этого файла – заглянуть в исходники программ, читающих и пишущих в него. Тех, кто не знаком с языком C, эта задача может смутить. К счастью, нет необходимости разбираться и даже смотреть в большую часть кода; достаточно разобраться с частью, определяющей формат файла.

Все программы операционной системы, читающие и пишущие в файл *wtmp*, берут определение файла из одного коротенького включаемого файла C, который скорее всего расположен в */usr/include/utmp.h*. Интересующая нас часть файла начинается с определения структуры данных C, которая будет использоваться для хранения информации. Если мы поищем `struct utmp {`, то найдем нужную нам часть. Строки, следующие за `struct utmp {`, определяют каждое поле этой структуры. Каждая из этих строк сопровождается комментарием в стиле `/* text */`.

Чтобы почувствовать, насколько могут отличаться две различные версии *wtmp*, сравним отрывки из *utmp.h* для двух операционных систем:

SunOS 4.1.4:

```
struct utmp {
        char    ut_line[8];     /* tty name */
        char    ut_name[8];     /* user id */
        char    ut_host[16];    /* host name, if remote */
        long    ut_time;        /* time on */
};
```

Digital Unix 4.0:

```
struct utmp {
        char    ut_user[32];    /* User login name */
        char    ut_id[14];      /* /etc/inittab id- IDENT_LEN in init */
        char    ut_line[32];    /* device name (console, lnxx) */
        short   ut_type;        /* type of entry */
        pid_t   ut_pid;         /* process id */
        struct exit_status {
            short e_termination; /* Process termination status */
            short e_exit;        /* Process exit status */
        } ut_exit;               /* The exit status of a process
                                 * marked as DEAD_PROCESS.
                                 */
        time_t  ut_time;         /* time entry was made */
        char    ut_host[64];     /* host name same as MAXHOSTNAMELEN */
};
```

В этих файлах есть все, что требуется для написания функции unpack(), которая в качестве первого аргумента принимает шаблон формата данных и с помощью этого шаблона определяет, как разобрать двоичные (обычно) данные, переданные во втором аргументе. unpack() разобьет данные так, как это указано, и вернет список, каждый элемент которого соответствует элементу шаблона.

Давайте построим шаблон по кусочкам, принимая за основу структуру на C из файла *utmp.h* в SunOS. Многие буквы разрешается использовать в шаблонах и здесь рассказывается именно о них, но вообще-то вы должны обратиться к разделу pack() из руководства *perlfunc* за подробными разъяснениями. Создание шаблонов – не всегда простое занятие; периодически компиляторы C дополняют поля структуры для того, чтобы выровнять их по требуемой границе памяти. Команда *pstruct*, входящая в состав Perl, часто помогает справиться с подобными особенностями.

С нашим форматом данных таких сложностей не возникает. Посмотрите на анализ файла *utmp.h* (табл. 9.1).

Таблица 9.1. Преобразование кода на C из utmp.h в шаблон unpack( )

Код на C	Шаблон unpack()	Буква шаблона/повтор
`char ut_line[8];`	A8	Строка ASCII (дополнена пробелами) длиной 8 байт
`char ut_name[8];`	A8	Строка ASCII (дополнена пробелами) длиной 8 байт
`char ut_host[16];`	A16	Строка ASCII (дополнена пробелами) длиной 16 байт
`long ut_time;`	l	«Длинное» целое значение со знаком (может и не совпадать с размером значения «long» на конкретной машине)

Шаблоны созданы, теперь используем их в настоящей программе:

```
# шаблон, который мы собираемся передать unpack()
$template = "A8 A8 A16 l";
# используем pack(), чтобы определить размер (в байтах) каждой записи
$recordsize = length(pack($template,()));

# открываем файл
open(WTMP,"/var/adm/wtmp") or die "Невозможно открыть wtmp:$!\n";

# считываем его по одной записи
while (read(WTMP,$record,$recordsize)) {
    # распаковываем, используя шаблон
    ($tty,$name,$host,$time)=unpack($template,$record);
    # специальным образом обрабатываем записи с двоичным нулем (см. ниже)
    if ($name and substr($name,0,1) ne "\0"){
        print "$tty:$name:$host:" ,
```

```
                scalar localtime($time),"\n";
        }
        else {
            print "$tty:(logout):(logout):",
                scalar localtime($time),"\n";
        }
    }

    # закрываем файл
    close(WTMP);
```

Вот как выглядит вывод этой маленькой программы:

```
~:reboot::Mon Nov 17 15:24:30 1997
:0:dnb::0:Mon Nov 17 15:35:08 1997
ttyp8:user:host.mcs.anl.go:Mon Nov 17 18:09:49 1997
ttyp6:dnb:limbo-114.ccs.ne:Mon Nov 17 19:03:44 1997
ttyp6:(logout):(logout):Mon Nov 17 19:26:26 1997
ttyp1:dnb:traal-22.ccs.neu:Mon Nov 17 23:47:18 1997
ttyp1:(logout):(logout):Tue Nov 18 00:39:51 1997
```

Приведем пару комментариев:

- В SunOS завершение работы с терминалов определенного типа отмечается символом с кодом 0 в первой позиции, поэтому:

```
if ($name and substr($name,1,1) ne "\0"){
```

- `read()` принимает в качестве третьего аргумента количество байт, которые нужно прочесть. Вместо того чтобы жестко определить размер записи как «32», мы воспользовались удобным свойством функции `pack()`. Если этой функции передать пустой список, то она возвращает пустую или заполненную пробелами строку размером, совпадающим с размером записи. Это позволяет передать функции `pack()` произвольный шаблон и узнать ее размер:

```
$recordsize = length(pack($template,()));
```

Вызов внешней программы

Работа с файлами *wtmp* – настолько распространенная задача, что в Unix есть специальная команда под названием *last*, предназначенная для вывода двоичных файлов в формате, удобном для человека. Вот образец ее вывода, показывающий примерно те же данные, что и в предыдущем примере:

```
dnb       ttyp6    traal-22.ccs.neu Mon Nov 17 23:47 - 00:39  (00:52)
dnb       ttyp1    traal-22.ccs.neu Mon Nov 17 23:47 - 00:39  (00:52)
dnb       ttyp6    limbo-114.ccs.ne Mon Nov 17 19:03 - 19:26  (00:22)
user      ttyp8    host.mcs.anl.go Mon Nov 17 18:09 - crash (27+11:50)
dnb       :0       :0               Mon Nov 17 15:35 - 17:35 (4+02:00)
reboot    ~                         Mon Nov 17 15:24
```

Мы свободно можем вызывать программы, такие как *last* из Perl. Эта программа выводит все уникальные имена пользователей, найденные в текущем файле *wtmp*:

```
# местоположение команды last
$lastexec = "/usr/ucb/last";

open(LAST,"$lastexec|") or die "Невозможно запустить $lastexec:$!\n";
while(<LAST>){
    $user = (split)[0];
    print "$user","\n" unless exists $seen{$user};
    $seen{$user}='';
}
close(LAST) or die "Невозможно правильно закрыть канал:$!\n";
```

Так зачем же применять этот метод, если `unpack()` делает все, что нам нужно? Из-за переносимости. Мы уже продемонстрировали, что формат файла *wtmp* в различных операционных системах отличается. Ко всему прочему, производитель может изменить формат *wtmp*, а это приведет к тому, что шаблоном `unpack()` в его существующем виде нельзя будет пользоваться.

Но вы *можете* рассчитывать на то, что команда *last*, читающая данный формат, будет присутствовать на вашей системе, независимо от каких-либо изменений формата. В случае применения метода `unpack()` придется создать и поддерживать различные строки шаблонов для каждого формата файла *wtmp*, который планируется использовать.[1]

Самый большой недостаток такого метода по сравнению с `unpack()` — это увеличение сложности анализа полей, выполняемого в программе. В случае с `unpack()` все необходимые поля извлекаются автоматически. При использовании *last* можно столкнуться с данными, которые сложно разобрать при помощи `split()` или регулярных выражений:

```
user   console                      Wed Oct 14 20:35 - 20:37  (00:01)
user   pts/12       208.243.191.21  Wed Oct 14 09:19 - 18:12  (08:53)
user   pts/17       208.243.191.21  Tue Oct 13 13:36 - 17:09  (03:33)
reboot system boot                  Tue Oct  6 14:13
```

На первый взгляд, не легко разобраться с полями, но любая программа, анализирующая подобный вывод, должна уметь правильно обрабатывать пропуски в первой и четвертой строках. Можно по-прежнему использовать `unpack()`, чтобы разделить эти данные, так как поля в нем имеют фиксированную ширину, но это не всегда возможно.

---

[1] Здесь опущены кое-какие подробности, потому что все равно надо следить за тем, где именно расположена команда *last* на каждой операционной системе, и следить за изменениями в формате вывода программы.

Использование API операционной системы для ведения журналов

Давайте перейдем к службе Event Log Service Windows NT/2000, чтобы рассмотреть этот подход. Как мы уже упоминали, в этом случае, к сожалению, журналы хранятся не в текстовых файлах. Самый лучший и единственный поддерживаемый способ, позволяющий добраться до этих данных, заключается в применении набора специальных API-вызовов. Большинство пользователей для получения этих данных полагаются на программу *Event Viewer* (рис. 9.1).

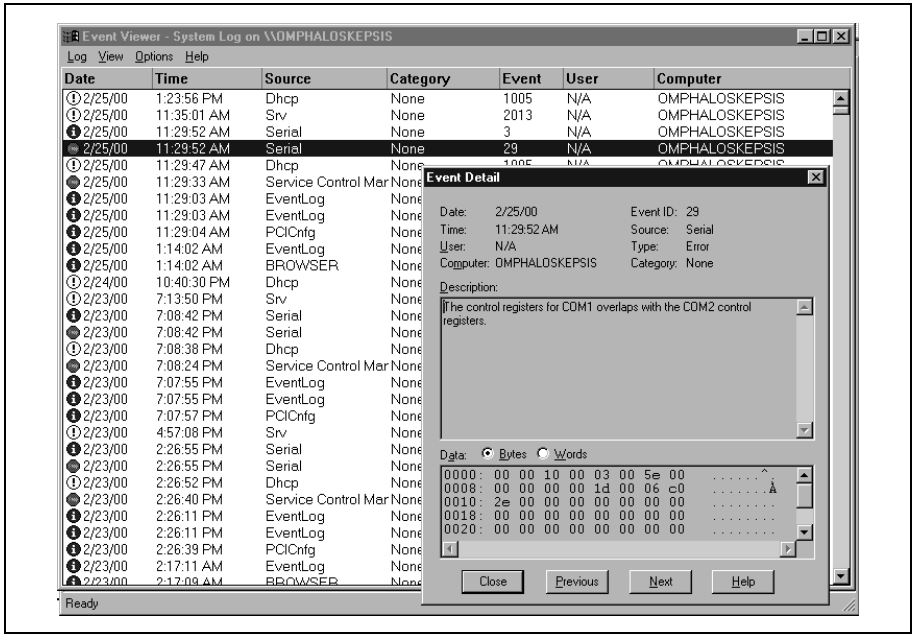

Рис. 9.1. *Программа Event Viewer в NT4*

К счастью, существует модуль, написанный Джесси Доэрти (Jesse Dougherty) (и обновленный Мартином Поли (Martin Pauley) и Бретом Гиддингсом (Bret Giddings)), обеспечивающий простой доступ к API-вызовам Event Log.[1] Вот простая программа, которая выводит список событий из журнала *System* в формате, подобном *syslog*. Позже мы подробно рассмотрим более сложную версию этой программы.

```
use Win32::EventLog;
# у каждого события есть тип, вот как выглядят самые
# распространенные типы
```

[1] Информацию из журналов в Windows 2000 можно получить средствами инструментария WMI (Window Management Instrumentation), о котором мы упоминали в главе 4. Модуль `Win32::EventLog` проще использовать и понять.

```
%type = (1  => "ERROR",
         2  => "WARNING",
         4  => "INFORMATION",
         8  => "AUDIT_SUCCESS",
         16 => "AUDIT_FAILURE");

# если это значение установлено, мы также получаем полный текст
# каждого сообщения при каждом вызове Read()
$Win32::EventLog::GetMessageText = 1;

# открываем журнал событий System
$log = new Win32::EventLog("System")
  or die "Невозможно открыть системный журнал:$^E\n";

# читаем его по одной записи, начиная с первой
while ($log->Read((EVENTLOG_SEQUENTIAL_READ|EVENTLOG_FORWARDS_READ),
            1,$entry)){
    print scalar localtime($entry->{TimeGenerated})." ";
    print $entry->{Computer}."[".($entry->{EventID} &
          0xffff)."] ";
    print $entry->{Source}.":".$type{$entry->{EventType}};
    print $entry->{Message};
}
```

В NT/2000 существуют также утилиты, работающие из командной строки, такие как *last*, выводящие события из журнала в текстовом виде. Позже мы посмотрим на эти утилиты в действии.

Данные с состоянием и без

Помимо формата, в котором хранятся данные из журналов, также важно подумать о содержимом этих файлов, поскольку на наши действия повлияет и то, *что* из себя представляют эти данные и *как* они представлены. Когда речь идет о содержимом файлов журналов, часто можно разделить данные, *имеющие состояние*, и данные, *состояния не имеющие*. Рассмотрим пару примеров, которые помогут разобраться в этих различиях.

Вот отрывок журнала веб-сервера Apache. В каждой строке представлен запрос к веб-серверу:

```
esnet-118.dynamic.rpi.edu - - [13/Dec/1998:00:04:20 -0500] "GET home/u1/tux/
tuxedo05.gif

HTTP/1.0" 200 18666 ppp-206-170-3-49.okld03.pacbell.net - - [13/Dec/
1998:00:04:21 -0500] "GET home/u2/news.htm

HTTP/1.0" 200 6748 ts007d39.ftl-fl.concentric.net - - [13/Dec/1998:00:04:22 -
0500] "GET home/u1/bgc.jpg HTTP/1.1" 304 -
```

А вот несколько строк из журнала демона принтера:

```
Aug 14 12:58:46 warhol  printer: cover/door open
```

```
Aug 14 12:58:58 warhol  printer: error cleared
Aug 14 17:16:26 warhol  printer: offline or intervention needed
Aug 14 17:16:43 warhol  printer: error cleared
Aug 15 20:40:45 warhol  printer: paper out
Aug 15 20:40:48 warhol  printer: error cleared
```

В обоих случаях каждая строка из журнала не зависит от других строк журнала. Можно найти шаблоны или сгруппировать вместе строки, собирая статистику, но в этих данных нет ничего, что связывало бы между собой записи из журнала.

Теперь давайте рассмотрим несколько подправленных записей из журнала *sendmail*:

```
Dec 13 05:28:27 mailhub sendmail[26690]: FAA26690:
from=<user@has.a.godcomplex.com>, size=643, class=0, pri=30643, nrcpts=1,
msgid=<199812131032.CAA22824@has.a.godcomplex.com>, proto=ESMTP,
relay=user@has.a.godcomplex.com [216.32.32.176]

Dec 13 05:29:13 mailhub sendmail[26695]: FAA26695:
from=<root@host.ccs.neu.edu>, size=9600, class=0, pri=39600, nrcpts=1,
msgid=<199812131029.FAA15005@host.ccs.neu.edu>, proto=ESMTP,
relay=root@host.ccs.neu.edu [129.10.116. 69]

Dec 13 05:29:15 mailhub sendmail[26691]: FAA26690: to=<user@ccs.neu.edu>,
delay=00:00:02, xdelay=00:00:01, mailer=local, stat=Sent

Dec 13 05:29:19 mailhub sendmail[26696]: FAA26695: to="|IFS=' '&&exec /usr/
bin/procmail -f-||exit 75 #user", ctladdr=user (6603/104), delay=00:00:06,
xdelay=00:00:06, mailer=prog, stat=Sent
```

В отличие от предыдущих примеров, на этот раз между строчками файла существует определенная связь, которая наглядно показана на рис. 9.2.

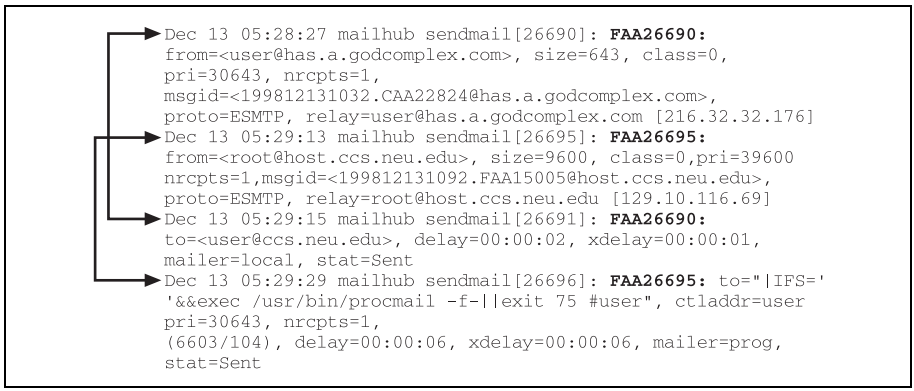

***Рис. 9.2.** Связанные записи из журнала sendmail*

Каждая строка имеет как минимум одну парную запись, в которой указаны источник и получатель каждого сообщения. Когда сообщение попадает в систему, ему присваивается уникальный идентификатор, выделенный на рисунке жирным шрифтом, нужный для «опознания» этого сообщения. Идентификатор сообщения позволяет нам связать соответствующие строки из журнала, определяя существование или «состояние» сообщения между записями в журнале.

Иногда нам нужно знать «расстояние» между переходами. Возьмем, к примеру, файл *wtmp*, рассмотренный ранее в этой главе. Нас интересует не только то, когда пользователи регистрируются в системе и завершают с ней работу (два вида смены состояния в журнале), но и время, прошедшее между этими двумя событиями, т. е. время, в течение которого они были зарегистрированы.

В более сложных журналах могут существовать и другие особенности. Вот выдержки из журнала почтового сервера, находящегося в режиме отладки. Имена и IP-адреса изменены во избежание недоразумений:

```
Jan 14 15:53:45 mailhub popper[20243]: Debugging turned on
Jan 14 15:53:45 mailhub popper[20243]: (v2.53) Servicing request from
"client" at 129.X.X.X
Jan 14 15:53:45 mailhub popper[20243]: +OK QPOP (version 2.53) at mailhub
starting.
Jan 14 15:53:45 mailhub popper[20243]: Received: "USER username"
Jan 14 15:53:45 mailhub popper[20243]: +OK Password required for username.
Jan 14 15:53:45 mailhub popper[20243]: Received: "pass xxxxxxxxx"
Jan 14 15:53:45 mailhub popper[20243]: +OK username has 1 message (26627
octets).
Jan 14 15:53:46 mailhub popper[20243]: Received: "LIST"
Jan 14 15:53:46 mailhub popper[20243]: +OK 1 messages (26627 octets)
Jan 14 15:53:46 mailhub popper[20243]: Received: "RETR 1"
Jan 14 15:53:46 mailhub popper[20243]: +OK 26627 octets
<message text appears here>
Jan 14 15:53:56 mailhub popper[20243]: Received: "DELE 1"
Jan 14 15:53:56 mailhub popper[20243]: Deleting message 1 at offset 0 of
length 26627
Jan 14 15:53:56 mailhub popper[20243]: +OK Message 1 has been deleted.
Jan 14 15:53:56 mailhub popper[20243]: Received: "QUIT"
Jan 14 15:53:56 mailhub popper[20243]: +OK Pop server at mailhub signing off.
Jan 14 15:53:56 mailhub popper[20243]: (v2.53) Ending request from "user" at
(client) 129.X.X.X
```

Можно увидеть не только установление соединения («Servicing request from...») и рассоединения («Ending request from...»), но и подробную информацию о том, что происходило в промежутке.

Каждое из этих промежуточных состояний сопровождается также и потенциально полезной информацией о «продолжительности». Если на POP-сервере возникнут какие-либо неполадки, можно будет узнать, сколько времени занимал каждый из приведенных выше шагов.

В случае с FTP-сервером из этих данных можно будет сделать некоторые выводы относительно того, как люди взаимодействуют с вашим сервером. Сколько времени, в среднем, люди проводят на сайте, прежде чем загрузить файлы? Много ли времени они проводят между выполнениями команд? Всегда ли они переходят из одной части сервера в другую, перед тем как загрузить одни и те же файлы? Промежуточные данные могут быть ценным источником информации.

Проблемы с пространством на диске

Недостаток программ, ведущих полезные и подробные журналы, заключается в том, что для хранения этих данных нужно место на диске. Это касается всех трех операционных систем, рассмотренных в данной книге: Unix, MacOS и Windows NT/2000. Среди них, вероятно, в NT/2000 это вызывает меньше всего проблем, потому что центральный механизм ведения журналов имеет встроенную поддержку автоматического отсечения. В MacOS центрального механизма ведения журналов нет, зато можно запустить несколько серверов, которые с удовольствием выведут в журналы достаточно данных, чтобы заполнить пространство на диске, дай им только такую возможность.

Обычно задача поддержания приемлемого размера для журналов ложится на плечи системного администратора. Большинство производителей Unix предоставляют некий механизм управления размерами журналов вместе с операционной системой, но он часто обслуживает только определенный набор журналов на машине. Как только на машине появляется новая служба, ведущая свой отдельный журнал, возникает необходимость подправить (или даже отбросить) используемый механизм.

Ротация журналов

Распространенное решение проблемы с дисковым пространством – ротация журналов. (Необычное решение мы рассмотрим позже в этом разделе). По истечении определенного времени или после того, как будет достигнут определенный размер файла, текущий журнал будет переименован, например, в *logfile.0*. Последующая запись будет производиться в пустой файл. В следующий раз процесс повторяется, но сперва резервный файл (*logfile.0*) переименовывается (например в *logfile.1*). Этот процесс повторяется до тех пор, пока не будет создано определенное количество резервных файлов. После этого самый старый резервный файл удаляется. Вот как выглядит графическое представление такого процесса (рис. 9.3).

Этот метод позволяет отвести под журналы приемлемое конечное дисковое пространство. Обратите внимание на способ ротации журналов и функции Perl, необходимые для выполнения каждого шага (табл. 9.2).

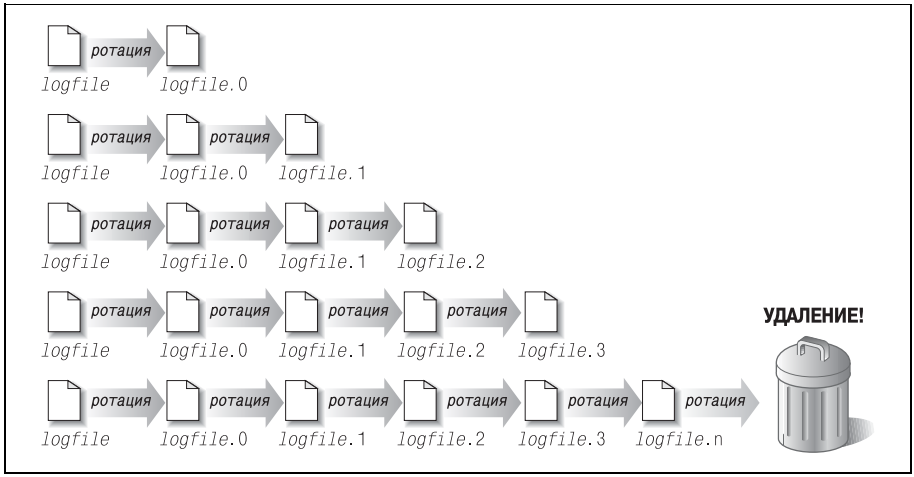

***Рис. 9.3.** Наглядное представление ротации журналов*

Таблица 9.2. Способ ротации журналов из Perl

Процесс	Perl
Переименуйте старые журналы, присвоив им следующий номер.	`rename()` или `&File::Copy::move()` если переносить файлы с одной файловой системы на другую.
Если необходимо, сообщите процессу, создающему файл журнала, о необходимости закрыть текущий файл и приостановить запись до тех пор, пока она не будет разрешена.	`kill()` для программ, принимающих сигналы, `system()` или `` ` ` `` (обратные кавычки), если необходимо вызвать для этого другую программу.
Скопируйте или переместите файлы журналов, которые сейчас использовались, в другой файл.	`&File::Copy` для копирования, `rename()`, чтобы переименовать (или `&File::Copy::move()` при перемещении с одной файловой системы на другую).
Если необходимо, урежьте текущий файл журнала.	`truncate()` или `open(FILE, ">`*filename*`")`.
Если необходимо, пошлите сигнал процессу о необходимости приостановить запись в журнал.	Шаг 2 из этой таблицы.
При желании сожмите или обработайте скопированный файл.	`system()` или обратные кавычки для запуска программы сжатия или другого программного кода, выполняющего обработку.
Удалите самые старые копии файлов.	`stat()`, чтобы выяснить размер файла и даты, `unlink()` для удаления файлов.

На эту тему существует много вариаций. Все, кому не лень, писали собственные сценарии для ротации журналов. Так что не удивительно, что такой модуль существует. Рассмотрим модуль `Logfile::Rotate` Пола Гэмпа (Paul Gampe).

`Logfile::Rotate` использует объектно-ориентированный подход для создания нового экземпляра объекта для журнала и для выполнения методов этого экземпляра. Сначала мы создаем новый экземпляр с заданными параметрами (табл. 9.3).

Таблица 9.3. Параметры Logfile::Rotate

Параметр	Назначение
`File`	Имя файла журнала для ротации
`Count` (необязательный, по умолчанию: `7`)	Число хранимых копий файлов
`Gzip` (необязательный, по умолчанию: путь, найденный при сборке Perl)	Полный путь к программе сжатия *gzip*
`Signal`	Код, выполняемый после завершения ротации, как в шаге 5 (табл. 9.2)

Вот небольшой пример программы, в которой используются эти параметры:

```
use Logfile::Rotate;
$logfile = new Logfile::Rotate(
                File   => "/var/adm/log/syslog",
                Count  => 5,
                Gzip   => "/usr/local/bin/gzip",
                Signal =>
                   sub {
                        open PID, "/etc/syslog.pid" or
                           die "Невозможно открыть pid-файл:$!\n";
                        chomp($pid = <PID>);
                        close PID;
                        # сначала надо проверить допустимость
                        kill 'HUP', $pid;
                        }
            );
```

В результате выполнения этого фрагмента программы указанный журнал будет заблокирован и будет подготовлен модуль для ротации данного журнала. После того как этот объект создан, сама ротация журнала не представляет никакого труда:

```
$logfile->rotate();
undef $logfile;
```

Строка `undef` нужна для того, чтобы убедиться, что файл журнала будет разблокирован после ротации (он заблокирован до тех пор, пока существует объект журнала).

Как говорится в документации, если с модулем работает привилегированный пользователь (например, пользователь *root*), необходимо кое-что учитывать. Во-первых, `Logfile::Rotate` прибегает к системному вызову для запуска программы *gzip*, что является потенциальной дырой в безопасности. Во-вторых, подпрограмма `Signal` должна быть реализована «оборонительным» способом. В предыдущем примере мы не проверяли, что идентификатор процесса, полученный из */etc/syslog.pid*, действительно является идентификатором процесса для *syslog*. Лучше было бы использовать таблицу процессов, о чем мы говорили в главе 4 «Действия пользователей», перед тем как посылать сигнал через `kill()`. Более подробно советы по «защищенному» программированию приведены в главе 1 «Введение».

Кольцевой буфер

Мы только что рассмотрели традиционный способ ротации журналов для контроля за пространством, занимаемым постоянно растущими журналами. Позвольте представить вам более необычный подход, который вы можете добавить в свою копилку.

Вот обычный сценарий: выполняется отладка сервера, который выводит целый поток данных в журнал. Нас интересует только малая часть всех этих данных, вероятно, только те строки, которые выводятся сервером после выполнения определенных тестов на определенном клиенте. Если сохранять в журнале весь вывод, как обычно, это быстро заполнит жесткий диск. Ротация журналов с нужной частотой при таком количестве выводимых данных замедлит работу сервера. Что же делать?

Я написал программу *bigbuffy* для решения этой головоломки, применив совершенно незамысловатый подход. *bigbuffy* считывает построчно поступающие на вход данные. Эти строки сохраняются в кольцевом буфере определенного размера. Когда буфер заполняется, он начинает вновь заполняться с вершины. Этот процесс чтения-записи продолжается до тех пор, пока *bigbuffy* не получит сигнал от пользователя. Получив сигнал, программа сбрасывает текущее содержимое буфера в файл и возвращается в свой нормальный цикл. На диске же остается лишь «окошко» в потоке данных из журнала, в котором показаны только те данные, которые нужны.

bigbuffy можно использовать в паре с программой наблюдения за службой (подобной тем, которые можно найти в главе 5 «Службы имен TCP/IP»). Как только наблюдающая программа (монитор) замечает проблему, она может послать сигнал *bigbuffy* сбросить содержимое буфера на диск. Теперь у нас есть выдержка из журнала, относящаяся

как раз к нужной проблеме (считаем, что буфер достаточно велик и монитор вовремя ее заметил).

Вот упрощенная версия *bigbuffy*. Этот код длиннее примеров из предыдущей главы, но он не очень сложный. Мы будем его использовать в качестве трамплина для разрешения некоторых важных вопросов, таких как блокировка ввода и безопасность:

```
$buffsize = 200; # размер кольцевого буфера по умолчанию (в строчках)

use Getopt::Long;

# анализируем параметры
GetOptions("buffsize=i" => \$buffsize,
           "dumpfile=s" => \$dumpfile);

# устанавливаем обработчик сигнала и инициализируем счетчик
&setup;

# простой цикл прочитать строку - сохранить строку
while (<>){
    # помещаем строку в структуру данных. Заметьте, мы делаем
    # это сначала, даже если получаем сигнал. Лучше записать
    # лишнюю строчку, чем потерять строку данных, если в
    # процессе сброса данных что-то пойдет не так.

    $buffer[$whatline] = $_;

    # куда деть следующую строку?
   ($whatline %= $buffsize)++;

    # если получаем сигнал, сбрасываем текущий буфер
    if ($dumpnow) {
      &dodump();
    }
}

sub setup {
    die "ИСПОЛЬЗОВАНИЕ: $0 [--buffsize=<lines>] --dumpfile=<filename>"
      unless (length($dumpfile));

    $SIG{'USR1'} = \&dumpnow; # устанавливаем обработчик сигнала

    $whatline = 1; # начальная строка кольцевого буфера
}

# простой обработчик сигнала, который просто устанавливает флаг
# исключения, см. perlipc(1)
sub dumpnow {
    $dumpnow = 1;
}

# сбрасываем кольцевой буфер в файл, дописывая его, если он уже
# существует
sub dodump{
    my($line);     # счетчик строк
```

```
    my($exists);   # флаг, существует ли уже файл?
    my(@firststat,@secondstat); # для хранения вывода lstats

    $dumpnow = 0;  # сбрасываем флаг и обработчик сигнала
    $SIG{'USR1'} = \&dumpnow;

    if (-e $dumpfile and (! -f $dumpfile or -l $dumpfile)) {
       warn "ПРЕДУПРЕЖДЕНИЕ: файл для сброса данных существует и не является
обычным текстовым файлом, пропускаем сброс данных.\n";
       return undef;
    }

    # необходимо принять специальные меры предосторожности при
    # дописывании. Следующий набор операторов "if" выполняет
    # несколько проверок при открытии файла для дописывания
    if (-e $dumpfile) {
       $exists = 1;
       unless(@firststat = lstat $dumpfile){
           warn "Невозможно выяснить состояние $dumpfile,
                  пропускаем сброс данных.\n";
           return undef;
       }
       if ($firststat[3] != 1) {
           warn "$dumpfile - жесткая ссылка, пропускаем сброс данных.\n";
           return undef;
       }
    }

    unless (open(DUMPFILE,">>$dumpfile")){
       warn "Невозможно открыть $dumpfile для дописывания,
             пропускаем сброс данных.\n";
       return undef;
    }
    if ($exists) {
       unless (@secondstat = lstat DUMPFILE){
           warn "Невозможно выяснить состояние открытого файла $dumpfile,
                  пропускаем сброс данных.\n";
           return undef;
       }

       if ($firststat[0] != $secondstat[0] or # проверяем номер устройства
           $firststat[1] != $secondstat[1] or # проверяем inode
           $firststat[7] != $secondstat[7])   # проверяем размеры
        {
          warn "ПРОБЛЕМА БЕЗОПАСНОСТИ: lstats не совпадают,
                пропускаем сброс данных.\n";
          return undef;
        }
    }

    $line = $whatline;
    print DUMPFILE "-".scalar(localtime).("-"x50)."\n";
    do {
```

```
            # если буфер не полный
            last unless (defined $buffer[$line]);
            print DUMPFILE $buffer[$line];
            $line = ($line == $buffsize) ? 1 : $line+1;
        } while ($line != $whatline);

        close(DUMPFILE);

        # проталкиваем активный буфер, чтобы не повторять данные
        # при последующем сбросе их в файл
        $whatline = 1;
        $buffer = ();

        return 1;
    }
```

Подобная программа может найти несколько интересных применений.

Блокировка ввода в программах обработки журналов

Я уже говорил, что это упрощенная версия программы *bigbuffy*. С упрощением реализации, в особенности на различных платформах, связана неприятная особенность этой версии: во время сброса данных на диск она не может продолжать считывать ввод. Во время сброса буфера программе, посылающей свой вывод *bigbuffy*, операционная система может дать указание приостановить операции, пока не будет очищен ее буфер вывода. К счастью, сброс данных происходит быстро и окно, в котором это может произойти, будет очень маленьким, но это все равно неприятно.

Вот два возможных решения этой проблемы:

- Переписать *bigbuffy*, используя двойную буферизацию и многозадачность. Вместо одного буфера можно создать два. Во время получения сигнала программа будет записывать журнал во второй буфер до тех пор, пока дочерний процесс или другой поток обрабатывает сброс данных из первого буфера. При получении следующего сигнала буферы вновь меняются местами.
- Переписать *bigbuffy*, чтобы разделить чтение и запись при сбросе данных в файл. Самая простая версия этого подхода предполагает, что несколько строк записываются в файл каждый раз после прочтения новой строки. Это может оказаться не простым делом, если журнал «разорван» и не поступает постоянным потоком. Вряд ли кому-то захочется ждать новую строку вывода для того, чтобы можно было сбросить буфер на диск. Так что придется использовать тайм-ауты или некий механизм внутренних часов, чтобы справиться с этой проблемой.

Оба этих подхода трудно реализовать так, чтобы они были переносимыми между различными платформами, отсюда и упрощенная версия, приведенная в книге.

Безопаснось в программах, обрабатывающих журналы

Вы могли заметить, что в *bigbuffy* операциям открытия файлов вывода и записи в них уделяется внимания больше, чем обычно. Это пример защищенного (оборонительного) стиля программирования, о котором упоминалось уже в разделе «Ротация журналов». Если эта программа предназначена для отладки сервера, почти наверняка она будет запущена привилегированным пользователем. Очень важно продумать все ситуации, которые могут привести к тому, что программой кто-то злоупотребит.

Например, представьте ситуацию, когда файл, в который выводятся данные, был злонамеренно заменен ссылкой на другой файл. Если наивно открыть и записать данные в этот файл, можно обнаружить, что мы перезаписали какой-нибудь важный файл, например */etc/passwd*. Даже если мы проверим файл вывода данных перед самым его открытием, злоумышленник может подменить его перед тем, как мы действительно начнем записывать в него данные. Во избежание таких неприятностей можно использовать такой сценарий:

- Мы проверяем, существует ли файл, в который выводятся данные. Если да, мы выполняем `lstat()`, чтобы получить о нем информацию.
- Открываем файл в режиме дозаписи.
- Перед тем как собственно записать в него данные, мы выполняем `lstat()` для открытого файлового дескриптора и проверяем, тот же это файл, что мы ожидаем, или нет. Если это другой файл (т. е. кто-то заменил его ссылкой прямо перед открытием), мы *не* записываем в него данные и выводим соответствующее предупреждение. Этот шаг позволяет избежать состояния перехвата, о котором говорилось в главе 1.

Если дописывать данные не надо, то можно открыть временный файл со случайным именем (чтобы его нельзя было угадать заранее) и потом переименовать его.

Подобные «уловки» необходимы в большинстве Unix-систем, поскольку первоначально Unix создавался без особой заботы о безопасности. Брешь в безопасности, связанная с символическими ссылками, не является проблемой в NT4, т. к. они являются малоиспользуемой частью подсистемы POSIX, не проблема это и в MacOS, поскольку тут не существует понятия «привилегированный пользователь».[1]

[1] Справедливости ради отметим, что и в NT, и в MacOS существуют собственные слабости в системе защиты. Кроме того, очень много усилий и времени было потрачено на то, чтобы «укрепить» различные дистрибутивы Unix (например, OpenBSD).

Анализ журналов

Некоторые системные администраторы никогда не заходят дальше фазы ротации в своих взаимоотношениях с журналами. До тех пор пока на диске существует информация, необходимая для отладки, они никогда не подумают об использовании информации из журналов в других целях. Хотелось бы намекнуть, что это недальновидный взгляд и что даже краткий анализ журналов может иметь большое значение. Мы рассмотрим несколько подходов, которые можно использовать для анализа журналов из Perl, начиная от самого простого и постепенно продвигаясь к более сложному.

Большая часть примеров из этого раздела использует журналы из Unix, т. к. в средней Unix-системе журналов больше, чем в двух других операционных системах вместе взятых, а применяемые подходы не зависят от операционной системы.

Чтение–подсчет потока

Самый простой подход – обычное «считывание и подсчет». Мы читаем поток данных из журнала, ищем интересующие нас данные и увеличиваем значение счетчика, когда их находим. Вот простой пример, подсчитывающий, сколько раз перегружалась машина, в котором использован файл *wtmpx* из Solaris 2.6:[1]

```
# шаблон для wtmpx из Solaris 2.6, подробности смотрите в
# документации по pack()
$template = "A32 A4 A32 l s s2 x2 l2 l x20 s A257 x";

# определяем размер записи
$recordsize = length(pack($template,()));
# открываем файл
open(WTMP,"/var/adm/wtmpx") or die "Невозможно открыть wtmpx:$!\n";

# считываем по одной записи
while (read(WTMP,$record,$recordsize)) {
    ($ut_user,$ut_id,$ut_line,$ut_pid,$ut_type,$ut_e_termination,
     $ut_e_exit,$tv_sec,$tv_usec,$ut_session,$ut_syslen,$ut_host)=
       unpack($template,$record);

    if ($ut_line eq "system boot"){
        print "rebooted ".scalar localtime($tv_sec)."\n";
        $reboots++;
```

[1] *wtmpx* – это файл, использующий расширенную версию формата *u/wtmp*. Он был создан для регистрации событий без ограничений на длину некоторых полей, существующих в классическом формате (например, 16 символов для имени удаленного узла). В Solaris каждая регистрация в системе и завершение работы регистрируются и в *wtmp*, и в *wtpmx*.

```
    }
}

close(WTMP);
print "Общее число перезагрузок: $reboots\n";
```

Расширим этот подход и рассмотрим пример сбора статистики при помощи Event Log из Windows NT. Как говорилось раньше, механизм ведения журналов в NT хорошо разработан и довольно сложен. Эта сложность несколько пугает начинающих программистов на Perl. Для получения основной информации из журналов мы будем использовать некоторые подпрограммы из модулей для Win32.

Программы в NT и компоненты операционной системы записывают свои действия, регистрируя «события» в одном из журналов событий. Регистрация событий операционной системой сопровождается записью основной информации, например, времени наступления события, имени программы или функции операционной системы, зарегистрировавших событие, типа наступившего события (просто информативное или что-то более серьезное) и т. д.

В отличие от Unix, само описание события, т. е. сообщение, не хранится вместе с записью о событии. Вместо этого в журнал помещается идентификатор EventID. Этот идентификатор содержит ссылку на определенное сообщение, хранящееся в библиотеке (*.dll*). Получить сообщение по идентификатору не просто. Этот процесс требует поиска нужной библиотеки в реестре и ее загрузки вручную. К счастью, этот процесс в текущей версии модуля `Win32::EventLog` выполняется автоматически (ищите `$Win32::EventLog::GetMessageText` в первом примере с использованием `Win32::Eventlog`).

В следующем примере мы сгенерируем простую статистику по числу записей в журнале *System*, содержащую сведения о том, откуда они поступили, и об уровне их важности. Мы напишем эту программу несколько иначе, чем первый пример в этой главе.

Первый наш шаг – загрузить модуль `Win32::EventLog`, обеспечивающий связь между Perl и программами для работы с журналами событий в Win32. Затем мы инициализируем хэш-таблицу, которая будет использоваться для хранения результатов вызовов программ чтения журналов. Обычно Perl заботится об этом за нас, но иногда стоит добавить подобный код ради тех, кто будет впоследствии читать программу. Наконец, мы определяем небольшой список типов событий, который позже будет использоваться для печати статистики:

```
use Win32::EventLog;

my %event=('Length',NULL,
           'RecordNumber',NULL,
           'TimeGenerated',NULL,
           'TimeWritten',NULL,
```

```
            'EventID',NULL,
            'EventType',NULL,
            'Category',NULL,
            'ClosingRecordNumber',NULL,
            'Source',NULL,
            'Computer',NULL,
            'Strings',NULL,
            'Data',NULL,);

# частичный список типов событий, то есть тип 1 -- "Error",
# 2 -- "Warning" и т. д.
@types = ("","Error","Warning","","Information");
```

Наш следующий шаг – открытие журнала событий *System*. `Open()` помещает дескриптор *EventLog* в `$EventLog`, который можно использовать для соединения с этим журналом:

```
Win32::EventLog::Open($EventLog,'System','')
  or die "Невозможно открыть журнал System:$^E\n";
```

Получив этот дескриптор, мы можем использовать его для подсчета событий в журнале и получения номера самой старой записи:

```
$EventLog->Win32::EventLog::GetNumber($numevents);
$EventLog->Win32::EventLog::GetOldest($oldestevent);
```

Эта информация указывается в первом операторе `Read()`, позиционирующем нас прямо перед первой записью. Это эквивалентно переходу в начало файла при помощи функции `seek()`:

```
$EventLog->Win32::EventLog::Read((EVENTLOG_SEEK_READ |
                                  EVENTLOG_FORWARDS_READ),
                                  $numevents + $oldestevent, $event);
```

Теперь в простом цикле прочитываем все записи. Флаг `EVENTLOG_SEQUENTIAL_READ` говорит: «Продолжайте читать с позиции после последней прочитанной записи». Флаг `EVENTLOG_FORWARDS_READ` перемещает нас вперед в хронологическом порядке.[1] Третий аргумент `Read()` – смещение, в данном случае равное 0, потому что мы продолжаем с той же позиции, на которой остановились. Считывая каждую запись, мы записываем в хэш-таблицу счетчиков ее источник (`Source`) и тип события (`EventType`).

```
# обходим в цикле все события, записывая количество различных
# источников (Source) и типов событий (EventTypes)
for ($i=0;$i<$numevents;$i++) {
```

[1] Вот еще один пример, когда программы для работы с журналом событий в Win32 гибче, чем обычно. Наша программа может дойти до конца журнала и затем читать журнал в обратном порядке, если это по какой-то причине необходимо.

```
    $EventLog->Read((EVENTLOG_SEQUENTIAL_READ |
                     EVENTLOG_FORWARDS_READ),
                     0, $event);
    $source{$event->{Source}}++;
    $types{$event->{EventType}}++;
}

# выводим полученные результаты
print "-->Event Log Source Totals:\n";
for (sort keys %source) {
    print "$_: $source{$_}\n";
}
print "-"x30,"\n";
print "-->Event Log Type Totals:\n";
for (sort keys %types) {
    print "$types[$_]: $types{$_}\n";
}
print "-"x30,"\n";
print "Total number of events: $numevents\n";
Мои результаты выглядят так:
--> Event Log Source Totals:
Application Popup: 4
BROWSER: 228
DCOM: 12
Dhcp: 12
EventLog: 351
Mouclass: 6
NWCWorkstation: 2
Print: 27
Rdr: 12
RemoteAccess: 108
SNMP: 350
Serial: 175
Service Control Manager: 248
Sparrow: 5
Srv: 201
msbusmou: 162
msi8042: 3
msinport: 162
mssermou: 151
qic117: 2
------------------------------
--> Event Log Type Totals:
Error: 493
Warning: 714
Information: 1014
------------------------------
Total number of events: 2220
```

Как я и обещал, вот пример кода, полагающегося на *last*-подобную программу для вывода содержимого журнала событий. В нем использу-

ется программа *ElDump* Джеспера Лоритсена (Jesper Lauritsen), которую можно загрузить с *http://www.ibt.ku.dk/jesper/JespersNTtools.htm*. *ElDump* похожа на *DumpEl* из NT Resource Kit:

```
$eldump = 'c:\bin\eldump';      # путь к ElDump
# выводим поля данных, разделяя их тильдой (~), и без полного
# текста сообщения (быстрее)
$dumpflags = '-l system -c ~ -M';

open(ELDUMP,"$eldump $dumpflags|") or die "Невозможно запустить
$eldump:$!\n";

print STDERR "Считываем системный журнал.";

while(<ELDUMP>){
    ($date,$time,$source,$type,$category,$event,$user,$computer) =
    split('~');
    $$type{$source}++;
    print STDERR ".";
}
print STDERR "done.\n";

close(ELDUMP);

# для каждого типа события выводим источники и количество
# событий
foreach $type (qw(Error Warning Information
                  AuditSuccess AuditFailure)){
    print "-" x 65,"\n";
    print uc($type)."s by source:\n";
    for (sort keys %$type){
        print "$_ ($$type{$_})\n";
    }
}
print "-" x 65,"\n";
```

Вот выдержка из получаемых данных:

```
ERRORs by source:
BROWSER (8)
Cdrom (2)
DCOM (15)
Dhcp (2524)
Disk (1)
EventLog (5)
RemoteAccess (30)
Serial (24)
Service Control Manager (100)
Sparrow (2)
atapi (2)
i8042prt (4)
-----------------------------------------------------------------
WARNINGs by source:
```

```
BROWSER (80)
Cdrom (22)
Dhcp (76)
Print (8)
Srv (82)
```

Вариация на тему предыдущего примера

Простая вариация предыдущего подхода включает в себя многократный обход данных. Иногда это необходимо в случае с данными большого объема и ситуаций, когда сначала приходится просмотреть все данные, чтобы отличить интересные данные от неинтересных. В плане реализации это означает, что после первого обхода данных надо:

- Перейти обратно к началу потока данных (который может быть файлом) при помощи `seek()`или API-вызова.

или

- Закрыть и вновь открыть дескриптор файла. Зачастую это единственный выбор, когда читаются данные из вывода программы, подобной *last*.

Вот пример, когда такой подход может пригодиться. Представьте, что надо справиться с проблемой в защите, связанной с тем, что кто-то получил несанкционированный доступ к одной из учетных записей. Один из первых вопросов, который приходит на ум, – «был ли получен доступ и к другим учетным записям с той же машины?». Найти полный ответ на такой кажущийся простым вопрос может оказаться сложнее, чем кажется. Давайте попробуем решить эту проблему. Приведенный ниже отрывок кода для SunOS принимает в качестве первого аргумента имя пользователя и необязательное регулярное выражение в качестве второго, чтобы отфильтровать узлы, которые мы хотим проигнорировать:

```
$template       = "A8 A8 A16 l"; # для SunOS 4.1.x
$recordsize     = length(pack($template,()));
($user,$ignore) = @ARGV;

print "-- ищем узлы, с которых регистрировался пользователь $user --\n";
open(WTMP,"/var/adm/wtmp") or die "Невозможно открыть wtmp:$!\n";
while (read(WTMP,$record,$recordsize)) {
    ($tty,$name,$host,$time)=unpack($template,$record);

    if ($user eq $name){
       next if (defined $ignore and $host =~ /$ignore/o);
       if (length($host) > 2 and !exists $contacts{$host}){
           $connect = localtime($time);
           $contacts{$host}=$time;
           write;
       }
    }
```

```
}
print "-- ищем другие соединения с этих узлов --\n";
die "Невозможно перейти в начало wtmp:$!\n"
  unless (seek(WTMP,0,0));

while (read(WTMP,$record,$recordsize)) {
    ($tty,$name,$host,$time)=unpack($template,$record);

    # если это запись не о завершении работы с системой и нас
    # интересует этот узел и это соединение установлено для
    # *другой* учетной записи, тогда записываем эти данные
    if (substr($name,1,1) ne "\0" and
        exists $contacts{$host} and
        $name ne $user){
            $connect = localtime($time);
            write;
    }
}
close(WTMP);

# вот формат вывода, вероятно, его потребуется скорректировать
# в зависимости от шаблона
format STDOUT =
@<<<<<<<<    @<<<<<<<<<<<<<<  @<<<<<<<<<<<<<<<<<<<
$name,$host,$connect
.
```

Сначала программа просматривает файл *wtmp* в поисках записей о регистрации в системе пользователей под «скомпрометированным» именем. По мере нахождения таких записей пополняется хэш, в который записываются имена всех узлов, где регистрировался пользователь под этим именем. Затем программа возвращается к началу файла и просматривает его заново, выполняя на этот раз поиск записей о соединениях с узлов из списка, и выводит совпадения по мере их появления. Не составит труда изменить эту программу так, чтобы она просматривала все файлы из каталога, в котором хранятся файлы ротации журнала *wtmp*.

Единственная проблема этой программы – ее «узкая специализация». Она будет искать только точное совпадение имен узлов. Если злоумышленник регистрировался в системе, используя динамический адрес, получаемый от провайдера (что бывает часто), то очень велика вероятность, что имена узлов будут отличаться при каждом соединении. Тем не менее, даже неполные решения, подобные этому, очень сильно помогают.

Помимо простоты, у рассмотренного нами подхода есть еще преимущества: он быстрее и требует меньших затрат памяти по сравнению с другими методами. Он лучше всего справляется с журналами, в которых регистрируются данные без поддержки состояния, о которых мы

говорили раньше в этой главе. Но иногда, особенно при работе с данными с состоянием, необходимо использовать другие методы.

Процесс «прочитал–запомнил»

Крайность, противоположная предыдущему подходу (там мы «пробегали» по данным как можно быстрее), заключается в чтении их в память и последующей обработке после чтения. Рассмотрим несколько версий этой стратегии.

Для начала приведем простой пример: скажем, у нас есть журнал FTP-сервера и требуется узнать, какие файлы скачивались чаще других. Вот несколько строк из журнала FTP-сервера *wu-ftpd*:

```
Sun Dec 27 05:18:57 1998 1 nic.funet.fi 11868 /net/ftp.funet.fi/CPAN/
MIRRORING.FROM a _ o a cpan@perl.org ftp 0 *
Sun Dec 27 05:52:28 1998 25 kju.hc.congress.ccc.de 269273 /CPAN/doc/FAQs/FAQ/
PerlFAQ.html a _ o a mozilla@ ftp 0 *
Sun Dec 27 06:15:04 1998 1 rising-sun.media.mit.edu 11868 /CPAN/
MIRRORING.FROM b _ o a root@rising-sun.media.mit. edu ftp 0 *
Sun Dec 27 06:15:05 1998 1 rising-sun.media.mit.edu 35993 /CPAN/RECENT.html b
_ o a
root@rising-sun.media.mit.edu ftp 0 *
```

А вот список полей, из которых состоят приведенные выше строки (все подробности о каждом поле ищите в страницах руководства *xferlog(5)* сервера *wu-ftpd*).

Номер поля	Имя поля
0	`current-time` (текущее время)
1	`transfer-time` (время передачи, в секундах)
2	`remote-host` (удаленный узел)
3	`filesize` (размер файла)
4	`filename` (имя файла)
5	`transfer-type` (тип передачи)
6	`special-action-flag` (специальный флаг)
7	`direction` (направление)
8	`access-mode` (режим доступа)
9	`username` (имя пользователя)
10	`service-name` (имя службы)
11	`authentication-method` (меод аутентификации)
12	`authenticated-user-id` (идентификатор аутентифицированного пользователя)

Вот пример программы, сообщающей о том, какие файлы передавались чаще других:

```
$xferlog = "/var/adm/log/xferlog";
open(XFERLOG,$xferlog) or die "Невозможно открыть $xferlog:$!\n";

while (<XFERLOG>){
    $files{(split)[8]}++;
}

close(XFERLOG);

for (sort {$files{$b} <=> $files{$a}||$a cmp $b} keys %files){
    print "$_:$files{$_}\n";
}
```

Мы считываем каждую строку файла, используя имя файла в качестве ключа хэша, и увеличиваем значение для этого ключа. Имя файла выделяется из каждой строки журнала при помощи индекса массива, ссылающегося на определенный элемент списка, возвращенного функцией `split()`:

```
$files{(split)[8]}++;
```

Вы могли заметить, что элемент, на который мы ссылаемся (8), отличается от 8-го поля из списка полей *xferlog*, приведенного выше. Это печальные последствия того, что в оригинальном файле отсутствуют разделители полей. Мы разбиваем строки из журнала по пробелам (что является значением по умолчанию для `split()`), так что дата разбивается на пять отдельных элементов списка.

В этом примере применяется искусный прием – сортировку значений выполняет анонимная функция `sort`:

```
for (sort {$files{$b} <=> $files{$a}||$a cmp $b} keys %files){
```

Обратите внимание, что переменные `$a` и `$b` в первой части расположены не в алфавитном порядке. Это приводит к тому, что `sort` выводит элементы в обратном порядке, т. е. первыми отображаются самые «популярные» файлы. Вторая часть анонимной функции `sort` (`||$a cmp $b`) гарантирует, что файлы с одинаковой популярностью будут перечислены в отсортированном порядке.

Для того чтобы этот сценарий подсчитывал только некоторые файлы и каталоги, можно задать регулярное выражение в качестве первого аргумента для сценария. Например, если добавить

```
next unless /$ARGV[0]/o;
```

в цикл `while()`, можно будет задавать регулярные выражения для ограничения учитываемых файлов.

Давайте посмотрим на другой пример подхода «прочитал-запомнил», в котором используется программа «поиска брешей» из предыдущего раздела. В предыдущем примере выводилась информация только об *успешной* регистрации с сайтов злоумышленника. Узнать о неудавшихся попытках мы не можем. Чтобы получить такую информацию, мы рассмотрим другой файл журнала.

Регулярные выражения

Регулярные выражения – одна из самых важных составляющих анализа журналов. Регулярные выражения используются как сито для отсева интересных данных от данных, не представляющих интереса. В этой главе применяются только основные регулярные выражения, но вы вполне можете создавать более сложные для собственных нужд. Так, применение подпрограмм или технологии создания регулярных выражений из предыдущей главы позволяет использовать их еще более эффективно.

Время, потраченное на получение навыков работы с регулярными выражениями, окупится с лихвой и не раз. Регулярные выражения лучше всего изучать по книге Джеффри Фридла (Jeffrey Friedl) «Mastering Regular Expressions» (Волшебство регулярных выражений) (O'Reilly).

Эта проблема иллюстрирует один из недостатков Unix: информация из журналов в Unix-системах хранится в различных местах и в различных форматах. Для того чтобы справиться с этими различиями, существует не так много инструментов (к счастью, у нас есть Perl). Нередко приходится использовать более одного источника данных для решения подобных задач.

Журнал, который сейчас больше всего нам пригодится, – это журнал, сгенерированный через *syslog* инструментом *tcpwrappers*, который предоставляет программы и библиотеки, позволяющие контролировать доступ к сетевым службам. Любую сетевую службу, например *telnet*, можно настроить так, чтобы все сетевые соединения для нее обрабатывались сначала программой *tcpwrappers*. После того как соединение будет установлено, программа *tcpwrappers* регистрирует попытку соединения через *syslog* и затем либо передает соединение настоящей службе, либо предпринимает некоторые действия (например, разрывает соединение). Решение, разрешить ли данное соединение, основывается на нескольких правилах, введенных пользователем (например, разрешать лишь для некоторых исходящих узлов). *tcpwrappers* также может принять меры предосторожности и послать запрос к DNS-серверу, чтобы убедиться, что соединение устанавливается оттуда, откуда

ожидается. Кроме того, программу можно настроить так, чтобы в журнале регистрировалось имя пользователя, устанавливающего соединение (через протокол идентификации, описанный в RFC931), если это возможно. Более подробное описание *tcpwrappers* можно найти в книге Симсона Гарфинкеля (Simson Garfinkel) и Джина Спаффорда (Gene Spafford) «Practical Unix & Internet Security» (Unix в практическом использовании и межсетевая безопасность) (O'Reilly).

Мы же просто добавим несколько строк к предыдущей программе, в которых просматривается журнал *tcpwrappers* (в данном случае *tcpdlog*) для поиска соединений с подозрительных узлов, найденных нами в *wtmp*. Если добавить этот код в конец предыдущего примера,

```
# местоположение журнала tcpd
$tcpdlog        = "/var/log/tcpd/tcpdlog";
$hostlen        = 16;  # максимальная длина имени узла в файле wtmp

print "-- просматриваем tcpdlog --\n";
open(TCPDLOG,$tcpdlog) or die "Невозможно прочитать $tcpdlog:$!\n";
while(<TCPDLOG>){
    next if !/connect from /; # нас беспокоят только соединения
    ($connecto,$connectfrom) = /(.+):\s+connect from\s+(.+)/;
    $connectfrom =~ s/^.+@//;

    # tcpwrappers может регистрировать имя узла целиком, а не
    # только первые N символов, как некоторые журналы wtmp. В
    # результате необходимо усечь имя узла до той же длины, что
    # и в wtmp файле, если мы собираемся искать имя узла в хэше
    $connectfrom = substr($connectfrom,0,$hostlen);
    print if (exists $contacts{$connectfrom} and
              $connectfrom !~ /$ignore/o);
}
```

то мы получим данные, подобные этим:

```
-- ищем узлы, с которых регистрировался пользователь --
user       host.ccs.neu  Fri Apr  3 13:41:47
-- ищем другие соединения с этих узлов --
user2      host.ccs.neu  Thu Oct  9 17:06:49
user2      host.ccs.neu  Thu Oct  9 17:44:31
user2      host.ccs.neu  Fri Oct 10 22:00:41
user2      host.ccs.neu  Wed Oct 15 07:32:50
user2      host.ccs.neu  Wed Oct 22 16:24:12
-- просматриваем tcpdlog --
Jan 12 13:16:29 host2 in.rshd[866]: connect from user4@host.ccs.neu.edu
Jan 13 14:38:54 host3 in.rlogind[4761]: connect from user5@host.ccs.neu.edu
Jan 15 14:30:17 host4 in.ftpd[18799]: connect from user6@host.ccs.neu.edu
Jan 16 19:48:19 host5 in.ftpd[5131]: connect from user7@host.ccs.neu.edu
```

Читатели могли обратить внимание, что в приведенных выше результатах были замечены соединения, устанавливаемые в различное вре-

мя. В файле *wtmp* были зарегистрированы соединения, установленные в период с 3 апреля по 22 октября, тогда как *tcpwrappers* показывает только январские соединения. Разница в датах говорит о том, что файлы *wtmp* и файлы *tcpwrappers* имеют различную скорость ротации. Необходимо учитывать такие детали, если вы пишете программу, которая полагается на то, что файлы журналов относятся к одному и тому же периоду времени.

В качестве последнего и более сложного примера, демонстрирующего подход «прочитал-запомнил», рассмотрим задачу, требующую объединения данных с состоянием и без него. Для того чтобы получить более полную картину действий на сервере *wu-ftpd*, можно установить соответствие между информацией о регистрации в системе из файла *wtmp* и информацией о передаче файлов, записанной в файле *xferlog* сервера *wu-ftpd*. Было бы здорово увидеть, когда начался сеанс работы с FTP-сервером, когда он закончился и какие файлы передавались в течение этого сеанса.

Вот отрывок вывода программы, которую мы собираемся написать. В нем показаны четыре FTP-сеанса за март. В первом сеансе на машину был передан один файл. В двух других файлы были переданы с этой машины, а в течение последнего сеанса файлов не было передано вообще:

```
Thu Mar 12 18:14:30 1998-Thu Mar 12 18:14:38 1998 pitpc.ccs.neu.ed
        -> /home/dnb/makemod

Sat Mar 14 23:28:08 1998-Sat Mar 14 23:28:56 1998 traal-22.ccs.neu
        <- /home/dnb/.emacs19

Sat Mar 14 23:14:05 1998-Sat Mar 14 23:34:28 1998 traal-22.ccs.neu
        <- /home/dnb/lib/emacs19/cperl-mode.el
        <- /home/dnb/lib/emacs19/filladapt.el

Wed Mar 25 21:21:15 1998-Wed Mar 25 21:36:15 1998 traal-22.ccs.neu
        (no transfers in xferlog)
```

Получить такие данные не очень просто, поскольку приходится добавить данные без информации о состоянии в журнал данных с информацией о состоянии. В журнале *xferlog* приводится только время, когда была совершена передача файла, и узел, участвующий в этой предаче. В журнале *wtmp* приводится информация о соединениях и завершении соединений других узлов с сервером. Давайте посмотрим, как объединить эти два типа данных при помощи подхода «прочитал-запомнил». В этой программе мы определим некоторые переменные, а затем вызовем подпрограммы для выполнения каждой задачи :

```
# для преобразования дата ->время-в-Unix (количество секунд с начала эпохи)
use Time::Local;

$xferlog = "/var/log/xferlog"; # местоположение журнала передачи файлов
```

```
$wtmp = "/var/adm/wtmp";        # местоположение wtmp
$template = "A8 A8 A16 l";      # шаблон для wtmp в SunOS 4.1.4
$recordsize = length(pack($template,())); # размер каждой записи в wtmp
$hostlen = 16;   # максимальная длина имени узла в wtmp
# карта соответствий имени месяца с номером
%month = qw{Jan 0 Feb 1 Mar 2 Apr 3 May 4 Jun 5 Jul 6
            Aug 7 Sep 8 Oct 9 Nov 10 Dec 11};

&ScanXferlog;    # просматриваем журнал передачи файлов
&ScanWtmp;       # просматриваем журнал wtmp
&ShowTransfers;  # приводим в соответствие и выводим информацию о передачах
```

Теперь рассмотрим процедуру, читающую журнал *xferlog*:

```
# просматриваем журнал передачи файлов сервера wu-ftpd и
# заполняем структуру данных %transfers

sub ScanXferlog {
    local($sec,$min,$hours,$mday,$mon,$year);
    my($time,$rhost,$fname,$direction);

    print STDERR "Просматриваю $xferlog...";
    open(XFERLOG,$xferlog) or
       die "Невозможно открыть $xferlog:$!\n";

    while (<XFERLOG>){
       # используем срез массива для выбора нужных полей
      ($mon,$mday,$time,$year,$rhost,$fname,$direction) =
          (split)[1,2,3,4,6,8,11];

       # добавляем к имени файла направление передачи,
       # i - это передача на сервер
      $fname = ($direction eq 'i' ? "-> " : "<- ") . $fname;

       # преобразуем время передачи к формату времени в Unix
      ($hours,$min,$sec) = split(':',$time);
      $unixdate =
         timelocal($sec,$min,$hours,$mday,$month{$mon},$year);

      # помещаем данные в хэш списка списков:
       push(@{$transfers{substr($rhost,0,$hostlen)}},
            [$unixdate,$fname]);
    }
    close(XFERLOG);
    print STDERR "Готово.\n";
}
```

Строка `push()`, вероятно, заслуживает объяснения. В этой строке формируется хэш списка списков, выглядящий примерно так:

```
$transfers{hostname} =
  ([time1, filename1], [time2, filename2],[time3, filename3]...)
```

Ключами хэша %transfers являются имена узлов, инициирующих передачи файлов. При создании каждой записи мы укорачиваем имя узла до максимальной длины, допустимой в *wtmp*.

Для каждого узла мы сохраняем список пар, состоящих из времени передачи файла и его имени. Время сохраняется в «секундах, прошедших с начала эпохи» для упрощения дальнейшего сравнения.[1] Подпрограмма timelocal() из модуля Time::Local помогает выполнить преобразование времени к этому стандарту. Поскольку мы просматриваем журнал передачи файлов, записанный в хронологическом порядке, список пар тоже строится в хронологическом порядке, а это нам пригодится позже.

Теперь перейдем к просмотру *wtmp* :

```
# просматриваем файл wtmp и заполняем структуру @sessions
# информацией о ftp-сеансах
sub ScanWtmp {
    my($record,$tty,$name,$host,$time,%connections);

    print STDERR "Просматриваю $wtmp...\n";
    open(WTMP,$wtmp) or die "Невозможно открыть $wtmp:$!\n";

    while (read(WTMP,$record,$recordsize)) {

        # если запись начинается не с ftp, даже не пытаемся ее
        # разбирать (unpack). ЗАМЕЧАНИЕ: мы получаем зависимость от
        # формата wtmp в обмен на скорость
       next if (substr($record,0,3) ne "ftp");

       ($tty,$name,$host,$time)=unpack($template,$record);

        # если мы находим запись об открытии соединения, мы
        # создаем хэш списка списков. Список списков позже
        # будет использован в качестве стэка.
       if ($name and substr($name,0,1) ne "\0"){
           push(@{$connections{$tty}},[$host,$time]);
       }
       # если мы находим запись о закрытии соединения, пытаемся
       # найти ей пару в записях об открытии соединений,
       # найденных раньше
       else {
               unless (exists $connections{$tty}){
               warn "Найдено только завершение соединения с $tty:" .
                     scalar localtime($time)."\n";
               next;
           }
```

[1] Число секунд, прошедших с некоторого момента времени. Например, «эпоха» в большинстве Unix-систем – это 00:00:00 по Гринвичу (GMT) 1 января 1970 года.

```
            # будем использовать предыдущую запись об открытии
            # соединения и эту запись о закрытии соединения в
            # качестве записи об одном сеансе. Чтобы сделать
            # это, мы создаем список списков, где каждый список
            # имеет вид (hostname, login, logout)
            push(@sessions,
                 [@{shift @{$connections{$tty}}},$time]);

            # если для этого терминала больше нет соединений в
            # стэке, удаляем запись из хэша
            delete $connections{$tty}
               unless (@{$connections{$tty}});
        }
    }
    close(WTMP);
    print STDERR "Готово.\n";
}
```

Давайте посмотрим, что происходит в этой программе. Мы считываем по одной записи из файла *wtmp*. Если эта запись начинается с `ftp`, мы знаем, что это сеанс FTP. Как говорится в комментарии, строка кода, в которой принимается это решение, явно привязана к формату записи в *wtmp*. Будь поле `tty` не первым полем записи, эта проверка не сработала бы. Однако возможность узнать, что строка не представляет для нас интереса, не выполняя для этого `unpack()`, того стоит.

Когда мы находим строчку, начинающуюся с `ftp`, мы разбиваем ее, чтобы выяснить, относится она к открытию FTP-соединения или к закрытию. Если это открытие соединения, то мы записываем его в `%connections`, структуру данных, хранящую сводку по открытым соединениям. Как и `%transfers` из предыдущей подпрограммы, это хэш списка списков, на этот раз его ключами являются терминалы для каждого соединения. Каждое значение этого хэша – это набор пар, представляющих имя узла, установившего соединение, и время установки соединения.

Зачем нужна такая сложная структура данных для слежения за открытием соединений? К сожалению, в *wtmp* нет простых пар строк «открытие-закрытие открытие-закрытие открытие-закрытие». Например, посмотрим на строки из *wtmp* (их выводила наша первая в этой главе программа, работающая с *wtmp*):

```
ftpd1833:dnb:ganges.ccs.neu.e:Fri Mar 27 14:04:47 1998
ttyp7:(logout):(logout):Fri Mar 27 14:05:11 1998
ftpd1833:dnb:hotdiggitydog-he:Fri Mar 27 14:05:20 1998
ftpd1833:(logout):(logout):Fri Mar 27 14:06:20 1998
ftpd1833:(logout):(logout):Fri Mar 27 14:06:43 1998
```

Обратите внимание на две записи об открытии FTP-соединения на одном и том же терминале (1-я и 3-я строчки). Если бы мы сохраняли по

одному соединению для терминала в простом хэше, то потеряли бы информацию о первом соединении, встретив второе.

Вместо этого мы в качестве стека используем список списков, ключами которого являются терминалы из %connections. Когда встречается запись об открытии соединения, пара *(host, login-time)* помещается в стек для этого терминала. Каждый раз, когда встречается информация о закрытии соединения с этого терминала, одна из записей об открытии соединения «выбрасывается» из стека и вся информация о сеансе целиком сохраняется в другой структуре данных. Для этого в программе есть такая строка:

```
push(@sessions,[@{shift @{$connections{$tty}}},$time]);
```

Давайте разберемся с этой строкой «изнутри», чтобы все прояснить. Выделенная жирным часть строки возвращает ссылку на стек/список открытых соединений для данного терминала:

```
push(@sessions,[@{shift @{$connections{$tty}}},$time]);
```

Эта часть выбрасывает из стека ссылку на первое соединение:

```
push(@sessions,[@{shift @{$connections{$tty}}},$time]);
```

Мы разыменовываем ее, чтобы получить сам список *(host, login-time)* для соединения. Если поместить эту пару в начало другого списка, заканчивающегося временем соединения, Perl интерполирует пары для соединения, и мы получим один список из трех элементов. Теперь у нас есть группа *(host, login-time, logout-time)*:

```
push(@sessions,[@{shift @{$connections{$tty}}},$time]);
```

Теперь, когда у нас есть все составляющие (узел, начало соединения и конец соединения) для сеанса FTP в одном списке, можно добавить ссылку на этот список в список @sessions, который будет использоваться позже:

```
push(@sessions,[@{shift @{$connections{$tty}}},$time]);
```

Благодаря одной очень насыщенной строке у нас есть список сеансов.

Чтобы завершить работу в подпрограмме &ScanWtmp, необходимо проверить, пуст ли стэк для каждого терминала, т. е. проверить, что не осталось больше записей об открытии соединения. Если это так, можно удалить эту запись из хэша; мы знаем, что соединение завершилось:

```
delete $connections{$tty} unless (@{$connections{$tty}});
```

Настало время поставить в соответствие два различных набора данных. Эта задача ложится на плечи подпрограммы &ShowTransfers. Для каждого сеанса она выводит список из трех элементов, относящихся к соединению, и файлы, переданные во время этого сеанса.

```
# обходим в цикле журнал соединений, ставя в соответствие
# сеансы с передачами файлов
sub ShowTransfers {
    local($session);

    foreach $session (@sessions){

       # выводим время соединения
       print scalar localtime($$session[1]) . "-" .
             scalar localtime($$session[2]) .
              " $$session[0]\n";

       # ищем все файлы, переданные в этом сеансе и выводим их
        print &FindFiles(@{$session}),"\n";
    }
}
```

Вот самая сложная часть, в которой приходится решать, передавались ли файлы в течение сеанса связи:

```
# возвращает все файлы, переданные в течение данного сеанса
sub FindFiles{
    my($rhost,$login,$logout) = @_;
    my($transfer,@found);

    # простой случай, передачи файлов не было
    unless (exists $transfers{$rhost}){
       return "\t(no transfers in xferlog)\n";
    }

    # простой случай, первая запись о передаче файлов записана
    # после регистрации
    if ($transfers{$rhost}->[0]->[0] > $logout){
       return "\t(no transfers in xferlog)\n";
    }

    # ищем файлы, переданные во время сеанса
    foreach $transfer (@{$transfers{$rhost}}){

        # если передача до регистрации
       next if ($$transfer[0] < $login);

        # если передача после регистрации
       last if ($$transfer[0] > $logout);

        # если мы уже использовали эту запись
       next unless (defined $$transfer[1]);

       push(@found,"\t".$$transfer[1]."\n");
       undef $$transfer[1];
    }
```

```
        ($#found > -1 ? @found : "\t(no transfers in xferlog)\n")
    }
```

Первым делом можно исключить простые случаи. Если мы не нашли записей о передаче файлов, выполненной этим узлом, или если первая передача произошла после завершения интересующего нас сеанса, это означает, что в течение данного сеанса файлы не передавались.

Если нельзя исключить простые случаи, необходимо просмотреть список всех передач. Мы проверяем, произошла ли передача, связанная с данным узлом, после начала сеанса, но до его завершения. Мы переходим к следующей передаче, если какое-либо из этих утверждений неверно. Кроме того, мы прекращаем проверку остальных передач для этого узла, когда находим запись о передаче, произошедшей после завершения соединения. Помните, уже говорилось о том, что все записи о передаче файлов добавляются в структуру данных в хронологическом порядке? Это окупается именно здесь.

Последняя проверка перед тем, как решить, засчитывать ли запись о передаче файла, выглядит несколько странно:

```
# если мы уже использовали эту запись
next unless (defined $$transfer[1]);
```

Если два анонимных сеанса с одного и того же узла происходят в одно и то же время, то нет никакого шанса выяснить, к какому из них относится запись о передаче этого файла. Ни в одном из журналов просто не существует информации, которая могла бы нам в этом помочь. Лучшее, что можно тут сделать, – определить правило и придерживаться его. Здесь правило такое: «Приписывать передачу первому возможному соединению». Эта проверка и последующий `undef` проводят его в жизнь.

Если последняя проверка пройдена, мы объявляем о победе и добавляем имя файла к списку файлов, переданных в течение этого сеанса. После этого выводим информацию о сеансах и выполненных передачах файлов.

Подобные программы, в которых выполняется поиск взаимосвязей, могут быть довольно сложными, особенно когда они объединяют источники данных, связи между которыми не являются достаточно четкими. Так что давайте посмотрим, можно ли подойти к этому проще.

Черные ящики

В мире Perl часто случается так, что когда вы пытаетесь написать что-то широко используемое, кто-то другой публикует свое решение этой задачи раньше. Это дает возможность просто передать свои данные в уже готовый модуль и получить результаты, не задумываясь о том, как выполняется данная задача. Это часто назвают «подходом черного ящика».

Один такой пример – это пакет SyslogScan Рольфа Харольда Нельсона (Rolf Harold Nelson). Раньше мы уже отмечали, что анализ почтового журнала *sendmail* может оказаться непростой задачей из-за информации о состоянии. Часто со строкой связана одна или несколько родственных строк, перемешанных с другими строками в этом же журнале. Пакет SyslogScan предоставляет простой способ обратиться к информации о доставке каждого сообщения, так что нет необходимости вручную просматривать файл и выбирать оттуда все связанные строки. Этот пакет позволяет найти в журнале определенные адреса и предоставляет некоторую статистику по найденным сообщениям.

Пакет SyslogScan объектно-ориентированный, так что первым делом нужно загрузить модуль и создать новый экземпляр объекта:

```
use SyslogScan::DeliveryIterator;

# список почтовых журналов syslog
$maillogs = ["/var/log/mail/maillog"];

$iterator = new SyslogScan::DeliveryIterator(syslogList => $maillogs);
```

Метод new модуля SyslogScan::DeliveryIterator возвращает *итератор* (*iterator*), т. е. указатель в файле, двигающийся от одной строки о доставке сообщения к другой. Применяя итератор, мы избавляемся от необходимости просматривать файл в поисках всех строк, относящихся к конкретному сообщению. Если вызвать метод next() для этого итератора, он вернет нас обратно к объекту доставки. Этот объект хранит информацию о доставке, прежде распределенную по нескольким строкам в журнале. Например, следующий код:

```
while ($delivery = $iterator -> next()){
    print $delivery->{Sender}." -> ".
           join(",",@{$delivery->{ReceiverList}}),"\n";
}
```

позволяет получить такую информацию:

```
root@host.ccs.neu.edu -> user1@cse.scu.edu
owner-freebsd-java-digest@freebsd.org -> user2@ccs.neu.edu
root@host.ccs.neu.edu -> user3@ccs.neu.edu
```

Можно сделать еще лучше. Если передать итератор из SyslogScan методу new модуля SyslogScan::Summary, new примет весь вывод метода next итератора и вернет итоговый объект. Этот объект содержит итоговую информацию по всем доставкам сообщений, которые только может вернуть итератор.

Но SyslogScan переносит эту функциональность на другой уровень. Если передать последний объект методу new из SyslogScan::ByGroup, мы получим объект bygroup, в котором вся информация сгруппирована по

доменам и приводится статистика по этим группам. Вот как применяется то, о чем мы только что говорили:

```
use SyslogScan::DeliveryIterator;
use SyslogScan::Summary;
use SyslogScan::ByGroup;
use SyslogScan::Usage;

# местоположение maillog
$maillogs = ["/var/log/mail/maillog"];

# получаем для этого файла итератор
$iterator = new SyslogScan::DeliveryIterator(syslogList => $maillogs);

# передаем итератор в ::Summary, получаем объект summary (сводка)
$summary = new SyslogScan::Summary($iterator);

# передаем сводку в ::ByGroup и получаем объект stats-by-group
# (статистика по группам)
$bygroup = new SyslogScan::ByGroup($summary);

# выводим содержимое этого объекта
foreach $group (sort keys %$bygroup){
    ($bmesg,$bbytes)=@{$bygroup->{$group}->
                       {groupUsage}->getBroadcastVolume()};
    ($smesg,$sbytes)=@{$bygroup->{$group}->
                       {groupUsage}->getSendVolume()};
    ($rmesg,$rbytes)=@{$bygroup->{$group}->
                       {groupUsage}->getReceiveVolume()};
    ($rmesg,$rbytes)=@{$bygroup->{$group}->
                       {groupUsage}->getReceiveVolume()};
    write;
}

format STDOUT_TOP =
Name                          Bmesg  BByytes  Smesg  SBytes   Rmesg  Rbytes
---------------------------  -----  -------- ------ -------- ------ -------
.
format STDOUT =
@<<<<<<<<<<<<<<<<<  @>>>>> @>>>>>>> @>>>>> @>>>>>>> @>>>>> @>>>>>>>
$group,$bmesg,$bbytes,$smesg,$sbytes,$rmesg,$rbytes
.
```

Результат представляет собой подробный отчет по количеству широковещательных, отправленных и полученных сообщений и их размера в байтах. Вот отрывок из получаемых результатов:

```
Name                 Bmesg  BByytes  Smesg  SBytes   Rmesg  Rbytes
-------------------  -----  -------- ------ -------- ------ -------
getreminded.com          1     3420      1     3420      0        0
gillette.com             1      984      1      984      4     7812
```

```
gis.net              3    10830     3    10830     1      787
globalserve.net      1     1245     1     1245     0        0
globe.com            0        0     0        0     1     2040
```

Положительная сторона такого подхода в том, что можно сделать многое благодаря тяжелой работе, проделанной автором модуля или сценария, не прикладывая больших усилий со своей стороны. Отрицательная сторона – необходимость во всем полагаться на код автора. В нем могут быть ошибки или может использоваться подход, не устраивающий вас. Всегда стоит сначала ознакомиться с программой, прежде чем «дать ей зеленую улицу» у себя на сайте.

Использование баз данных

Последний подход, который мы обсудим, для своей реализации требует других знаний помимо Perl. Так что мы просто рассмотрим технологию, которая со временем, вероятно, станет более популярной.

Все рассмотренные предыдущие примеры хорошо работают с данными приемлемого размера и на машинах с приемлемым количеством памяти, но они не масштабируемы. В ситуациях, когда у вас много данных, особенно если они поступают из различных источников, естественным инструментом становятся базы данных.

Существует по крайней мере два способа использования баз данных из Perl. Первый из них я называю методом «только Perl». В этом случае все действия осуществляются в Perl или в библиотеках, тесно связанных с Perl. Во втором применяются модули, например из семейства DBI, позволяющие сделать Perl клиентом баз данных, таких как MySQL, Oracle или MS-SQL. Рассмотрим оба подхода для обработки и анализа журналов.

Использование баз данных, встроенных вPerl

До тех пор пока данных не слишком много, можно применять только Perl. В качестве примера мы будем использовать расширенную версию вездесущего «искателя брешей в системе безопасности». До сих пор наша программа имела дело с соединениями только на одной машине. Как поступить, если захочется узнать о регистрации злоумышленников и на других наших машинах?

Первый шаг – поместить все данные из *wtmp* для наших машин в ту или иную базу данных. Будем считать, что все машины имеют прямой доступ к некоторым разделяемым каталогам через некую сетевую файловую систему наподобие NFS. Перед тем как двигаться дальше, необходимо выбрать формат базы данных.

В качестве «формата баз данных для Perl» я выбрал формат Berkeley DB. Я беру «формат баз данных для Perl» в кавычки потому, что хоть поддержка DB встроена в Perl, сами библиотеки DB необходимо дос-

тать в другом месте (*http://www.sleepycat.com*) и установить их до того, как поддержка Perl будет скомпилирована. Ниже приведено сравнение между различными поддерживаемыми форматами баз данных (табл. 9.4).

Таблица 9.4. Сравнение поддерживаемых в Perl форматов баз данных

Название	Поддержка в Unix	Поддержка в NT/2000	Поддержка в MacOS	Ограничения на размеры ключей или значений	Независимость от порядка байтов
«старый» dbm	Да	Нет	Нет	1K	Нет
«новый» dbm	Да	Нет	Да	4K	Нет
Sdbm	Да	Да	Нет	1K (по умолчанию)	Нет
Gdbm	Да[a]	Да[b]	Нет	Нет	Нет
DB	Да[1]	Да[1]	Да	Нет	Да

[a] Может потребоваться отдельно загрузить сами библиотеки баз данных.

[b] Библиотеку и модуль необходимо загрузить из сети (*http://www.roth.net*).

Мне нравится формат Berkeley-DB, поскольку он может обрабатывать большие объемы данных и не зависит от порядка байт. Независимость от порядка байт особенно важна для программы, которую мы собираемся рассмотреть, т. к. мы будем считывать и записывать данные в один и тот же файл с различных машин, у которых может быть различная архитектура.

Начнем с заполнения базы данных. В целях простоты и переносимости мы остановим свой выбор на программе *last*, чтобы не использовать `unpack()` для различных файлов *wtmp*. Вот программа, за которой следуют объяснения:

```
use DB_File;
use FreezeThaw qw(freeze thaw);
use Sys::Hostname; # чтобы получить текущее имя узла
use Fcntl;         # для определения O_CREAT и O_RDWR

# ищем исполняемый файл для программы last
(-x "/bin/last" and $lastex = "/bin/last") or
  (-x "/usr/ucb/last" and $lastex = "/usr/ucb/last");

$userdb    = "userdata";     # файл базы данных пользователей
$connectdb = "connectdata";  # файл базы данных соединений
$thishost  = &hostname;

open(LAST,"$lastex|") or
    die "Невозможно запустить программу $lastex:$!\n";
```

```
# считываем каждую строку вывода last
while (<LAST>){
    next if /^reboot\s/ or /^shutdown\s/ or
           /^ftp\s/     or /^wtmp\s/;
    ($user,$tty,$host,$day,$mon,$date,$time) = split;
    next if $tty =~ /^:0/ or $tty =~ /^console$/;
    next if (length($host) < 4);
    $when = $mon." ".$date." ".$time;

   # сохраняем каждую запись в хэше списка списков
    push(@{$users{$user}},[$thishost,$host,$when]);
    push(@{$connects{$host}},[$thishost,$user,$when]);
}

close(LAST);

# создаем файл базы данных (для чтения и записи); если он не
# существует, смотрите сноску в тексте re: $DB_BTREE
tie %userdb, "DB_File",$userdb,O_CREAT|O_RDWR, 0600, $DB_BTREE
  or die "Невозможно открыть базу данных $userdb для чтения/записи:$!\n";

# обходим в цикле пользователей и сохраняем информацию в базе
# данных при помощи freeze
foreach $user (keys %users){
    if (exists $userdb{$user}){
       ($userinfo) = thaw($userdb{$user});
       push(@{$userinfo},@{$users{$user}});
       $userdb{$user}=freeze $userinfo;
    }
    else {
       $userdb{$user}=freeze $users{$user};
    }
}
untie %userdb;

# делаем то же самое для соединений
tie %connectdb, "DB_File",$connectdb,O_CREAT|O_RDWR,
                0600, $DB_BTREE
  or die "Невозможно открыть базу данных $connectdb для чтения/записи:$!\n";
foreach $connect (keys %connects){
    if (exists $connectdb{$connect}){
       ($connectinfo) = thaw($connectdb{$connect});
       push(@{$connectinfo},@{$connects{$connect}});
       $connectdb{$connect}=freeze($connectinfo);
    }
    else {
       $connectdb{$connect}=freeze($connects{$connect});
    }
}
untie %connectdb;
```

Программа принимает вывод команды *last* и делает следующее:

1. Отфильтровывает бесполезные строки.
2. Сохраняет вывод в двух хэшах списка списков, структура данных которых выглядит так:

```
$users{username} =
    [[current host, connecting host, connect time],
     [current host, connecting host, connect time]
     ...
    ];
$connects{host} =
    [[current host, username1, connect time],
     [current host, username2, connect time],
     ...
    ];
```

3. Помещает структуру данных в память и пытается добавить ее в базу данных.

Этот последний шаг самый интересный, поэтому рассмотрим его подробно. Мы связываем хэши `%userdb` и `%connectdb` с файлами баз данных.[1] Это позволяет легко обращаться к хэшам, в то время как Perl «за сценой» обрабатывает сохранение и получение данных из файлов базы данных. Но в хэшах хранятся только простые строки. Как преобразовать наш «хэш списка списков» в одно значение?

Модуль `FreezeThaw` Ильи Захаревича используется для хранения сложной структуры данных в одном скалярном значении, которое можно применять в качестве значения хэша. `FreezeThaw` принимает произвольную структуру данных Perl и представляет ее в виде строки. Существуют и другие модули, подобные этому, самые распространенные из которых `Data::Dumper` Гурусами Сарати (Gurusamy Sarathy) (входит в состав Perl) и `Storable` Рафаэля Манфреди (Raphael Manfredi). `FreezeThaw` обеспечивает наиболее компактное представление сложной структуры данных, поэтому он и используется здесь. Каждый из этих модулей имеет свои плюсы, так что внимательно изучите возможности всех трех, если вам нужно будет решать задачу, подобную нашей.

В программе мы проверяем, существует ли запись для этого пользователя или узла. Если нет, мы просто «замораживаем» структуру данных в строку и сохраняем эту строку в базе данных при помощи связанного хэша. Если существует, мы «размораживаем» существующую

[1] Обычно в случае применения `DB_File` нет необходимости использовать форму хранения BTREE, но в этой программе, возможно, понадобится хранить очень длинные значения. Такие значения приводят к повреждению данных, если применяется метод хранения `DB_HASH` из версии 1.85, тогда как метод хранения BTREE, похоже, справляется с дроблением. В последних версиях библиотек DB может и не быть этой ошибки.

структуру данных из базы данных в память, добавляем наши данные, вновь ее «замораживаем» и сохраняем.

Выполнив эту программу на нескольких машинах, мы получим базу данных с некоторой потенциально полезной информацией, которую можно будет добавить в следующую версию нашей программы.

Подходящее время для заполнения подобной базы данных – сразу после операции ротации журналов *wtmp*.

Код, используемый здесь для заполнения базы данных, настолько прост (это скорее план, чем реальная программа), что его не стоит широко применять в жизни. Один недостаток, который бросается в глаза, это отсутствие механизма, предотвращающего попытки одновременного обновления базы данных несколькими экземплярами программы. Учитывая, что блокировка файлов через NFS по крайней мере неочевидна, было бы проще вызывать подобную программу из большей программы, которая возьмет на себя сбор информации от каждой машины.

Теперь, заполнив базу данных, рассмотрим улучшенную версию программы, использующей эту информацию:

```
use DB_File;
use FreezeThaw qw(freeze thaw);
use Fcntl;

# принимаем имя пользователя и узлы, которые мы игнорируем, в
# командной строке
($user,$ignore) = @ARGV;

# файлы баз данных, которые мы используем
$userdb    ="userdata";
$connectdb ="connectdata";

tie %userdb, "DB_File",$userdb,O_RDONLY,666,$DB_BTREE
  or die "Невозможно открыть базу данных $userdb для чтения:$!\n";
tie %connectdb, "DB_File",$connectdb,O_RDONLY,666,$DB_BTREE
  or die "Невозможно открыть базу данных $connectdb для чтения:$!\n";
```

Мы загрузили нужные нам модули, получили необходимые данные, установили несколько переменных и связали их с файлами базы данных. Теперь пришло время немного поработать:

```
# можно выходить, если этот пользователь не устанавливал
# соединений
unless (exists $userdb{$user}){
    print "Этот пользователь не регистрировался..\n";
    untie %userdb;
    untie %connectdb;
    exit;
```

```
}

($userinfo) = thaw($userdb{$user});

print "-- first host contacts from $user --\n";
foreach $contact (@{$userinfo}){
    next if (defined $ignore and $contact->[1] =~ /$ignore/o);
    print $contact->[1] . " -> " . $contact->[0] .
          " on ".$contact->[2]."\n";
    $otherhosts{$contact->[1]}='';
}
```

Вот как работает этот код: если мы видели этого пользователя, то воспроизводим в памяти записи о его соединениях при помощи `thaw()`. Для каждого контакта мы проверяем, нужно ли игнорировать соединения с этого узла. Если нет, то выводим информацию об этом соединении и записываем узел, с которого оно было установлено, в хэш `%otherhosts`.

Здесь хэш применяется как простой способ собрать список уникальных узлов из всех записей о соединениях. Теперь, когда у нас есть список узлов, с которых мог зарегистрироваться злоумышленник, необходимо выяснить, какие еще пользователи регистрировались с этих подозрительных узлов.

Найти эту информацию будет не сложно, потому что когда мы записывали, какие пользователи регистрировались на каких машинах, мы также записывали и обратное (т. е. на каких машинах регистрировались какие пользователи) в другом файле базы данных. Теперь мы смотрим на записи, соответствующие найденным на предыдущем шаге узлам. Если этот узел не надо игнорировать и с него было зарегистрировано соединение, мы собираем список пользователей, регистрировавшихся на этом узле, при помощи хэша `%userseen`:

```
print "-- other connects from source machines  --\n";
foreach $host (keys %otherhosts){
    next if (defined $ignore and $host =~ /$ignore/o);
    next unless (exists $connectdb{$host});

   ($connectinfo) = thaw($connectdb{$host});

   foreach $connect (@{$connectinfo}){
       next if (defined $ignore and $connect->[0] =~ /$ignore/o);
       $userseen{$connect->[1]}='';
   }
}
```

Последнее действие этой драмы в трех актах имеет элегантный конец. Мы возвращаемся к первоначальной базе данных пользователей, чтобы найти все соединения, установленные подозрительными пользователями с подозрительных машин:

```
foreach $user (sort keys %userseen){
```

```
        next unless (exists $userdb{$user});

        ($userinfo) = thaw($userdb{$user});

        foreach $contact (@{$userinfo}){
            next if (defined $ignore and $contact->[1] =~ /$ignore/o);
            write if (exists $otherhosts{$contact->[1]});
        }
    }
```

Нам осталось только подмести сцену и уйти домой:

```
untie %userdb;
untie %connectdb;

format STDOUT =
@<<<<<<<< @<<<<<<<<<<<<<<<< -> @<<<<<<<<<<<<<<<< on @<<<<<<<<<<<
$user.":",$contact->[1],$contact->[0],$contact->[2]
.
```

Вот как выглядит вывод этой программы (опять же, имена пользователей и машин изменены):

```
-- first host contacts from baduser --
badhost1.exampl -> machine1.ccs.neu.edu on Jan 18 09:55
badhost2.exampl -> machine2.ccs.neu.edu on Jan 19 11:53
-- other connects from source machines  --
baduser2:  badhost1.exampl -> machine2.ccs.neu.e on Dec 15 13:26
baduser2:  badhost2.exampl -> machine2.ccs.neu.e on Dec 11 12:45
baduser3:  badhost1.exampl -> machine1.ccs.neu.ed on Jul 13 16:20
baduser4:  badhost1.exampl -> machine1.ccs.neu.ed on Jun 9 11:53
baduser:   badhost1.exampl -> machine1.ccs.neu.ed on Jan 18 09:55
baduser:   badhost2.exampl -> machine2.ccs.neu.e on Jan 19 11:53
```

Эта программа хороша в качестве примера, но она не масштабируется дальше, чем на небольшую группу машин. Для каждого последующего вызова программы необходимо прочитать запись из базы данных, «растопить» (`thaw()`) ее в памяти, добавить новые данные, снова их «заморозить» (`freeze()`) и сохранить опять в базе данных. Это может потребовать больших затрат процессорного времени и памяти. Потенциально весь процесс происходит для каждого пользователя и соединения, так что все замедляется очень быстро.

Использование баз данных SQL

Теперь рассмотрим один из способов обращения с очень большими наборами данных. Вам может потребоваться загрузить данные в более сложную базу данных SQL (коммерческую или нет) и запрашивать из нее информацию, используя SQL. Тем, кто не знаком с SQL, я рекомендую ознакомиться с приложением D «Пятнадцатиминутное руководство по SQL», перед тем как смотреть на этот пример.

Заполнить базу данных можно так:

```
use DBI;
use Sys::Hostname;

$db = "dnb"; # имя используемой базы данных

# ищем местоположение last
(-x "/bin/last" and $lastex = "/bin/last") or
  (-x "/usr/ucb/last" and $lastex = "/usr/ucb/last");

# подсоединяемся к базе данных Sybase как пользователь "dnb",
# указывая пароль в командной строке
$dbh = DBI->connect('dbi:Sybase:','dnb',$ARGV[0]);
die "Невозможно соединиться: $DBI::errstr\n"
  unless (defined $dbh);

# переходим на базу данных, которую мы будем использовать
$dbh->do("use $db") or
  die "Невозможно перейти к $db: ".$dbh->errstr."\n";

# создаем таблицу lastinfo, если ее еще не существует
unless ($dbh->selectrow_array(
          q{SELECT name from sysobjects WHERE name="lastinfo"})){
          $dbh->do(q{create table lastinfo (username char(8),
                                            localhost char(40),
                                            otherhost varchar(75),
                                            when char(18))}) or
      die "Невозможно создать таблицу lastinfo: ".$dbh->errstr."\n";
}

$thishost  = &hostname;

$sth = $dbh->prepare(
  qq{INSERT INTO lastinfo(username,localhost,otherhost,when)
   VALUES (?,'$thishost', ?, ?)}) or
  die "Невозможно подготовить запрос insert: ".$dbh->errstr."\n";

open(LAST,"$lastex|") or die "Невозможно выполнить программу $lastex:$!\n";

while (<LAST>){
    next if /^reboot\s/ or /^shutdown\s/ or
            /^ftp\s/    or /^wtmp\s/;
    ($user,$tty,$host,$day,$mon,$date,$time) = split;
    next if $tty =~ /^:0/ or $tty =~ /^console$/;
    next if (length($host) < 4);
    $when = $mon." ".$date." ".$time;

    $sth->execute($user,$host,$when);
}
```

```
close(LAST);
$dbh->disconnect;
```

Эта программа создает таблицу *lastinfo* с полями *username*, *localhost*, *otherhost* и *when*. Мы обходим в цикле вывод команды *last*, добавляя настоящие записи в таблицу.

Теперь можно использовать базу данных по назначению. Вот набор простеньких SQL-запросов, которые легко можно выполнить из Perl при помощи интерфейсов DBI или ODBC, о которых мы говорили в главе 7 «Администрирование баз данных SQL»:

```
-- сколько всего записей в таблице?
select count (*) from lastinfo;

 -----------
       10068
-- сколько пользователей было зарегистрировано?
select count (distinct username) from lastinfo;

 -----------
         237

-- сколько различных узлов устанавливали соединение с нашими машинами?
select count (distinct otherhost) from lastinfo;

 -----------
        1000

-- на каких локальных машинах регистрировался пользователь "dnb"?
select distinct localhost from lastinfo where username = "dnb";
 localhost
 ----------------------------------------
 host1
 host2
```

Эти примеры должны помочь читателю прочувствовать, как можно «исследовать» данные, когда они хранятся в настоящей базе данных. Каждый из этих запросов требует для выполнения лишь около секунды. Базы данных могут быть быстрым, мощным инструментом для системного администрирования.

Анализ журналов – бесконечная тема для разговора. К счастью, эта глава снабдила вас кое-какими инструментами и некоторым вдохновением.

Информация о модулях из этой главы

Модуль	Идентификатор на CPAN	Версия
`Win32::EventLog` (распространяется с ActivePerl)		0.062
`Logfile::Rotate`	PAULG	1.03
`Getopt::Long` (распространяется с Perl)		2.20
`Time::Local` (распространяется с Perl)		1.01
`SyslogScan`	RHNELSON	0.32
`DB_File` (распространяется с Perl)	PMQS	1.72
`FreezeThaw`	ILYAZ	0.3
`Sys::Hostname` (распространяется с Perl)		
`Fcntl` (распространяется с Perl)		1.03
`DBI`	TIMB	1.13

Рекомендуемая дополнительная информация

«*Essential System Administration*», (2nd Edition), Eleen Frisch (O'Reilly, 1995). А книге есть хорошее, краткое введение в *syslog*.

http://www.heysoft.de/index.htm – домашняя страница Франка Хэйне (Frank Heyne) – человека, предоставляющего программное обеспечение для анализа журнала событий в Win32. Также здесь есть хороший список часто задаваемых вопросов по Event Log.

http://www.le-berre.com/ – домашняя страница Филиппа Ле Бера (Philippe Le Berre); содержит отличный отчет по использованию `Win32::EventLog` и других пакетов для Win32.

«*Managing NT Event Logs with Perl for Win32*», Bob Wells, *Windows NT Magazine*, February/March 1998.

«*Practical Unix & Internet Security*», (2nd Edition), Simson Garfinkel, Gene Spafford (O'Reilly, 1996). Еще одно хорошее (и несколько более подробное) введение в *syslog*, также содержит информацию по *tcpwrappers*.

«*Windows NT Event Logging*», James D. Murray (O'Reilly, 1998).

- *Обращаем внимание на неожиданные или несанкционированные изменения*
- *Обращайте внимание на подозрительную активность*
- *Протокол SNMP*
- *Опасность на проводе*
- *Предотвращение*
- *подозрительных действий*
- *Информация о модулях из этой главы*
- *Рекомендуемая дополнительная литература*

10

Безопасность и наблюдение за сетью

Любой разговор о *безопасности* сопряжен с риском. Существует по крайней мере три ловушки, которые могут обречь на неудачу разговор о безопасности:

1. Под словом «безопасность» для разных людей скрываются различные вещи. Если вы придете на конференцию ученых, изучающих Древнюю Грецию и Рим, и спросите их о Риме, первый вдохновенно прочтет вам лекцию об акведуках (инфраструктура и снабжение), второй расскажет о Римской империи (идеология и политика), третий растолкует о римских легионах (армия), четвертый поведает о Сенате (администрация) и т. д. Необходимость иметь дело сразу с каждой гранью безопасности – это первая ловушка.
2. Люди думают, что что-то может быть безопасным, будь то программа, компьютер, сеть или что-либо еще. Мы не претендуем на то, чтобы показать, как сделать что-то безопасным; но попробуем помочь вам сделать что-то более безопасным или уж, в крайнем случае, поможем различать, когда что-нибудь является менее безопасным. Безопасность это континуум.
3. Наконец, одна из самых серьезных ловушек в этом деле – специфичность. Верно, что сущность безопасности часто заключается в деталях, но это постоянно меняющийся набор деталей. Факт, что вы залатали дыры A, B и C на своей системе, гарантирует только то

(да и то не всегда), что именно эти дыры больше не будут источником проблем. Это никак не поможет, если будет найдена дыра D. Вот почему эта глава рассказывает о принципах и инструментах для увеличения безопасности. Читатель не найдет здесь ничего о том, как исправить конкретную неприятность, например, переполнение буфера, ненадежный ключ реестра или доступный всем для записи системный файл.

Один из хороших способов ввязаться в дискуссию об этих принципах – выяснить, как безопасность проявляется в физическом мире. Как в реальном, так и в виртуальном мире все сводится к *опасениям*. Будет ли то, чем я дорожу, повреждено, потеряно или обнаружено? Могу ли я сделать что-то, чтобы предотвратить это? Происходит ли это *прямо сейчас*?

Если разобраться, как обходятся с этими опасениями в реальном мире, то можно уяснить, как справляться с ними и в нашей области. Один из способов разобраться с такими опасениями – придумать более надежный способ оградить пространство. В случае с физическим пространством мы используем конструкции, подобные банковским сейфам; если речь идет об интеллектуальном пространстве, мы применяем методы сокрытия данных, например, гриф «совершенно секретно» или шифрование. Но это бесконечная игра в догонялки. На каждый час, потраченный на проектирование системы безопасности, приходится как минимум один час, потраченный на поиск способа обойти ее. В нашем случае есть еще полчища скучающих подростков с компьютерами и раздраженных уволенных служащих, которым просто не терпится применить где-то излишки энергии.

Более правильный подход к повышению безопасности, выдержавший испытание временем, заключается в том, чтобы прибегнуть к услугам специального человека, уменьшающего количество опасений. Когда-то давным-давно не было ничего более утешительного, чем звук шагов ночного сторожа, обходящего город и проверяющего, что все дома заперты и находятся в безопасности. Мы будем использовать этот не совсем обычный образ в качестве отправной точки наших исследований безопасности и наблюдения за сетью с помощью Perl.

Обращаем внимание на неожиданные или несанкционированные изменения

Хороший сторож замечает перемены. Он знает, когда что-то оказывается не на месте в вашем окружении. Если ценного мальтийского сокола заменят подделкой, сторож будет первым, кто должен это заметить. Точно так же пользователь хочет услышать рев сирены, если кто-то изменит или заменит основные файлы в системе. В большинстве случаев эти изменения будут безвредными. Но когда кто-то впервые действительно нарушит безопасность вашей системы и начнет делать что-то с

файлами */bin/login*, *msgina.dll* или *Finder*, то вы, заметив это, будете настолько счастливы, что простите все предыдущие ложные тревоги.

Изменения локальной файловой системы

Файловые системы – это отличное место для начала исследований программ, следящих за изменениями. Мы собираемся исследовать способы проверки неизменности важных файлов, в частности, исполняемых файлов операционной системы или файлов, связанных с безопасностью (например */etc/passwd* или *msgina.dll*). Изменения, внесенные в эти файлы без ведома администратора, часто являются признаками вмешательства злоумышленника. В сети существует ряд довольно сложных инструментов, которые устанавливают троянские версии важных файлов и заметают следы. Это самые злобные изменения, которые можно обнаружить. С другой стороны, иногда просто полезно знать, что важные файлы изменились (особенно если одну и ту же систему администрируют несколько человек). Технологии, которые мы рассмотрим, будут одинаково хорошо работать в обоих случаях.

Самый простой способ выяснить, был ли файл изменен – использовать функции `stat()` и `lstat()`. Эти функции принимают имя файла или файловый дескриптор и возвращают массив с информацией об этом файле. Единственное различие между этими двумя функциями проявляется в операционных системах, подобных Unix, поддерживающих символические ссылки. В таких случаях `lstat()` применяется для получения информации о файле, на который указывает ссылка, а не о самой символической ссылке. На всех остальных операционных системах информация, возвращаемая функцией `lstat()`, будет совпадать с информацией, возвращаемой функцией `stat()`.

Использовать `stat()` или `lstat()` очень просто:

```
@information = stat("filename");
```

Как сказано в главе 3 «Учетные записи пользователей», можно также применять модуль `File::Stat` Тома Кристиансена, чтобы получить эту же информацию, используя объектно-ориентированный синтаксис.

Информация, возвращаемая функциями `stat()` и `lstat()`, зависит от операционной системы. `stat()` и `lstat()` происходят от системных вызовов в Unix, поэтому Perl-документация по этим функциям ссылается на значения, возвращаемые в Unix. Можно посмотреть (табл. 10.1), как эти значения соотносятся с тем, что возвращается функцией `stat()` в Windows NT/2000 и MacOS. В первых двух столбцах приведены порядковый номер поля и его описание для систем Unix.

Таблица 10.1. Сравнение значений, возвращаемых функцией stat( )

№	Описание поля в Unix	Действительно в NT/2000	Действительно в MacOS
0	Номер устройства файловой системы	Да (порядковый номер диска)	Да (но является vRefNum)
1	Inode	Нет (всегда 0)	Да (но fileID/dirID)
2	Режим файла (тип и права)	Да	Да (но 777 для каталогов и приложений, 666 для незаблокированных документов, 444 для заблокированных документов)
3	Количество (жестких) ссылок на файл	Да (для NTFS)	Нет (всегда 1)
4	Численный идентификатор владельца файла	Нет (всегда 0)	Нет (всегда 0)
5	Численный идентификатор группы владельца файла	Нет (всегда 0)	Нет (всегда 0)
6	Идентификатор устройства (только для специальных файлов)	Да (порядковый номер диска)	Нет (всегда null)
7	Размер файла в байтах	Да (но не включает размер каких-либо альтернативных потоков данных)	Да (но возвращает только размер данных)
8	Время последнего доступа относительно начала эпохи	Да	Да (только эпоха начинается на 66 лет раньше, чем в Unix, то есть 1/1/1904, и значение то же, что и для поля №9) [a]
9	Время последней модификации относительно начала эпохи	Да	Да (только эпоха начинается 1/1/1904 и значение то же, что и для поля №8)
10	Время последнего изменения inode относительно начала эпохи	Да (но время создания файла)	Да (только эпоха начинается 1/1/1904, и это время создания файла)
11	Предпочтительный размер блока для ввода/вывода	Нет (всегда null)	Да
12	Количество занятых блоков	Нет (всегда null)	Да

[a] Кроме того, эпоха в MacOS отсчитывается относительно *локального* времени, а не UTC. Так что если системные часы двух компьютеров с MacOS синхронизированы, но на одном из них используется временная зона –0800, а на другом –0500, то значения, возвращаемые функцией `time()` на этих компьютерах, будут отличаться на три часа.

Для возвращения атрибутов, специфичных для операционной системы, в других не-Unix-версиях Perl помимо `stat()` и `lstat()` используются специальные функции. Рассказ о таких функциях, как `MacPerl::GetFileInfo()` и `Win32::FileSecurity::Get()`, можно найти в главе 2 «Файловые системы».

После того как с помощью `stat()` для файла будут получены значения, на следующем шаге надо будет сравнить «интересные» значения с уже известными. Если они изменились, значит, изменилось и что-то в этом файле. Ниже приведена программа, которая генерирует строку значений `lstat()` и проверяет для файлов некоторые из этих значений. Мы намеренно исключили 8-е поле (время последнего доступа), потому что оно меняется при каждом прочтении файла.

Программа принимает либо аргумент *–p filename*, чтобы вывести значения `lstat()` для заданного файла, либо аргумент *–c filename*, чтобы проверить значения `lstat()` для всех файлов, перечисленных в *filename*.

```
use Getopt::Std;

# используем это для создания более симпатичного вывода
# позже в &printchanged()
@statnames = qw(dev ino mode nlink uid gid rdev
                size mtime ctime blksize blocks);

getopt('p:c:');

die "Использование: $0 [-p <filename>|-c <filename>]\n"
  unless ($opt_p or $opt_c);

if ($opt_p){
    die "Невозможно получить информацию о файле $opt_p:$!\n"
      unless (-e $opt_p);
    print $opt_p,"|",join('|',(lstat($opt_p))[0..7,9..12]),"\n";
    exit;
}

if ($opt_c){
    open(CFILE,$opt_c) or
      die "Невозможно открыть файл $opt_c:$!\n";
    while(<CFILE>){
        chomp;
        @savedstats = split('\|');
        die "Неверное количество полей в строке, начинающейся с
            $savedstats[0]\n"
          unless ($#savedstats == 12);
        @currentstats = (lstat($savedstats[0]))[0..7,9..12];

        # выводим измененные поля, только если что-то изменилось
        &printchanged(\@savedstats,\@currentstats)
          if ("@savedstats[1..13]" ne "@currentstats");
```

```
    }
    close(CFILE);
}

# обходим в цикле списки атрибутов и выводим все изменения
sub printchanged{
    my($saved,$current)= @_;

    # выводим имя файла после того, как выбрасываем его из
    # массива, прочитанного из файла

    print shift @{$saved},":\n";

    for (my $i=0; $i < $#{$saved};$i++){
         if ($saved->[$i] ne $current->[$i]){
             print "\t".$statnames[$i]." is now ".$current->[$i];
             print " (should be ".$saved->[$i].")\n";
         }
    }
}
```

Для использования этой программы можно набрать *checkfile –p /etc/passwd >> checksumfile*. В файле *checksumfile* теперь будет храниться строка, которая выглядит так:

```
/etc/passwd|1792|11427|33060|1|0|0|24959|607|921016509|921016509|8192|2
```

Этот шаг нужно повторить для каждого файла, за которым мы наблюдаем. Затем вызов сценария с аргументом *checkfile –c checksumfile* будет сообщать обо всех изменениях. Например, если я удалю один символ из */etc/passwd*, сценарию это не понравится, и он выведет такое сообщение:

```
/etc/passwd:
        size is now 606 (should be 607)
        mtime is now 921020731 (should be 921016509)
        ctime is now 921020731 (should be 921016509)
```

Перед тем как двигаться дальше, необходимо сказать об одном приеме, который мы применили в программе. В следующей строке проверяется равенство двух списков (сделано это на скорую руку):

```
if ("@savedstats[1..12]" ne "@currentstats");
```

Perl автоматически преобразовывает список в строку, склеивая элементы списка через пробел:

```
join(" ",@savedstats[1..12]))
```

и затем уже сравнивает получившиеся строки. Этот прием хорошо работает для коротких списков, в которых имеет значение порядок и количество элементов. В большинстве других случаев необходимо ис-

пользовать итеративный подход или хэши, как описано в списках часто задаваемых вопросов perlfaq, входящих в состав Perl.[1]

Теперь, когда вы выяснили атрибуты файлов, я вынужден вас огорчить. Проверка того, что атрибуты файлов не изменились, – это хорошая идея, но не больше. Не представляет большого труда изменить файл, оставив неизменными такие атрибуты, как время доступа и модификации. В Perl даже есть функция `utime()`, предназначенная для изменения времени доступа и модификации. Так что пришло время применить более мощные инструменты.

Обнаружение изменений в данных – это одна из сильных сторон алгоритмов, известных как криптографические хэш-функции («message-digest algorithms»). Вот как Рон Райвест (Ron Rivest) описывает алгоритм «RSA Data Security, Inc. MD5 Message-Digest Algorithm» в RFC1321:

> Алгоритм на вводе принимает сообщение произвольной длины и создает подпись (message digest или fingerprint) длиной 128 бит. Считается, что просто невозможно создать два сообщения, у которых совпадали бы подписи; также невозможно создать сообщение, подпись которого совпадала бы с заранее заданной.

Для нас это означает, что если применить к файлу алгоритм MD5, то он будет снабжен уникальной подписью. Если данные из этого файла изменятся, то независимо от того, насколько они незначительны, подпись файла тоже будет изменена. Самый простой способ воспользоваться этой чудесной возможностью из Perl – применить модуль `Digest::MD5` из семейства модулей `Digest`.

Использовать модуль `Digest::MD5` просто. Нужно создать объект `Digest::MD5`, добавить в него данные при помощи методов `add()` или `addfile()`, а затем попросить модуль создать подпись.

Можно сделать нечто подобное для подсчета подписи MD5 для файла паролей в Unix:

```
use Digest::MD5 qw(md5);

$md5 = new Digest::MD5;

open(PASSWD,"/etc/passwd") or die "Невозможно открыть passwd:$!\n";
$md5->addfile(PASSWD);
close(PASSWD);

print $md5->hexdigest."\n";
```

В документации по `Digest::MD5` сказано, что для создания более компактных программ можно связывать несколько методов вместе. Так, предыдущую программу можно переписать:

[1] Файлы perlfaq1.pod, perlfaq2.pod ... perlfaq[N].pod. – *Примеч. науч. ред.*

```
use Digest::MD5 qw(md5);

open(PASSWD,"/etc/passwd") or die "Невозможно открыть passwd:$!\n";
print Digest::MD5->new->addfile(PASSWD)->hexdigest,"\n";
close(PASSWD);
```

Обе программы выводят следующее:

```
a6f905e6b45a65a7e03d0809448b501c
```

Если в файл внести незначительные изменения, то вывод станет другим. Вот что получилось, когда я поменял местами всего *два символа* в файле паролей:

```
335679c4c97a3815230a4331a06df3e7
```

Теперь любые изменения становятся очевидными. Давайте расширим предыдущую программу проверки атрибутов и добавим к ней MD5:

```
use Getopt::Std;
use Digest::MD5 qw(md5);

@statnames =
 qw(dev ino mode nlink uid gid rdev size mtime ctime blksize blocks md5);

getopt('p:c:');

die "Использование: $0 [-p <filename>|-c <filename>]\n"
  unless ($opt_p or $opt_c);

if ($opt_p){
    die "Невозможно получить информацию о файле $opt_p:$!\n"
      unless (-e $opt_p);

    open(F,$opt_p) or die "Невозможно открыть $opt_p:$!\n";
    $digest = Digest::MD5->new->addfile(F)->hexdigest;
    close(F);

    print $opt_p,"|",join('|',(lstat($opt_p))[0..7,9..12]),
          "|$digest","\n";
    exit;
}

if ($opt_c){
    open(CFILE,$opt_c) or
      die "Невозможно открыть файл $opt_c:$!\n";

    while (<CFILE>){
        chomp;
        @savedstats = split('\|');
        die "Неверное количество полей в \'$savedstats[0]\' line.\n"
          unless ($#savedstats == 13);
```

```
        @currentstats = (lstat($savedstats[0]))[0..7,9..12];

        open(F,$savedstats[0]) or die "Невозможно открыть $opt_c:$!\n";
        push(@currentstats,Digest::MD5->new->addfile(F)->hexdigest);
        close(F);

        &printchanged(\@savedstats,\@currentstats)
          if ("@savedstats[1..13]" ne "@currentstats");
    }
    close(CFILE);
}

sub printchanged {
    my($saved,$current)= @_;

    print shift @{$saved},":\n";

    for (my $i=0; $i <= $#{$saved};$i++){
         if ($saved->[$i] ne $current->[$i]){
             print " ".$statnames[$i]." is now ".$current->[$i];
             print " (".$saved->[$i].")\n";
         }
     }
}
```

Изменения сетевых служб

Мы узнали, как обнаружить изменения в локальных файловых системах. Как насчет того, чтобы заметить изменения на других машинах или в службах, ими поддерживаемых? Мы уже видели способы запроса NIS и DNS в главе 5 «Службы имен TCP/IP». Не должна вызвать затруднений проверка изменений в повторяющихся запросах к этим службам. Например, можно притвориться вторичным сервером и запросить копию данных (т. е. выполнить зонную пересылку) с сервера для определенного домена, если, конечно, DNS-сервер настроен так, что позволит сделать это:

```
use Net::DNS;

# принимает два аргумента в командной строке: первый - сервер
# имен, к которому посылается запрос, а второй - интересующий
# нас домен
$server = new Net::DNS::Resolver;
$server->nameservers($ARGV[0]);

print STDERR "Выполняется передача...";
@zone = $server->axfr($ARGV[1]);
die $server->errorstring unless (defined @zone);
print STDERR "готово.\n";
```

```
for $record (@zone){
  $record->print;
}
```

Объединим эту идею с MD5. Вместо того чтобы получать информацию о зоне, давайте просто сгенерируем для нее подпись:

```
use Net::DNS;
use FreezeThaw qw{freeze};
use Digest::MD5 qw(md5);

$server = new Net::DNS::Resolver;
$server->nameservers($ARGV[0]);

print STDERR "Выполняется передача...";
@zone = $server->axfr($ARGV[1]);
die $server->errorstring unless (defined @zone);
print STDERR "готово.\n";

$zone = join('',sort map(freeze($_),@zone));

print "MD5 fingerprint for this zone transfer is: ";
print Digest::MD5->new->add($zone)->hexdigest,"\n";
```

MD5 работает со скалярными данными (сообщение), но не со структурами типа списка хэшей, как `@zone`. Вот почему нужна такая строчка:

```
$zone = join('',sort map(freeze($_),@zone));
```

Для преобразования каждой записи из структуры данных `@zone` в обычные строки воспользуемся модулем `FreezeThaw`, который мы уже видели в главе 9 «Журналы». Перед тем как записи будут склеены в одно большое скалярное значение, они будут отсортированы. Сортировка позволяет проигнорировать порядок, в котором возвращались записи при пересылке зоны.

Пересылка всего файла зоны с сервера – это крайняя мера, особенно, если зона большая, поэтому имеет смысл наблюдать только за важным подмножеством адресов. Такой пример можно найти в главе 5. Кроме того, в целях безопасности было бы неплохо разрешить выполнение зонных пересылок только на минимальном количестве машин.

Все, что мы видели до сих пор, не очень помогло нам преодолеть затруднения. Возможно, следует прояснить несколько вопросов:

- Что если кто-то подделает базу данных MD5 подписей и подставит действительные подписи под поддельные троянские файлы или изменения служб?

- Что если кто-то подделает ваш сценарий таким образом, что он будет только создавать видимость проверки подписей со значениями из базы данных?
- Что если кто-то сделает что-нибудь с модулем MD5 на вашей системе?
- Это уже, конечно, предел паранойи, но что если кто-то сделает что-то с самим исполняемым файлом Perl, одной из его разделяемых библиотек или самим ядром операционной системы?

Обычные ответы на эти вопросы (какими бы они ни были неудовлетворительными): храните хорошие копии всего, что имеет отношение к этому процессу (базы данных подписей, модули, Perl и т. д.) на устройствах, к которым разрешен доступ только для чтения.

Эта головоломка – еще одна иллюстрация бесконечности безопасности. Всегда можно найти что-то, чего можно опасаться.

Обращайте внимание на подозрительную активность

Хорошему ночному сторожу нужно больше, чем просто возможность наблюдения за изменениями. Он также должен иметь возможность реагировать на подозрительные действия или обстоятельства. Обязательно нужно сообщить кому-то о дыре в заборе или необъяснимых ударах среди ночи. Мы можем написать программы, которые возьмут на себя эту роль.

Локальные признаки опасности

К сожалению, зачастую мы учимся замечать признаки подозрительной активности только в результате потерь и желания избежать их в дальнейшем. Достаточно всего нескольких взломов, чтобы заметить, что злоумышленники часто действуют по определенным шаблонам и оставляют за собой предательские улики. Зная, что эти признаки собой представляют, заметить их из Perl не сложно.

После каждого взлома системы безопасности очень важно уделить некоторое время анализу случившегося. Документируйте (по мере своих знаний), куда проникли взломщики, какие инструменты и «дыры» они использовали, что они сделали, кого еще атаковали, что вы сделали в ответ и т. д.

Заманчиво было бы вернуться к обычной жизни и забыть про взлом. Если вы не подвергнетесь этому соблазну, то позже поймете, что инцидент научил вас кое-чему, и что вы не просто потеряли время и силы. Принцип Ницше – «то, что вас не убивает, делает вас сильнее» – часто справедлив и в системном администрировании.

Например, очень часто взломщики, а особенно менее опытные, пытаются замести следы, создавая «скрытые» каталоги для хранения данных. В Unix и Linux они помещают свои программы и вывод подслушивающих программ в каталоги с именами «...» (точка точка точка), «.» (точка пробел) или «Mail» (пробел Mail). Такие имена, скорее всего, останутся без внимания при беглом просмотре вывода команды *ls*.

При помощи инструментов, о которых мы узнали в главе 2, можно легко написать программу для поиска таких имен. Ниже приведена программа, использующая модуль `File::Find` (вызываемый *find.pl*) для поиска «ненормальных» имен каталогов.

```
require "find.pl";

# Обходим нужные файловые системы

&find('.');

sub wanted {

   (-d $_) and                              # каталог
     $_ ne "." and $_ ne ".." and           # не . и не ..

      (/[^-.a-zA-Z0-9+,:;_~$#()]/ or        # содержит "плохой" символ
       /^\.{3,}/ or                        # или начинается как минимум с трех
точек
       /^-/) and                            # или начинается с дефиса

       print "'".&nice($name)."'\n";
}

# печатаем "хорошую" версию имени каталога, то есть, не
# раскрывая управляющие символы. Эта подпрограмма -- лишь
# немного измененная версия &unctrl() из dumpvar.pl
sub nice {
    my($name) = $_[0];
    $name =~ s/([\001-\037\177])/'^'.pack('c',ord($1)^64)/eg;

    $name;
}
```

Помните врезку «Регулярные выражения» из главы 9? Подобные программы, тщательно анализирующие файловую систему, – еще один пример, где высказанная в главе 9 мысль справедлива. Эффективность таких программ зачастую зависит от качества и количества используемых в них регулярных выражений. Слишком мало регулярных выражений – и вы пропустите то, что хотели найти. Слишком много регулярных выражений или они неэффективны – и ваша программа будет выполняться чрезмерно долго и захватит непомерное количество ресурсов. Если использовать слишком общие регулярные выражения – будет найдено много ложных совпадений. Здесь тонкий баланс.

Поиск проблемных образцов

Теперь воспользуемся тем, что мы узнали в главе 9 и перейдем дальше. Только что мы говорили о поиске подозрительных объектов; давайте теперь рассмотрим *образцы* (*patterns*), которые могут быть признаками подозрительной активности. Покажем это на примере программы, выполняющей примитивный анализ журналов в поисках потенциальных взломов.

Пример строится на предположении, что большинство пользователей, удаленно регистрирующихся в системе, делают это из одного и того же места или нескольких мест. Обычно они регистрируются либо с одной машины, либо используя адреса из диапазона, принадлежащего одному и тому же провайдеру. Если вы обнаружите, что пользователь регистрировался из нескольких доменов, это верный признак того, что данная учетная запись была «взломана» и ее пароль стал доступен многим. Очевидно, что такое предположение не справедливо для постоянно перемещающихся пользователей, но если вы обнаружите, что пользователь регистрировался сначала из Бразилии, а потом из Финляндии с интервалом в два часа, то это достаточный повод заподозрить что-то неладное.

Рассмотрим программу, реализующую поиск таких признаков. Сама программа написана для Unix, но демонстрируемые в ней приемы не зависят от платформы. Во-первых, это встроенная документация. Неплохо поместить нечто подобное ближе к началу программы для тех, кто будет смотреть на исходный код. Перед тем как двигаться дальше, обязательно взгляните, какие аргументы поддерживаются программой:

```
sub usage {
    print <<"EOU"
lastcheck - проверяет вывод команды last, чтобы определить, регистрировались
ли какие-нибудь пользователи более чем с N доменов (идея Даниэля Райнхарта)

   ИСПОЛЬЗОВАНИЕ:  lastcheck [args], где вместо args может быть:
    -i:           для IP-адресов, считать сеть класса C одним "доменом"
    -h:           помощь (это сообщение)
    -f <domain>   считать только другие домены, определяет домашний домен
    -l <command>: использовать <command> вместо используемой по умолчанию
                  /usr/ucb/last
                  замечание: проверка формата вывода не производится!
    -m <#>:       максимальное количество допустимых уникальных доменов, по
                  умолчанию - 3
    -u <user>:    выполнить проверку только для этого имени пользователя EOU
    exit;
}
```

Сначала мы анализируем аргументы из командной строки. В строке `getopts` просматриваются аргументы программы и соответствующим

образом устанавливается $opt_<*буква_флага*>. Двоеточие после буквы говорит о том, что этот параметр принимается как аргумент:

```
use Getopt::Std;        # стандартный процессор параметров
getopts('ihf:l:m:u:'); # анализируем данные, введенные пользователем

&usage if (defined $opt_h);

# допустимое количество уникальных доменов
$maxdomains = (defined $opt_m) ? $opt_m : 3;
```

В следующих строчках реализуется выбор, сделанный в пользу переносимости (но в ущерб эффективности) – об этом мы говорили в главе 9. На этот раз мы решили вызвать внешнюю программу. Для того чтобы сделать программу менее переносимой, но несколько более эффективной, можно было использовать unpack(), о чем тоже говорилось в той главе:

```
$lastex = (defined $opt_l) ? $opt_l : "/usr/ucb/last";

open(LAST,"$lastex|") || die "Невозможно выполнить программу $lastex:$!\n";
```

Перед тем как двигаться дальше, давайте взглянем на хэш списков, с помощью которого программа обрабатывает данные, полученные от *last*. Ключами этого хэша являются имена пользователей, а значениями – ссылки на список уникальных доменов, с которых регистрировался пользователь.

К примеру, запись может выглядеть так:

```
$userinfo { laf } = [ 'ccs.neu.edu', 'xerox.com', 'foobar.edu' ]
```

Эта запись говорит о том, что пользователь *laf* регистрировался с доменов *ccs.neu.edu*, *xerox.com* и *foobar.edu*.

Начинаем мы с того, что обходим в цикле вывод команды *last*. На нашей системе он выглядит примерно так:

```
cindy    pts/10   sinai.ccs.neu.ed  Fri Mar 27 13:51   still logged in
michael  pts/3    regulus.ccs.neu.  Fri Mar 27 13:51   still logged in
david    pts/5    fruity-pebbles.c  Fri Mar 27 13:48   still logged in
deborah  pts/5    grape-nuts.ccs.n  Fri Mar 27 11:43 - 11:53  (00:09)
barbara  pts/3    152.148.23.66     Fri Mar 27 10:48 - 13:20  (02:31)
jerry    pts/3    nat16.aspentec.c  Fri Mar 27 09:24 - 09:26  (00:01)
```

Заметьте, что имена узлов (в 3-й колонке) в выводе команды *last* усечены. В главе 9 мы уже говорили об ограничениях на длину имени узла, но до сих пор мы обходили стороной это препятствие. Когда мы попробуем заполнить нашу структуру данных, проблемы станут очевидными.

Раньше в цикле `while` мы пытались пропустить строчки, содержащие данные, которые нас не интересуют. Как правило, проверка особых случаев в самом начале цикла до какой-либо обработки данных (например, при помощи `split()`) – неплохая идея. Это позволяет программе быстро определить, что можно пропустить определенную строчку и перейти к дальнейшему чтению данных:

```
while (<LAST>){

    # игнорируем специальных пользователей
    next if /^reboot\s|^shutdown\s|^ftp\s/;

    # если использовался ключ -u для определения конкретного
    # пользователя, пропускаем все записи, не относящиеся к
    # нему (имя сохраняется в $opt_u функцией getopts).
    next if (defined $opt_u && !/^$opt_u\s/);

    # игнорируем вход с консоли X
    next if /:0\s+:0/;

    # ищем имя пользователя, терминал и имя удаленного узла
    ($user, $tty,$host) = split;

    # игнорируем, если запись в журнале имеет "плохое" имя
    # пользователя
    next if (length($user) < 2);

    # игнорируем, если для данного имени нет информации о домене
    next if $host !~ /\./;

    # ищем доменное имя узла (см. приведенное ниже объяснение)
    $dn = &domain($host);

    # игнорируем, если доменное имя фиктивное
    next if (length ($dn) < 2);

    # игнорируем эту строку, если она находится в домене,
    # заданном ключом -f
    next if (defined $opt_f && ($dn =~ /^$opt_f/));

    # если мы не встречали раньше имя этого пользователя,
    # просто создаем список доменов для этого пользователя и
    # сохраняем эту информацию в хэше списков
    unless (exists $userinfo{$user}){
       $userinfo{$user} = [$dn];
    }
    # в противном случае нам придется нелегко;
    # см. приведенное ниже объяснение
    else {
      &AddToInfo($user,$dn);
   }
}
close(LAST);
```

Теперь рассмотрим отдельные подпрограммы, предназначенные для разрешения сложных ситуаций в программе. Первая подпрограмма `&domain()` принимает полностью заданное доменное имя, т. е. имя узла с полным доменным именем, и возвращает лучшую догадку о доменном имени для этого узла. Есть две причины, по которым подпрограмма должна быть довольно умна:

1. Не все имена узлов из журналов будут именами. Это вполне может быть и IP-адрес. В этом случае, если пользователь устанавливает ключ *–i*, мы полагаем, что любой получаемый нами IP-адрес – это адрес сети класса C, разделенной на подсети по границе байта. На практике это означает, что доменным именем мы считаем первые три октета адреса. Это позволяет нам считать регистрацию в системе с адресов 192.168.1.10 и 192.168.1.12 регистрацией из одного логического источника. Вероятно, это не лучшее предположение, но это лучшее, что мы можем сделать, не обращаясь при этом к другому источнику информации (да и в большинстве случаев это работает). Если пользователь не указывает ключ *–i*, мы считаем весь IP-адрес доменом.
2. Как говорилось раньше, имена узлов могут быть усечены. Это приводит к тому, что мы имеем дело с неполными записями, подобными `grape-nuts.ccs.n` и `nat16.aspentec.c`. Это не так страшно, как кажется, потому что полностью определенное имя домена в журнале каждый раз будет усекаться на одном и том же месте. В подпрограмме `&AddToInfo()` мы попробуем сделать все возможное, чтобы справиться с этим ограничением. Но об этом чуть позже.

А пока вернемся к программе:

```
# принимаем полностью определенное имя домена и пытаемся
# определить домен
sub domain{
    # ищем IP-адреса
    if ($_[0] =~ /^\d+\.\d+\.\d+\.\d+$/) {

          # если пользователь не указал ключ -i, просто
       # возвращаем IP-адрес как есть
          unless (defined $opt_i){
              return $_[0];
          }

          # иначе возвращаем все, кроме последнего октета
          else {
              $_[0] =~ /(.*)\.\d+$/;
              return $1;
          }
    }

    # если мы имеем дело не с IP-адресом
    else {
```

```
            # переводим все в нижний регистр, чтобы потом было
        # проще и быстрее обрабатывать информацию
            $_[0] = lc($_[0]);

            # затем возвращаем все после первой точки
            $_[0] =~ /^[^.]+\.(.*)/;
            return $1;
    }
}
```

Следующая очень короткая подпрограмма заключает в себе самую сложную часть программы. Подпрограмма `&AddToInfo()` работает с усеченными именами узлов и сохраняет информацию в хэш-таблицу. Мы применим способ сравнения подстрок, который может пригодиться и в ином контексте.

В нашем случае было бы неплохо, если бы все эти имена доменов считались бы и сохранялись бы в массиве уникальных имен доменов для пользователя как одно имя:

```
ccs.neu.edu
ccs.neu.ed
ccs.n
```

Решая, является ли имя домена уникальным, необходимо проверить три вещи:

1. Совпадает ли имя домена полностью с чем-нибудь, что уже сохранено для этого пользователя?
2. Является ли это имя домена подстрокой уже сохраненных данных?
3. Являются ли подстрокой проверяемого имени домена сохраненные данные?

Если верно что-либо из этого списка, значит, нет необходимости добавлять новую запись к структуре данных, поскольку эквивалентная подстрока уже сохранена в списке доменов для пользователя. Если выполняется пункт 3, мы заменим сохраненную запись текущей, если, конечно, мы сохраняем строки максимальной длины. Внимательные читатели могли заметить, что выполнение первых двух пунктов можно проверять одновременно, поскольку точное совпадение эквивалентно совпадению подстроки по всем символам.

Если же не справедлив ни один из этих случаев, то необходимо сохранить новую запись. Посмотрим сначала на код, а потом обсудим, как он работает:

```
sub AddToInfo{
    my($user, $dn) = @_;

    for (@{$userinfo{$user}}){
```

```
        # проверка 1-го и 2-го случаев: есть ли полное или
        # частичное совпадение?
        return if (index($_,$dn) > -1);

        # проверка 3-го случая, то есть, являются ли подстрокой
        # сохраненные данные
        if (index($dn,$_) > -1){
          $_ = $dn; # меняем местами текущее и сохраненное
                    # значения
          return;
        }
      }

      # в противном случае это новый домен, добавляем его в список
      push @{$userinfo{$user}}, $dn;
}
```

Конструкция `@{$userinfo{$user}}` возвращает список доменов, сохраненных для этого пользователя. Мы обходим в цикле все элементы из этого списка, проверяя, можно ли найти среди них `$dn`. Если можно, то мы выходим из подпрограммы, т. к. эквивалентная подстрока уже сохранена.

Если эта проверка пройдена, то можно перейти к пункту 3. Мы проверяем каждую запись из списка, чтобы выяснить, встречается ли она в текущем домене. Мы заменяем запись из списка на текущий домен, если совпадение найдено, тем самым сохраняя более длинную из двух строк. Поскольку это не вредит, замена производится и при точном совпадении. Мы переписываем запись, используя специальное свойство операторов `for` и `foreach` в Perl. Присваивая значение переменной `$_` в середине цикла `for`, Perl в действительности присваивает значение текущему элементу списка. Переменная цикла становится псевдонимом для переменной списка. После того как мы поменяли местами значения, можно выходить из подпрограммы. Если были пройдены все три проверки, то в последней строке к списку доменов для пользователя добавляется рассматриваемое имя домена.

Это все, что касается «кровавых» деталей просмотра файла и создания структуры данных. Чтобы завершить эту программу, рассмотрим всех найденных пользователей и проверим, со скольких доменов они регистрировались (т. е. выясним длину сохраненного для каждого из них списка). По тем записям, для которых найдено больше доменов, чем можно, мы выводим полный список:

```
for (sort keys %userinfo){
    if ($#{$userinfo{$_}} > $maxdomains){
         print "\n\n$_ регистрировался с:\n";
         print join("\n",sort @{$userinfo{$_}});
    }
}
print "\n";
```

Что ж, программу вы видели и вас, вероятно, интересует, действительно ли работает этот метод. Вот реальный отрывок вывода этой программы для пользователя, чей пароль был украден:

```
username регистрировался с:
38.254.131
bu.edu
ccs.neu.ed
dac.neu.ed
hials.no
ipt.a
tnt1.bos1
tnt1.bost
tnt1.dia
tnt2.bos
tnt3.bos
tnt4.bo
toronto4.di
```

Некоторые из этих записей выглядят нормально для пользователя, живущего в Бостоне. Однако запись *toronto4.di* выглядит несколько подозрительной, а сайт *hials.no* вообще находится в Норвегии. Схвачены с поличным!

Программу можно усовершенствовать, добавив проверку времени или сравнение с другими журналами, например, полученных при помощи *tcpwrappers*. Но как видите, поиск шаблонов часто важен сам по себе.

Протокол SNMP

Давайте отвлечемся от вопросов безопасности и перейдем к более общим темам наблюдения. В предыдущем разделе мы рассмотрели способ наблюдения за определенной сетевой службой. Простой протокол управления сетью (SNMP) совершает «квантовый скачок», предлагая общий способ удаленного наблюдения и настройки сетевых устройств и компьютеров. Стоит только разобраться с основами протокола SNMP, и вы сможете применять его для хранения таблиц (а зачастую для конфигурирования) практически любого устройства в вашей сети.

По правде говоря, *простой* протокол управления сетью не очень-то и прост. С этим предметом связано много тонкостей. Если вы еще не знакомы с SNMP, загляните в приложение E «Двадцатиминутное руководство по SNMP».

Использование протокола SNMP из Perl

Один из способов использовать протокол SNMP из Perl – вызвать программу, работающую в командной строке, наподобие UCD-SNMP, приведенной в демонстрационных целях в приложении E. Этот процесс оче-

виден и ничем не отличается от вызова внешних программ, о чем мы раньше упоминали в книге. Ничему новому тут научиться нельзя, так что не будем уделять этому подходу много времени. Приведу лишь одно предостережение: тем, кто использует SNMPv1 или SNMPv2C, скорее всего, придется указывать имя сообщества (community name) в командной строке. Если эта программа выполняется в многопользовательской системе, любой, кто обратится к списку процессов, сможет увидеть имя сообщества и завладеть «ключами от города». Эта угроза существует в примерах, выполняемых в командной строке из приложения E, но она становится более серьезной в автоматически выполняемых программах, которые неоднократно вызывают внешние программы. Лишь для наглядности в следующих примерах имя узла и имя сообщества определяются в командной строке. В настоящих программах от этого нужно избавиться.

Если мы не вызываем внешнюю программу для выполнения SNMP-операций из Perl, другим вариантом является использование модуля SNMP. Существует по крайней мере три очень похожих модуля: `Net::SNMP` Дэвида M. Тауна (David M. Town), *SNMP_Session.pm*, написанный Саймоном Лейненом (Simon Leinen) и «SNMP Extension Module v3.1.0 for the UCD SNMPv3 Library» (Модуль SNMP расширений v3.1.0 для библиотек UCD SNMPv3, который мы будем называть просто `SNMP` из-за способа его загрузки) Дж. С. Марзота (G.S. Marzot). Все три модуля реализуют SNMPv1. `Net::SNMP` и `SNMP` частично поддерживают SNMPv2. И лишь в `SNMP` предлагается некоторая поддержка SNMPv3.

Помимо различного уровня поддержки протокола SNMP, самое большое различие между этими модулями заключается в их зависимости от внешних библиотек. Первые два (`Net::SNMP` и *SNMP_Session.pm*) реализованы только на Perl, а `SNMP` должен быть скомпонован с прекомпилированной библиотекой UCD-SNMP. Основной недостаток применения `SNMP` – это дополнительная зависимость и лишний шаг компиляции (если считать, что вы можете собрать библиотеку UCD-SNMP на своей платформе).

Положительная сторона зависимости от библиотеки UCD-SNMP в том, что она придает модулю дополнительную силу и мощь. Например, `SNMP` может анализировать файлы описания административных баз данных (Management Information Base, MIB) и выводить для анализа SNMP-пакеты, чего не могут два других модуля. Для уменьшения разницы в возможностях существуют другие модули (например модуль `SNMP::MIB::Compiler` Фабьена Тассэна (Fabien Tassin) способен анализировать MIB), но если нужно, чтобы один модуль выполнял всю работу, то лучше модуля `SNMP` ничего нет.

Давайте рассмотрим небольшой пример на Perl. Для того чтобы узнать количество интерфейсов на определенном устройстве, можно обратиться к переменной `interfaces.ifNumber`. Сделать это при помощи модуля `Net::SNMP` очень просто:

```
use Net::SNMP;

# в качестве аргументов задаются имя узла и имя сообщества
($session,$error) = Net::SNMP->session(Hostname => $ARGV[0],
                                       Community => $ARGV[1]);

die "Ошибка сеанса: $error" unless ($session);

# iso.org.dod.internet.mgmt.mib-2.interfaces.ifNumber.0 =
#   1.3.6.1.2.1.2.1.0
$result = $session->get_request("1.3.6.1.2.1.2.1.0");

die "Ошибка запроса: ".$session->error unless (defined $result);

$session->close;

print "Количество интерфейсов: ".$result->{"1.3.6.1.2.1.2.1.0"}."\n";
```

Если указать на рабочую станцию с интерфейсом обратной петли и интерфейсом Ethernet, программа вернет: `Количество интерфейсов: 2`; если же указать на портативный компьютер с интерфейсом обратной петли и интерфейсами Ethernet и PPP, то программа вернет `Количество интерфейсов: 3`; для небольшого маршрутизатора она вернет `Количество интерфейсов: 7`.

Важно обратить внимание на использование идентификаторов объекта (Object Identifiers, OID) вместо имен переменных. И `Net::SNMP`, и *SNMP_Session.pm* обрабатывают взаимодействие только по протоколу SNMP. Они не претендуют на выполнение каких-либо второстепенных задач, связанных с SNMP, например, анализ описаний SNMP MIB. Для выполнения этих действий нужно обратиться к другим модулям, таким как `SNMP::MIB::Compiler` или *SNMP_util.pm* Майка Митчела (Mike Mitchell) для применения их с `SNMP_Session.pm` (не путайте с модулем `SNMP::Util` Вейна Маркетта (Wayne Marquette), который используется с модулем `SNMP`).

Для тех, кто предпочитает работать с текстовыми идентификаторами вместо численных, не создавая самостоятельно таблицу соответствия и не используя дополнительные модули, остается единственный вариант – обратиться к модулю `SNMP`, в котором есть встроенный анализатор MIB. Давайте рассмотрим таблицу ARP (Address Resolution Protocol, протокол преобразования адресов) на машине при помощи этого модуля:

```
use SNMP;

# в качестве аргументов задаются имя узла и имя сообщества
$session = new SNMP::Session(DestHost => $ARGV[0], Community => $ARGV[1],
                             UseSprintValue => 1);
```

```
die "Ошибка создания сессии: $SNMP::Session::ErrorStr" unless
  (defined $session);

# создаем структуру данных для команды getnext
$vars = new SNMP::VarList(['ipNetToMediaNetAddress'],
                          ['ipNetToMediaPhysAddress']);

# получаем первую запись
($ip,$mac) = $session->getnext($vars);
die $session->{ErrorStr} if ($session->{ErrorStr});

# и все последующие
while (!$session->{ErrorStr} and
       $$vars[0]->tag eq "ipNetToMediaNetAddress"){
    print "$ip -> $mac\n";
    ($ip,$mac) = $session->getnext($vars);
};
```

Вот как выглядит пример вывода этой программы:

```
192.168.1.70 -> 8:0:20:21:40:51
192.168.1.74 -> 8:0:20:76:7c:85
192.168.1.98 -> 0:c0:95:e0:5c:1c
```

Этот пример похож на предыдущий, где использовался модуль `Net::SNMP`. Для выявления различий рассмотрим его подробнее:

```
use SNMP;

$session = new SNMP::Session(DestHost => $ARGV[0], Community => $ARGV[1],
                             UseSprintValue => 1);
```

После загрузки модуля `SNMP` мы создаем объект сессии так же, как и в случае с `Net::SNMP`. Дополнительный аргумент `UseSprintValue => 1` указывается лишь для того, чтобы выводить возвращаемые значения более аккуратно. Если этого не сделать, то Ethernet-адреса будут выводиться в закодированном виде.

```
# создаем структуру данных для команды getnext
$vars = new SNMP::VarList(['ipNetToMediaNetAddress'],
                          ['ipNetToMediaPhysAddress']);
```

`SNMP` со своими командами использует такие простые строки, как `sysDescr.0`, но предпочитает работать со специальным объектом, который называет «Varbind». Модуль применяет эти объекты для хранения значений, возвращаемых в результате запросов. Например, в нашей программе для отправки запроса `get-next-request` вызывается метод `getnext()`, прямо как в примере таблицы маршрутизации из приложения Е. Правда, на этот раз `SNMP` сохранит полученные индексы в Varbind, и нам не придется вручную следить за ними. Используя этот мо-

дуль, достаточно передать Varbind методу getnext, если необходимо получить следующее значение.

Varbind – это обычный анонимный Perl-массив, состоящий из четырех элементов: obj, iid, val и type. Нас интересуют только obj и iid. Первый элемент, obj – это объект, к которому посылается запрос. obj может быть задан в одном из нескольких форматов. В данном случае мы пользуемся форматом *leaf identifier*, т. е. определяем лист дерева, с которым мы связаны. IpNetToMediaNetAddress – это лист дерева:

```
.iso.org.dod.internet.mgmt.mib-
2.ip.ipNetToMediaTable.ipNetToMediaEntry.ipNetToMediaNetAddress
```

Второй элемент в Varbind – это iid, или идентификатор экземпляра (instance identifier). В предыдущем примере мы использовали только 0 (например system.sysDescr.0), поскольку видели объекты, имеющие только один экземпляр. Скоро мы увидим примеры, где iid может иметь и другие значения, отличные от нуля. Например, позже мы сошлемся на определенный сетевой интерфейс в коммутаторе с несколькими Ethernet-интерфейсами. Для get необходимо указывать только два компонента Varbind – obj и iid. Методу getnext iid не нужен, т. к. он по умолчанию возвращает следующий экземпляр.

В приведенной выше строке используется метод VarList(), создающий список из двух Varbind, для каждого из которых определен только один элемент obj. Этот список мы предаем методу getnext():

```
# получаем первую запись
($ip,$mac) = $session->getnext($vars);
die $session->{ErrorStr} if ($session->{ErrorStr});
```

getnext() возвращает значения, полученные из запроса, и соответствующим образом обновляет структуры данных Varbind. Теперь остается только вызывать getnext() до тех пор, пока мы не дойдем до конца таблицы:

```
while (!$session->{ErrorStr} and
       $$vars[0]->tag eq "ipNetToMediaNetAddress"){
        print "$ip -> $mac\n";
       ($ip,$mac) = $session->getnext($vars);
};
```

Давайте вернемся к миру безопасности, чтобы рассмотреть последний пример о SNMP. Задачу, которую мы будем решать, сложно или, по крайней мере, скучно выполнять при помощи имеющихся утилит командной строки.

Задача заключается в следующем: вас попросили выследить в коммутируемой сети Ethernet (switched Ethernet network) плохо ведущего себя пользователя. Единственная информация, которой вы обладаете, это Ethernet-адрес машины, на которой работает пользователь. Это не тот

Ethernet-адрес, который содержится в файле (сам файл можно хранить в базе данных узлов, рассмотренной в главе 5, если эту базу несколько расширить), а прослушать коммутируемую сеть у вас не получится, так что придется проявить сообразительность, чтобы вычислить эту машину. Лучший выход из этого положения – обратиться к одному или всем коммутаторам и узнать, видели ли они этот адрес на одном из своих портов.

Для большей конкретизации скажем, что сеть состоит из нескольких коммутаторов Cisco Catalyst 5500; это даст нам возможность указать на конкретные переменные MIB. Основные методы, которые мы будем использовать для решения этой проблемы, также применимы для других продуктов и других производителей. Если информация относится только к определенному коммутатору или производителю, мы об этом скажем. А теперь давайте шаг за шагом рассмотрим решение проблемы.

Как и раньше, сначала необходимо выполнить поиск по корректным файлам модулей MIB. Обратившись к службе технической поддержки Cisco, мы узнаем, что нам понадобится доступ к четырем объектам:

- `vlanTable`, которую можно найти в `enterprises.cisco.workgroup.ciscoStackMIB.vlanGrp` из описания CISCO-STACK-MIB.
- `dot1dTpFdbTable` (таблица прозрачной трансляции портов), которую можно найти в `dot1dBridge. dot1dTp` из описания RFC1493 BRIDGE-MIB.
- `dot1dBasePortTable`, которую можно найти в `dot1dBridge.dot1dBase` в том же RFC.
- `ifXTable`, которую можно найти в RFC1573 IF-MIB (Интерфейсы).

Зачем нужны четыре различные таблицы? В каждой из них есть что-то нужное нам, но ни в одной нет целиком всей информации, которую мы ищем. Первая таблица предоставляет список VLAN (Virtual Local Area Networks, виртуальные локальные сети), или виртуальных «сегментов сети» на коммутаторе. В Cisco решили хранить на коммутаторе отдельные таблицы для каждой виртуальной локальной сети, поэтому нам придется за один раз запрашивать информацию об одной виртуальной локальной сети. Подробнее об этом чуть позже.

Вторая таблица предоставляет список Ethernet-адресов и *номер порта* (*bridge port*) на коммутаторе, на котором этом адрес был замечен последний раз. К сожалению, этот номер порта в коммутаторе является внутренним параметром, и он не соответствует имени физического порта на нем же. Нам нужно знать имя физического порта, т. е. с какой сетевой карты и порта последний раз «общалась» машина с указанным Ethernet-адресом, так что нужно «копать» дальше.

Таблицы, связывающей номер порта (bridge port) с именем физического порта, не существует (что было бы очень просто), но `dot1dBasePortTable` позволяет выяснить соответствие между номером порта и номером ин-

терфейса. Имея номер интерфейса, можно найти его в таблице `ifXTable` и получить имя порта.

Вот схема четырехуровневого согласования, необходимого для выполнения поставленной перед нами задачи (рис. 10.1).

А вот программа, в которой все четыре таблицы собраны вместе для вывода нужной информации:

```
use SNMP;

# Дополнительные модули MIB, нужные нам, которые можно найти в
# том же каталоге, что и сценарий
$ENV{'MIBFILES'}=
  "CISCO-SMI.my:FDDI-SMT73-MIB.my:CISCO-STACK-MIB.my:BRIDGE-MIB.my";

# соединяемся и получаем список виртуальных локальных сетей с
# этого коммутатора
$session = new SNMP::Session(DestHost => $ARGV[0],
                             Community => $ARGV[1]);
die "Ошибка создания сессии: $SNMP::Session::ErrorStr" unless
  (defined $session);

# enterprises.cisco.workgroup.ciscoStackMIB.vlanGrp.vlanTable.vlanEntry
# из CISCO-STACK-MIB
$vars = new SNMP::VarList(['vlanIndex']);

$vlan = $session->getnext($vars);
die $session->{ErrorStr} if ($session->{ErrorStr});

while (!$session->{ErrorStr} and $$vars[0]->tag eq "vlanIndex"){

    # На CISCO CATALYST 5XXX просто не может быть более 1000
    # виртуальных локальных сетей (этот предел, скорее всего,
    # отличается для различных коммутаторов)
    push(@vlans,$vlan) if $vlan < 1000;

    $vlan = $session->getnext($vars);
};

undef $session,$vars;

# для каждой виртуальной локальной сети запрашиваем номер
# порта, номер интерфейса, связанного с этим портом и затем имя # интерфейса
foreach $vlan (@vlans){
    # обратите внимание на использование "индексирования строки
    # сообщества" при настройке соединения
    $session = new SNMP::Session(DestHost => $ARGV[0],
                                 Community => $ARGV[1]."@".$vlan,
```

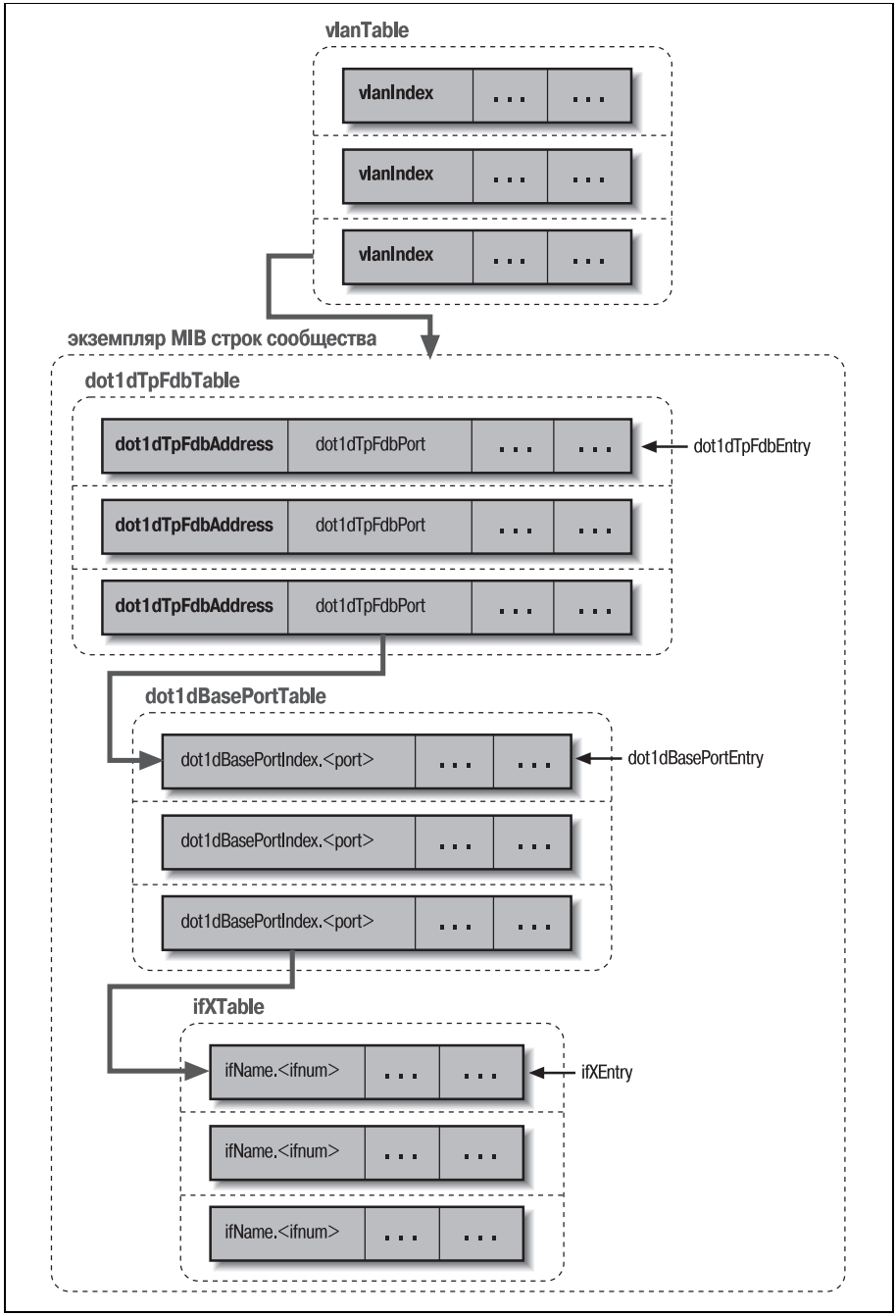

Рис. 10.1. *Набор SNMP-запросов, необходимых для поиска имени порта на Cisco 5000*

```
                                    UseSprintValue => 1);

    die "Ошибка создания сессии: $SNMP::Session::ErrorStr"
      unless (defined $session);

    # из таблицы прозрачной трансляции портов из
    # dot1dBridge.dot1dTp.dot1dTpFdbTable.dot1dTpFdbEntry
    # из RFC1493 BRIDGE-MIB
    $vars = new SNMP::VarList(['dot1dTpFdbAddress'],['dot1dTpFdbPort']);

    ($macaddr,$portnum) = $session->getnext($vars);
    die $session->{ErrorStr} if ($session->{ErrorStr});

    while (!$session->{ErrorStr} and
           $$vars[0]->tag eq "dot1dTpFdbAddress"){

        # dot1dBridge.dot1dBase.dot1dBasePortTable.dot1dBasePortEntry
        # из RFC1493 BRIDGE-MIB
        $ifnum =
          (exists $ifnum{$portnum}) ? $ifnum{$portnum} :
            ($ifnum{$portnum} =
               $session->get("dot1dBasePortIfIndex\.$portnum"));

        # из ifMIB.ifMIBObjects.ifXTable.ifXEntry из RFC1573 IF-MIB
        $portname =
          (exists $portname{$ifnum}) ? $portname{$ifnum} :
            ($portname{$ifnum}=$session->get("ifName\.$ifnum"));

        print "$macaddr в виртуальной локальной сети $vlan на $portname\n";

        ($macaddr,$portnum) = $session->getnext($vars);
    };

    undef $session, $vars, %ifnum, %portname[1];
}
```

Если вы уже читали приложение E, то большая часть программы должна быть вам знакома. Приведем лишь комментарии по новым фрагментам:

```
$ENV{'MIBFILES'}=
  "CISCO-SMI.my:FDDI-SMT73-MIB.my:CISCO-STACK-MIB.my:BRIDGE-MIB.my";
```

[1] **Если запустить данную программу с ключом *–w*, который устанавливает режим предупреждений, Perl начнет выдавать предупреждения на этот оператор, из которых можно понять, что здесь не все в порядке с приоритетами операций. Фактически, запись** `undef $session, $vars, %ifnum, %portname;` **» означает следующее:** `(undef $session), $vars, %ifnum, %portname;` **. Синтаксис не позволяет написать** `undef ($session, $vars, %ifnum, %portname);` **», поэтому правильной была бы более длинная строка:** `undef $session, undef $vars, undef %ifnum, undef %portname;` **. –** ***Примеч. науч. ред.***

Эта программа устанавливает переменную окружения MIBFILES для библиотеки UCD-SNMP. Будучи установленной, эта переменная дает инструкцию библиотеке проанализировать приведенный список дополнительных файлов для определения объекта MIB. В этом списке присутствует один странный файл модуля MIB – *FDDI-SMT73-MIB.my*. Он добавлен из-за того, что *CISCO-STACK-MIB.my* имеет следующий оператор для включения некоторых определений из других записей MIB:

```
IMPORTS
        MODULE-IDENTITY, OBJECT-TYPE, Integer32, IpAddress, TimeTicks,
        Counter32, Counter64, NOTIFICATION-TYPE
                FROM SNMPv2-SMI
        DisplayString, RowStatus
                FROM SNMPv2-TC
        fddimibPORTSMTIndex, fddimibPORTIndex
                FROM FDDI-SMT73-MIB
        OwnerString
                FROM IF-MIB
        MODULE-COMPLIANCE, OBJECT-GROUP
                FROM SNMPv2-CONF
        workgroup
                FROM CISCO-SMI;
```

Хотя мы и не ссылаемся на объекты, использующие fddimibPORTSMTIndex или fddimibPORTIndex, мы все же добавляем (намеренно) этот файл в список, чтобы анализатор MIB не выдавал сообщений. Все остальные определения MIB из этого оператора IMPORTS включаются либо в списке, либо в списке по умолчанию библиотеки. При работе с MIB вам часто придется заглядывать в раздел IMPORTS модуля MIB для изучения зависимостей этого модуля.

Двигаясь дальше, мы находим еще один странный оператор:

```
$session = new SNMP::Session(DestHost => $ARGV[0],
                             Community => $ARGV[1]."@".$vlan,
                             UseSprintValue => 1);
```

Вместо того чтобы просто передать имя сообщества, введенное пользователем, мы дописываем к нему нечто вроде *@VLAN-NUMBER*. На жаргоне Cisco это означает «индексация строки сообщества». При работе с виртуальными сетями и мостами устройства Cisco следят за несколькими «экземплярами» MIB, по одному на каждую виртуальную сеть. Наша программа выполняет одни и те же запросы для каждой виртуальной сети, найденной на коммутаторе:

```
$ifnum =
         (exists $ifnum{$portnum}) ? $ifnum{$portnum} :
            ($ifnum{$portnum} =
                $session->get("dot1dBasePortIfIndex\.$portnum"));
```

Приведем два комментария к этому отрывку. Во-первых, по ряду причин мы используем в качестве аргумента `get()` простую строку. Хотя с таким же успехом это могло быть что-то более Varbind-подобное:

```
($ifnum{$portnum}=$session->get(['dot1dBasePortIfIndex',$portnum]));
```

Во-вторых, обратите внимание, что тут мы выполняем простое кэширование. Перед тем как выполнять `get()`, мы смотрим в простую хэш-таблицу (`%ifnum`), чтобы проверить, выполнялся ли уже этот запрос. Если нет, то запрос выполняется, а его результаты помещаются в хэш-таблицу. После просмотра всей виртуальной локальной сети кэширующий хэш удаляется (`undef %ifnum`)[1], чтобы исключить возможность дезинформации при использовании данных для другой виртуальной локальной сети.

При написании программ для работы с SNMP о таких вещах стоит помнить всегда. Если вы хотите «пожалеть» свою сеть и сетевые устройства, очень важно посылать возможно более компактные запросы как можно реже. Если не проявить в этом благоразумия, то, отвечая на шквал запросов, устройство выделит меньше ресурсов для выполнения обычных задач.

Вот отрывок из полученных в результате выполнения программы данных:

```
"00 10 1F 2D F8 FB " в виртуальной локальной сети 1 на 1/1
"00 10 1F 2D F8 FD " в виртуальной локальной сети 1 на 1/1
"08 00 36 8B A9 03 " в виртуальной локальной сети 115 на 2/18
"08 00 36 BA 16 03 " в виртуальной локальной сети 115 на 2/3
"08 00 36 D1 CB 03 " в виртуальной локальной сети 115 на 2/15
```

Эту программу улучшить нетрудно. Помимо более аккуратного или более упорядоченного вывода можно сохранять состояния между запусками. При каждом запуске программа могла бы сообщать об изменениях: какие адреса появились, какие порты изменились и т. д. Правда, нужно заметить, что большая часть коммутаторов относится к разновидности «обучаемых», поэтому они считают устаревшими записи об адресах, которые они давно не встречали. Это означает, что программу необходимо запускать как минимум с той же периодичностью, с которой устаревает информация о порте.

Опасность на проводе

SNMP хорош для активного наблюдения (а также в некоторых предусматривающих реакцию ситуациях мониторинга, когда используются

[1] После того, как мы исправили ошибку с `undef` на стр. 415, хэш действительно стал удаляться, и логика работы программы восстанавливается. – *Примеч. науч. ред.*

SNMP-прерывания (traps)), но он не всегда помогает, если происходит что-то незапланированное, например авария в сети. В таких случаях, возможно, придется наблюдать за сетью другими способами, которые не охватываются доступными SNMP переменными.

Perl спасает положение

Вот правдивая история об этом, рассказывающая, как в таких случаях может помочь Perl. Однажды субботним вечером я, как обычно, зарегистрировался на одной из машин в моей сети, чтобы почитать почту, и, к своему удивлению, обнаружил, что наш почтовый и веб-серверы находятся практически «при смерти». Попытки прочитать или отправить почту или посмотреть на веб-сайт заканчивались медленными ответами, обрывами соединений и невозможностью установить соединение. Почтовая очередь начинала достигать критического размера.

Первым делом я проверил состояние серверов. Интерактивные ответы были нормальными, загрузка процессора – велика, но не смертельна. Признаком проблем было количество запущенных почтовых процессов. В соответствии с файлами журналов большее, чем обычно, количество процессов было вызвано тем, что многие транзакции не были завершены. Повисшими были процессы, запущенные для обработки входящих соединений, они и увеличивали загрузку. А эта загрузка уже мешала появлению новых исходящих соединений. Такое странное поведение сети заставило меня изучить таблицу текущих соединений сервера с помощью *netstat*.

Последний столбец вывода *netstat* говорил о том, что с внешним миром действительно было установлено много соединений. Большой неприятностью было состояние этих соединений. Вместо того чтобы выглядеть примерно так:

```
tcp    0      0  mailhub.3322    mail.mel.aone.ne.smtp  ESTABLISHED
tcp    0      0  mailhub.3320    edunet.edunet.dk.smtp  CLOSE_WAIT
tcp    0      0  mailhub.1723    kraken.mvnet.wne.smtp  ESTABLISHED
tcp    0      0  mailhub.1709    plover.net.bridg.smtp  CLOSE_WAIT
```

они больше напоминали следующее:

```
tcp    0      0  mailhub.3322    mail.mel.aone.ne.smtp  SYN_RCVD
tcp    0      0  mailhub.3320    edunet.edunet.dk.smtp  SYN_RCVD
tcp    0      0  mailhub.1723    kraken.mvnet.wne.smtp  SYN_RCVD
tcp    0      0  mailhub.1709    plover.net.bridg.smtp  CLOSE_WAIT
```

На первый взгляд, это было похоже на классическую атаку типа «отказ от обслуживания» (Denial of Service), называемую SYN Flood, или атакой SYN-ACK. Давайте отвлечемся на некоторое время и поговорим о том, как работает протокол TCP/IP, чтобы понять, что собой представляют эти атаки.

Каждое TCP/IP-соединение начинается с «рукопожатия» между участниками. Это позволяет и инициатору, и получателю сообщить о своей готовности к беседе. Первый шаг предпринимает инициатор соединения, посылая получателю пакет SYN (от SYNchronize – синхронизировать). Если получатель готов к «общению», он посылает в ответ пакет SYN-ACK, подтверждение (от ACKnowledgment) запроса, и записывает в таблице отложенных соединений, что должна начаться беседа. Инициатор отвечает на пакет SYN-ACK пакетом ACK, подтверждая, что пакет SYN-ACK был получен. При получении пакета ACK получатель удаляет запись из таблицы, и начинается передача данных.

По крайней мере, именно так все и должно происходить. В случае с атакой SYN Flood, злоумышленник посылает на машину лавину пакетов SYN, часто подделывая при этом адрес источника. Ничего не подозревающая машина посылает по поддельным адресам пакеты SYN-ACK и добавляет запись в таблицу ожидающих соединений для каждого полученного пакета SYN. Эти записи остаются в таблице до тех пор, пока операционная система не признает их устаревшими по прошествии определенного времени. Если было послано достаточно много пакетов, таблица ожидающих запросов переполнится, и все попытки установить вполне законное соединение завершатся неудачей. А это приведет к появлению описанных мною симптомов и подобному выводу *netstat*.

Но в выводе команды *netstat* была одна аномалия, которая ставила под сомнение мой диагноз – разнообразие узлов в таблице. Возможно, что кто-то обладает программой с отличными способностями к подделкам, но обычно соединения устанавливаются с меньшего числа фальшивых узлов. Кроме того, многие из этих узлов казались настоящими. Ничего не прояснили и даже ухудшили ситуацию некоторые выполненные мною проверки. Попытки выполнить команды *ping* или *traceroute* для случайно выбранных узлов из списка, предоставленного командой *netstat*, иногда завершались успешно, а иногда нет. Мне не хватало данных. Надо было лучше разобраться с соединениями по этим удаленным узлам. Тут мне на помощь пришел Perl.

Поскольку я писал программу, находясь «под дулом пистолета», для работы с самой сложной частью проблемы у меня получился очень маленький сценарий, полагающийся на вывод двух внешних программ, анализирующих сеть и проделывающих наиболее сложную часть работы. Я покажу вам эту версию, используя ее в качестве трамплина для дальнейшего улучшения.

На этот раз задача свелась к одному вопросу: могу ли я добраться до узлов, пытающихся связаться со мной? Для поиска узлов, пытающихся связаться с моей машиной, я воспользовался программой *clog* Брайена Митчела (Brian Mitchell), которую можно найти на *ftp://coast.cs.purdue.edu/pub/mirrors/ftp.saturn.net/clog*. Для прослушивания сети в поисках запросов TCP-соединений, т. е. пакетов SYN, *clog* использует библиотеку *libpcap* от Lawrence Berkeley National Laboratory's Network Re-

search Group. Эту же библиотеку использует и эффективная программа наблюдения за сетью *tcpdump*. Библиотека *libpcap* с *ftp://ftp.ee.lbl.gov/libpcap.tar.Z* работает и для машин с Linux. Перенесенная версия *libpcap* для NT/2000 доступна на *http://netgroup-serv.polito.it/windump/* или *http://www.ntop.org/libpcap.html*, но хотелось бы также увидеть версию и для MacOS.

clog сообщает о пакетах SYN таким образом:

```
Mar 02 11:21|192.168.1.51|1074|192.168.1.104|113
Mar 02 11:21|192.168.1.51|1094|192.168.1.104|23
```

Из примера видно, что получено два запроса на соединение от машины 192.168.1.51 к 192.168.1.104. Первый – это попытка соединиться с портом 113 (ident), а второй – с портом 23 (telnet).

Программа *clog* помогла мне выяснить, какие узлы пытались установить соединение со мной. Но мне надо было знать, могу ли я до них добраться. Эта задача выпала на долю программы *fping* Роланда Дж. Шемерса III (Roland J. Schemers III). Программа *fping*, которую можно найти на *http://www.stanford.edu/~schemers/docs/fping/fping.html*, – это быстрая и шикарная версия программы *ping* для тестирования работоспособности сети в Unix и его вариантах. Воспользовавшись этими двумя внешними программами, получаем маленькую программу на Perl:

```
$clogex   = "/usr/local/bin/clog";      # местоположение/ключи для clog
$fpingex  = "/usr/local/bin/fping -r1"; # местоположение/ключи для fping

$localnet = "192.168.1";                # префикс локальной сети

open CLOG, "$clogex|" or die "Невозможно запустить clog:$!\n";
while(<CLOG>){
    ($date,$orighost,$origport,$desthost,$destport) = split(/\|/);
    next if ($orighost =~ /^$localnet/);
    next if (exists $cache{$orighost});
    print `$fpingex $orighost`;
    $cache{$orighost}=1;
}
```

Эта программа запускает команду *clog* и считывает ее вывод до бесконечности. Поскольку наша внутренняя сеть вне подозрений, каждый узел сравнивается с префиксом локальной сети и весь трафик из локальной сети игнорируется.

На этот раз, как и в последнем примере, мы используем некоторое кэширование. Мы – добропорядочные жители сети и не собираемся закидывать внешние машины множеством пакетов *ping*, так что мы следим за тем, к каким узлам мы уже обращались с запросами. Флаг *–r1* у *fping* служит для ограничения количества попыток обращения к узлу программой *fping* (по умолчанию предпринимается три попытки).

Эту программу необходимо выполнять с повышенными привилегиями, т. к. и программе *clog*, и программе *fping* нужен привилегированный доступ к сетевому интерфейсу компьютера. Вывод этой программы выглядит так:

```
199.174.175.99 is unreachable
128.148.157.143 is unreachable
204.241.60.5 is alive
199.2.26.116 is unreachable
199.172.62.5 is unreachable
130.111.39.100 is alive
207.70.7.25 is unreachable
198.214.63.11 is alive
129.186.1.10 is alive
```

Очевидно, что здесь творится нечто подозрительное. С чего вдруг половина узлов доступна, а половина – нет? Перед тем как ответить на этот вопрос, давайте посмотрим, что можно сделать для улучшения этой программы. Первый шаг – избавиться от зависимости от внешних программ. Умение прослушивать сеть и посылать пакеты *ping* из Perl открывает целый диапазон возможностей. Сначала позаботимся о том, чтобы удалить простую зависимость.

Модуль `Net::Ping` Рассела Мосмана (Russell Mosemann), который можно найти в дистрибутиве Perl, помогает проверить работоспособность сети. `Net::Ping` позволяет посылать пакеты *ping* трех типов и проверять возвращаемые ответы: ICMP, TCP и UDP. ICMP-пакеты (Internet Control Message Protocol) – это «классика *ping*», и их посылает подавляющее большинство программ, производных от *ping*. У пакетов этого типа есть два недостатка:

1. Как и в случае с программой, вызывающей *clog*/*fping*, все сценарии `Net::Ping`, использующие ICMP, необходимо выполнять с повышенными привилегиями.
2. Perl на MacOS в настоящее время не поддерживает ICMP. Возможно, в будущем это будет исправлено, а пока не следует забывать об этом ограничении переносимости.

Два других варианта пакетов `Net::Ping` – это пакеты TCP и UDP. В обоих случаях пакеты посылаются на порт службы *echo* удаленной машины. Эти две возможности обеспечивают переносимость, но вам они могут показаться менее надежными, чем ICMP. ICMP встроен во все стандартные стеки TCP/IP, но не на всех машинах может быть запущена служба *echo*. В результате, если ICMP не отфильтровывается намеренно, вы получите ответы на ICMP-пакеты с большей вероятностью, чем на пакеты других типов.

В `Net::Ping` применяется стандартная модель объектно-ориентированного программирования, поэтому первым делом нужно создать новый экземпляр объекта *ping*:

```
use Net::Ping;
$p = new Net::Ping("icmp");
```

Этот объект очень просто использовать:

```
if ($p->ping("host")){
    print "ping succeeded.\n";
else{
    print "ping failed\n";
}
```

Теперь вернемся к сложной части нашего первоначального сценария, прослушиванию сети с помощью *clog*. К сожалению, с этого момента нам придется отпустить пользователей MacOS. Программа на Perl, которую мы собираемся рассматривать, привязана к библиотеке *libpcap*, о которой мы говорили раньше, поэтому применение программы где-либо, кроме Unix и его вариаций, затруднено или невозможно.

Первый шаг, который необходимо выполнить, – собрать библиотеку *libpcap*. Я советую скомпилировать и *tcpdump*. Как и в случае с утилитами из командной строки для UCD-SNMP, *tcpdump* можно использовать для выяснения возможностей *libpcap* перед тем, как писать на Perl, а также для перепроверки кода, написанного для этой программы.

Имея *libpcap*, легко скомпилировать модуль `Net::Pcap`, первоначально написанный Питером Листером (Peter Lister), а затем полностью переписанный Тимом Поттером (Tim Potter). Этот модуль открывает вам доступ ко всем возможностям *libpcap*. Давайте посмотрим, как можно использовать его для поиска пакетов `SYN`, как это делает *clog*.

Программа начинается с запроса о доступном/допускающем прослушивание сетевом интерфейсе и его настройках:

```
use Net::Pcap;

# поиск сетевого устройства, допускающего прослушивание
$dev = Net::Pcap::lookupdev(\$err) ;
die "Невозможно найти подходящее устройство: $err\n" unless $dev;

# выясняем номер и маску сети этого устройства
die "Невозможно выяснить информацию об устройстве:$err\n"
  if (Net::Pcap::lookupnet($dev,\$netnum,\$netmask,\$err));
```

В большинстве функций *libpcap* действуют соглашения о кодах возврата, принятые в C, и возвращают `0` в случае успеха и `−1` в случае неудачи, поэтому в программах, использующих `Net::Pcap`, часто применяется идиома `die if ...`. В страницах руководства по *pcap(3)* можно выяснить смысл аргументов, передаваемых каждой функции.

Получив информацию о сетевом интерфейсе, можно сообщить *libpcap* о намерении прослушивать сеть (вместо того, чтобы читать пакеты из

сохраненного ранее файла пакетов). `Net::Pcap::open_live` возвращает дескриптор, используемый для ссылки на этот сеанс:

```
# открываем интерфейс для "живого" захвата пакетов
$descript = Net::Pcap::open_live($dev, 100, 1, 1000, \$err) ;
die "Невозможно получить дескриптор:$err\n" unless $descript;
```

Программа *libpcap* позволяет перехватывать как весь сетевой трафик, так и ограниченное подмножество пакетов, выбираемое в соответствии с заданным критерием фильтрации. Ее фильтрующий механизм очень эффективен, поэтому зачастую лучше напрямую вызвать его, чем потом в программе отсеивать ненужные пакеты. В данном случае нас интересуют только пакеты `SYN`.

Что собой представляет пакет `SYN`? Чтобы понять это, нужно иметь представление о том, как собираются TCP-пакеты. Посмотрите на рисунок из RFC973, где приведен TCP-пакет и его заголовок (рис. 10.2).

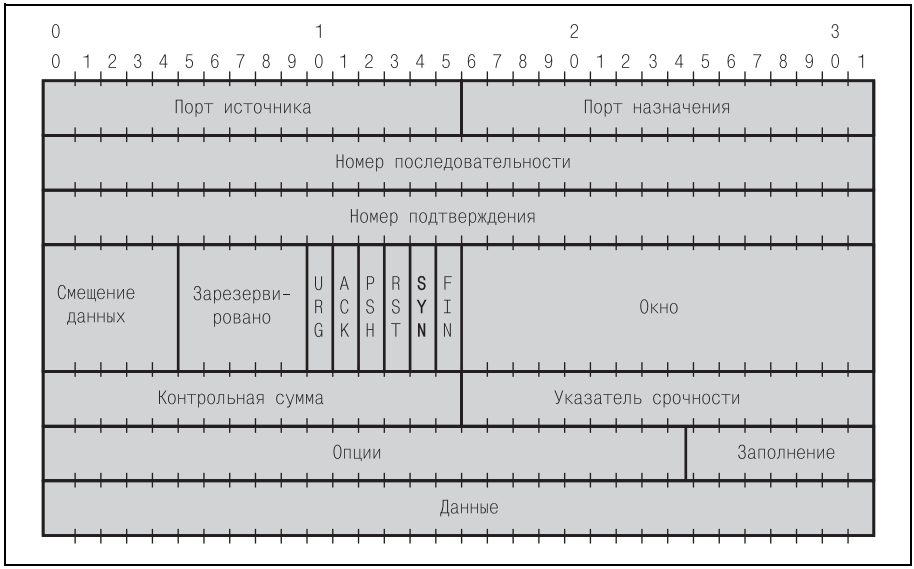

***Рис. 10.2.** Схема TCP-пакета*

Пакет `SYN` – это тот пакет, в заголовке которого установлен только флаг `SYN` (он выделен жирным шрифтом на рис. 10.2). Для того чтобы *libpcap* «знала», что надо перехватывать только такие пакеты, следует определить, какой именно байт в пакете она должна искать. Каждый штрих наверху соответствует одному биту, так что нетрудно подсчитать байты. Тот же пакет, но уже с номерами байтов, изображен на рис. 10.3.

Нам необходимо проверить, равен ли 13-й байт двоичному числу 00000010 (десятичное число 2). В качестве фильтра нам нужна строка `tcp[13] = 2`. Если бы мы хотели найти пакеты, у которых установлен

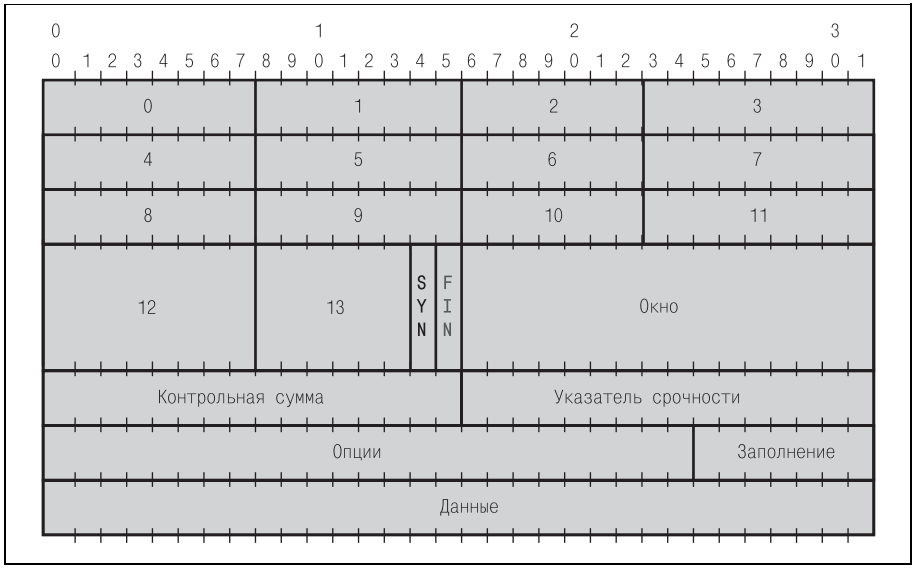

Рис. 10.3. *Ищем нужный байт в TCP-пакете*

по крайней мере флаг SYN, то могли бы использовать строку `tcp[13] & 2 != 0`. Этот фильтр затем компилируется в программу фильтрации и устанавливается:

```
$prog = "tcp[13] = 2";

# компилируем и устанавливаем "программу фильтрации"
die "Невозможно скомпилировать $prog\n"
  if (Net::Pcap::compile($descript ,\$compprog,$prog,0,$netmask)) ;
die "Невозможно установить фильтр\n"
  if (Net::Pcap::setfilter($descript,$compprog));
```

Еще чуть-чуть и можно запускать *libpcap*. Но перед этим нужно определить, что делать с найденными пакетами. Для каждого пакета, соответствующего фильтру, она может по нашему выбору запустить подпрограмму обратного вызова. Этой подпрограмме передаются три аргумента:

1. Пользовательский строковый идентификатор, который может устанавливаться в начале перехвата и позволяет процедуре различать несколько сеансов перехвата пакетов.
2. Ссылка на хэш, описывающая заголовок пакета (отметки о времени и пр.).
3. Копия всего пакета.

Начнем мы с очень маленькой процедуры, выводящей длину полученного пакета:

```
sub printpacketlength {
  print length($_[2]),"\n";
}
```

Имея нужную подпрограмму, мы ищем пакеты SYN:

```
die "Невозможно перехватить пакеты: ".Net::Pcap::geterr($descript)."\n"
  if (Net::Pcap::loop($descript,-1,\&printpacketlength, ''));

die "Невозможно закрыть устройство\n"
  if (Net::Pcap::close($descript));
```

Второй аргумент -1 метода Net::Pcap::loop()определяет количество пакетов, которые мы хотим перехватить до выхода. В данном случае мы будем перехватывать пакеты до бесконечности.

Приведенная выше программа перехватывает пакеты SYN и выводит их длину, но это не совсем то, чего мы хотели добиться в начале этого раздела. Нам нужна программа, которая будет искать пакеты SYN из других сетей и попытается «прощупать» (ping) источник. У нас есть практически все; пока не хватает только способа, позвляющего определить источник полученного SYN-пакета.

Как и в примере из главы 5, работающем с DNS, нам придется разбить пакет на части. Обычно такая процедура требует обращения к спецификации (RFC) и создания нужных шаблонов unpack(). Тим Поттер (Tim Potter) проделал сложную работу и написал несколько модулей NetPacket: NetPacket::Ethernet, NetPacket::IP, NetPacket::TCP, NetPacket::ICMP и т. д. Каждый из них поддерживает два метода: strip() и decode().

Метод strip() просто возвращает данные из пакета, выкидывая все, что касается уровня сети. Запомните, что TCP/IP-пакет в сети Ethernet – это, на самом деле, обычный пакет TCP, «обернутый» в пакет IP, а тот, в свою очередь, обернут в пакет Ethernet. Так что если $pkt хранит TCP/IP-пакет, то NetPacket::Ethernet::strip($pkt) вернет IP-пакет (удалив уровень Ethernet). Если бы нам нужна была TCP-часть от $pkt, можно было бы использовать NetPacket::IP::strip(NetPacket::Ethernet::strip($packet)) для удаления и IP-, и Ethernet-уровня.

decode() продвигается глубже еще на один шаг. Он разбивает пакет на его составляющие и возвращает экземпляр объекта, содержащего все эти части. Например:

```
NetPacket::TCP->decode(
   NetPacket::IP::strip(NetPacket::Ethernet::strip($packet)))
```

вернет экземпляр объекта со следующими полями:

Поле	Описание
src_port	TCP-порт источника
dest_port	TCP-порт приемника

Поле	Описание
Seqnum	Порядковый номер
Acknum	Номер подтверждения
Hlen	Длина заголовка
Reserved	6-битное «зарезервированное» пространство в TCP-заголовке
Flags	Флаги URG, ACK, PSH, RST, SYN и FIN
Winsize	Размер TCP-окна
Cksum	Контрольная сумма
Urg	Указатель на экстренные данные
Options	Любые TCP-параметры в двоичном виде
Data	Данные для пакета

Это уже должно быть знакомо читателю (рис. 10.2). Чтобы выяснить порт приемника для пакета, можно сделать следующее:

```
$pt = NetPacket::TCP->decode(
         NetPacket::IP::strip(
             NetPacket::Ethernet::strip($packet)))->{dest_port};
```

Теперь соберем все вместе и кое-что изменим. Поттер создал оболочку для инициализации и циклов `Net::Pcap` и выпустил ее как модуль `Net::PcapUtils`. Модуль обрабатывает некоторые из выполняемых нами шагов, делая наши программы короче. Продемонстрируем все это в действии, учитывая все, что мы узнали в последнем разделе:

```
use Net::PcapUtils;

use NetPacket::Ethernet;
use NetPacket::IP;

use Net::Ping;

# локальная сеть
$localnet = "192.168.1";
# фильтр для поиска SYN-пакетов не из локальной сети
$prog = "tcp[13] = 2 and src net not $localnet";

$| = 1; # снимаем буферизацию с STDIO

# создаем объект ping, который будем использовать позже
$p = new Net::Ping("icmp");

die "Невозможно перехватить пакеты:".Net::Pcap::geterr($descript)."\n"
  if (Net::PcapUtils::open_live(\&grab_ip_and_ping, FILTER => $prog));

# ищем IP-адрес источника пакета, пингуем его (один раз на
# каждый запуск)
```

```
sub grab_ip_and_ping{
    my ($arg,$hdr,$pkt) = @_ ;

    # получаем IP-адрес источника
    $src_ip = NetPacket::IP->decode(
                NetPacket::Ethernet::strip($pkt))->{src_ip};

    print "$src_ip is ".(($p->ping($src_ip)) ?
                         "alive" : "unreachable")."\n"
      unless $cache{$src_ip}++;
}
```

Теперь, когда мы достигли своей цели и написали целиком на Perl (хоть и с помощью некоторых модулей, являющихся Perl-оболочками для программ на C) программу, позвольте мне рассказать, чем закончилась эта история.

В воскресенье утром центральная группа поддержки из другого отдела нашла ошибку в настройках маршрутизатора. Студент в одном из общежитий установил на свою машину Linux и неверно настроил демон маршрутизации сети. Эта машина посылала широковещательные сообщения в университетскую сеть о том, что она является маршрутом по умолчанию. Неправильно настроенный маршрутизатор, обслуживающий наш отдел, услышав это сообщение, безропотно изменил таблицу маршрутизации и добавил второй маршрут во внешний мир. Пакеты будут приходить из внешнего мира, и этот маршрутизатор, честно выполняя свои обязанности, поровну разделит ответы между двумя маршрутами. Такое распределение, когда «один пакет посылается настоящему маршрутизатору, другой посылается на машину студента, один – настоящему маршрутизатору, другой – на машину студента», создало асимметричную ситуацию маршрутизации. Когда фиктивный маршрут был удален и были задействованы фильтры, предотвращающие появление такой ситуации в дальнейшем, наша жизнь вернулась к нормальному ритму. Я не буду говорить, что стало со студентом, вызвавшим эту проблему.

В этом разделе вы познакомились с применением модулей `Net::Pcap`, `Net::PcapUtils` и семейства модулей `NetPacket::*` для диагностики. Не останавливайтесь на этом! Эти модули позволяют написать множество программ, способных помочь разобраться с проблемами в сети или активно наблюдать за сетью в поисках опасности.

Предотвращение подозрительных действий

Самый последний атрибут ночного сторожа, который нас интересует, – это способность предотвращать. Это тот голос, который подсказывает: «Не стоит ставить на подоконник только что испеченный пирог, чтобы охладить его».

Мы завершим эту главу примером, разумное применение которого может положительным образом повлиять на одну машину или целую инфраструктуру. В качестве символического жеста, завершающего эту книгу, мы напишем собственный модуль, вместо того чтобы показывать, как пользоваться чужими.

Цель, которую я преследую, состоит в предотвращении использования или хотя бы в уменьшении количества плохих паролей. Хорошие механизмы защиты становятся бесполезными из-за выбора плохих паролей. Паролем Ога для возвращения в пещеру клана, скорее всего, было слово «ог». В наше время ситуация обостряется из-за повсеместного наплыва хитроумных программ для взлома паролей, таких как *John the Ripper* (Solar Designer, Солар Дизайнер), *L0phtCrack* (Mudge, Мадж и Weld Pond, Вельд Понд) и *Crack* (Alec Muffett, Алек Маффет)).

Единственный способ избежать уязвимости, которой подвержены ваши системы благодаря этим программам, – избавиться от плохих паролей. Вам нужно помочь пользователям получить пароли, которые сложно отгадать. Один из способов сделать это в Unix (хотя эту программу можно перенести и на NT, и на MacOS) – использовать *libcrack*, также написанную Алеком Маффетом. В процессе написания программы *Crack* Маффет оказал огромную услугу системным администраторам, взяв несколько методов, используемых в *Crack*, и создав из них библиотеку проверки паролей, написанную на C.

В библиотеке имеется лишь одна функция для пользовательского интерфейса: `FascistCheck()`. Эта функция принимает два аргумента: строку для проверки и полный префикс пути для файла словаря, созданного при установке *libcrack*. Функция возвращает либо `NULL`, если строка является «безопасным» паролем, либо объяснение, например, «это словарное слово», если пароль легко взломать. Было бы очень удобно, если бы существовала возможность использовать эту функциональность как часть программы на Perl, устанавливающей или меняющей пароли[1], так что давайте посмотрим, как можно написать модуль, применяя эту функцию. Нам потребуется заглянуть ненадолго в программу на C, но я обещаю, что это не займет много времени и пройдет безболезненно.

Первый наш шаг – собрать пакет *libcrack* с *http://www.users.dircon.co.uk/~crypto/*. Этот процесс подробно описан в дистрибутиве и не вызывает затруднений. Я лишь приведу пару советов:

- Чем больше будет словарь, который вы соберете, тем лучше. Хороший источник для слов, которые можно включить в словарь, – это *ftp://ftp.ox.ac.uk/pub/wordlists*. Процесс сборки требует значитель-

[1] Похожий пример, в котором *libcrack* применяется во благо, это *npasswd* (можно найти на *http://www.utexas.edu/cc/unix/software/npasswd/*), отличная замена программе смены паролей в Unix *passwd*, написанная Клайдом Гувером (Clyde Hoover).

ного дискового пространства (для процесса *sort* в *utils/mkdict*), имейте это в виду.

- Убедитесь, что вы собираете *libcrack* при помощи тех же средств разработки, что и Perl. Например, если при компиляции Perl вы пользовались *gcc*, обязательно используйте *gcc* и при компиляции *libcrack*. Это справедливо для всех модулей, которые нужно связать с дополнительными библиотеками C.

После того как библиотека C *libcrack.a* собрана, необходимо выбрать метод для вызова функции `FascistCheck()` из Perl. Для создания подобной связи существует два популярных метода: XS и SWIG. Мы будем применять XS, т. к. его легко использовать для простых задач, и все необходимые для этого инструменты входят в состав дистрибутива Perl. Подробное сравнение этих двух методов можно найти в книге «Advanced Perl Programming» (Углубленное программирование на Perl) Шрирама Шринивасана (Sriram Srinivasan) (O'Reilly).

Самый простой способ начать работать с XS – использовать программу *h2xs* для создания прототипа модуля:

```
$ h2xs -A -n Cracklib
Writing Cracklib/Cracklib.pm
Writing Cracklib/Cracklib.xs
Writing Cracklib/Makefile.PL
Writing Cracklib/test.pl
Writing Cracklib/Changes
Writing Cracklib/MANIFEST
```

Вот описание файлов, создаваемых этой командой (табл. 10.2).

Таблица 10.2. Файлы, созданные командой h2xs -A -n Cracklib

Имя файла	Описание
Cracklib/Cracklib.pm	Заглушка с прототипами и документацией
Cracklib/Cracklib.xs	Склейка с кодом на C
Cracklib/Makefile.PL	Код на Perl для создания файла Makefile
Cracklib/test.pl	Тестовый код прототипа
Cracklib/Changes	Документирование версий
Cracklib/MANIFEST	Список файлов, входящих в состав модуля

Чтобы получить нужную нам функциональность, следует изменить два файла. Начнем с более сложного: склейки с кодом на C. Вот как эта функция определяется в документации *libcrack*:

```
char *FascistCheck(char *pw, char *dictpath);
```

В нашем файле *Cracklib/Cracklib.xs* мы повторим это определение:

```
PROTOTYPES: ENABLE
```

```
char *
FascistCheck(pw,dictpath)
      char *pw
      char *dictpath
```

Директива PROTOTYPES создает Perl-прототипы для функций из этого файла. В программе, которую мы пишем, это не имеет значения, но мы включаем директиву для подавления предупреждений в процессе сборки.

Сразу же после определения функции мы описываем, как она вызывается и что возвращает:

```
      CODE:
      RETVAL = (char *)FascistCheck(pw,dictpath);
      OUTPUT:
      RETVAL
```

RETVAL – это настоящая склейка. Она представляет собой точку передачи между кодом на C и интерпретатором Perl. Именно тут мы говорим Perl, что он должен получить строку символов, возвращенную библиотечной функцией FascistCheck(), и сделать их доступными в качестве возвращаемого значения (т. е. OUTPUT) Perl-функции Cracklib::FascistCheck(). Больше нам не придется иметь дело с кодом на C.

В другом файле, который нужно поменять, мы изменим только одну строку. Нам требуется добавить еще один аргумент в вызов WriteMakefile() в *Makefile.PL*, чтобы убедиться, что Perl может найти файл *libcrack.a*. Вот как выглядит эта новая строка в нашем контексте:

```
'LIBS'      => [''],   # например, '-lm'
'MYEXTLIB' => '/usr/local/lib/libcrack$(LIB_EXT)' # местоположение cracklib
'DEFINE'    => '',     # например, '-DHAVE_SOMETHING'
```

Это тот минимум, который необходим для работы модуля. Если мы наберем:

```
perl Makefile.PL
make
make install
```

то сможем начать использовать наш модуль примерно так:

```
use Cracklib;
use Term::ReadKey;     # для чтения паролей
$dictpath = "/usr/local/etc/cracklib/pw_dict";

print "Введите пароль: ";
ReadMode 2;            # отключаем вывод символов на экран
chomp($pw = ReadLine);# читаем пароль
ReadMode 0;            # возвращаем терминал в предыдущее состояние
print "\n";
```

```
$result = Cracklib::FascistCheck($pw,$dictpath);
if (defined $result){
    print "Пароль не подходит, потому что $result.\n";
}
else {
    print "Пароль подходит, спасибо!\n";
}
```

Но не стоит использовать этот модуль в таком виде. Давайте, перед тем как устанавливать модуль, доведем его до профессионального уровня.

Во-первых, добавим сценарий, позволяющий удостовериться, что модуль работает корректно. Сценарий должен вызывать нашу функцию с некоторыми известными значениями и сообщать каким-нибудь специфичным образом, получил ли он правильные ответы. В самом начале проверки нужно напечатать диапазон номеров тестов. Например, если мы собираемся провести 10 тестов, нужно сначала напечатать 1..10. Затем для каждого выполняемого теста следует напечатать либо «ok», либо «not ok» и номер теста. Стандартная программа сборки модуля интерпретирует этот вывод и выводит пользователю итоги результатов проверки.

h2xs предоставляет пример сценария проверки, который можно изменять. Создадим каталог *t* (стандартный каталог, назначенный по умолчанию для проверки модуля) и переименуем *test.pl* в *t/cracklib.t*. Вот фрагмент кода на Perl, который нужно добавить в конец *t/cracklib.t* для выполнения ряда тестов:

```
# местоположение файлов словарей
$dictpath = "/usr/local/etc/pw_dict";

# проверочные строки и известные для них ответы cracklib
%test =
  ("happy"        => "it is too short",
   "a"            => "it's WAY too short",
   "asdfasdf"     => "it does not contain enough DIFFERENT characters",
   "minicomputer" => "it is based on a dictionary word",
   "1ftm2tgr3fts" => "");

# Просматриваем в цикле все ключи из хэша, проверяя, возвращает
# ли cracklib предполагаемые ответы. Если да, то пишем "ok", в
# противном случае -- "not ok"
$testnum = 2;
foreach $pw (keys %test){
    my ($result) = Cracklib::FascistCheck($pw,$dictpath);
    if ((defined $result and $result eq $test{$pw}) or
       (!defined $result and $test{$pw} eq "")){
          print "ok ",$testnum++,"\n";
    }
    else {
        print "not ok ",$testnum++,"\n";
```

```
    }
}
```

Всего было сделано шесть тестов (пять из хэша `%test` и проверка загрузки модуля), значит, нужно изменить строку из *t/cracklib.t* с:

```
BEGIN { $| = 1; print "1..1\n"; }
```

на:

```
BEGIN { $| = 1; print "1..6\n"; }
```

Теперь можно набрать *make test* и *Makefile* и запустить программу проверки, чтобы убедиться, что модуль работает верно.

Разумеется, сценарий проверки очень важен, но наш сценарий вряд ли заслужит уважение, если мы пропустим такой решающий компонент, как документацию. Потратьте время и дополните файлы *Cracklib.pm* и *Changes*, заменив заглушки на полезную информацию о модуле. Также неплохо добавить файл *README* или *INSTALL*, в котором рассказано, как собрать модуль, где найти нужные компоненты, такие как *libcrack*, приведены примеры программ и т. д. Об этих новых файлах и переименовании файла *test.pl* нужно сказать в файле *MANIFEST*, чтобы не вводить в заблуждение программу компиляции модуля.

Наконец, установите модуль там, где нужно. Используйте вызовы `Cracklib::FascistCheck()` везде, где нужно установить или сменить пароли. Если количество плохих паролей в вашей системе уменьшится, «ночной сторож» с удовольствием одобрит вас.

Информация о модулях из этой главы

Модуль	Идентификатор на CPAN	Версия
`Getopt::Std` (входит в состав Perl)		1.01
`Digest::MD5`	GAAS	2.09
`Net::DNS`	MFUHR	0.12
`FreezeThaw`	ILYAZ	0.3
`File::Find` (входит в состав Perl)		
`Net::SNMP`	DTOWN	3.01
`SNMP`	GSM	3.10
`Net::Ping` (входит в состав Perl)	RMOSE	2.02
`Net::Pcap`	TIMPOTTER	0.03
`Net::PcapUtils`	TIMPOTTER	0.01
`NetPacket`	TIMPOTTER	0.01
`Term::ReadKey`	KJALB	2.14

Рекомендуемая дополнительная литература

Инструменты для обнаружения изменений

http://www.securityfocus.com – один из лучших сайтов в Сети, посвященных безопасности. Помимо того что здесь поддерживаются несколько лучших списков рассылки по вопросам безопасности, на этом сайте есть превосходная библиотека бесплатных инструментов. В разделе «auditing, file integrity» этой библиотеки можно найти много *tripwire*-подобных инструментов.

«MacPerl:Power and Ease», Викки Браун (Vicki Brown) и Крис Нандор (Chris Nandor) (Prime Time Freeware, 1998). Эта книга и страницы руководства по *perlport* были основными источниками информации по таблице `stat()` в первом разделе данной главы.

«RFC1321:The MD5 Message-Digest Algorithm», R. Rivest, 1992.

http://www.tripwire.com/tripwire – бесплатная утилита для обнаружения изменений файловой системы. Недавно утилита стала коммерческой, но старые версии по-прежнему доступны свободно.

SNMP

Существует примерно 60 документов RFC со словом SNMP в своих названиях (и около 100, в которых SNMP лишь упоминается). Здесь приведен список только тех документов RFC, на которые мы ссылались в этой главе или в приложении E.

«RFC1157:A Simple Network Management Protocol (SNMP)», J. Case, M. Fedor, M. Schoffstall, J. Davin, 1990.

«RFC1213:Management Information Base for Network Management of TCP/IP-based internets:MIB-II», K. McCloghrie, M. Rose, 1991.

«RFC1493:Definitions of Managed Objects for Bridges», E. Decker, P. Langille, A. Rijsinghani, K. McCloghrie, 1993.

«RFC1573:Evolution of the Interfaces Group of MIB-II», K. McCloghrie, F. Kastenholz, 1994.

«RFC1905:Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)», J. Case, K. McCloghrie, M. Rose, S. Waldbusser, 1996.

«RFC1907:Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)», J. Case, K. McCloghrie, M. Rose, S. Waldbusser, 1996.

«RFC2011:SNMPv2 Management Information Base for the Internet Protocol using SMIv2», K. McCloghrie, 1996.

«RFC2012:SNMPv2 Management Information Base for the Transmission Control Protocol using SMIv2», K. McCloghrie, 1996.

«RFC2013:SNMPv2 Management Information Base for the User Datagram Protocol using SMIv2», K. McCloghrie, 1996.

«RFC2274:User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)», U. Blumenthal, B. Wijnen, 1998.

«RFC2275:View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP)», B. Wijnen, R. Presuhn, K. McCloghrie, 1998.

«RFC2578:Structure of Management Information Version 2 (SMIv2)», K. McCloghrie, D. Perkins, J. Schoenwaelder, 1999.

Вот несколько хороших обычных источников информации о SNMP:

http://ucd-snmp.ucdavis.edu – домашняя страница проекта UCD-SNMP.

http://www.cisco.com/public/sw-center/netmgmt/cmtk/mibs.shtml – место нахождения MIB-файлов от Cisco. У других производителей тоже есть подобные сайты.

http://www.snmpinfo.com – домашняя страница компании SNMPinfo и Дэвида Перкинса («гуру» SNMP, активно пишущий в *comp.protocols.snmp*, и один из авторов *Understanding SNMP MIBs*).

http://www.ibr.cs.tu-bs.de/ietf/snmpv3/ – отличный источник информации по SNMP Version 3.

http://www.mrtg.org и *http://www.munitions.com/~jra/cricket/* – домашние страницы Multi Router Traffic Grapher (MRTG) и его потомка Cricket (написан на Perl!). Это два хороших примера использования SNMP для длительного наблюдения за устройствами.

«Understanding SNMP MIBs», David Perkins, Evan McGinnis (Prentice-Hall, 1996).

http://www.snmp.org – домашняя страница компании SNMP Research. В разделе «SNMP Framework» этого сайта приведено много ссылок на хорошие источники, включая список часто задаваемых вопросов *comp.protocols.snmp*.

Другие ресурсы

«Advanced Perl Programming», Sriram Srinivasan (O'Reilly, 1997). В книге есть хороший раздел о создании модулей.

http://www.bb4.com и http://www.kernel.org/software/mon/ – домашние страницы BigBrother и Mon, два удачных примера пакетов, обеспечивающих общий интерфейс для наблюдения за событиями в реальном времени (в отличие от наблюдения за уже свершившимся в MRTG и Cricket).

http://www.tcpdump.org – домашняя страница *libpcap* и *tcpdump*.

«RFC793:Transmission Control Protocol», J. Postel, 1981.

A

Пятиминутное руководство по RCS

Это короткое руководство научит вас всему, что нужно знать о применении системы контроля версий (Revision Control System, RCS) для системного администрирования. Если вы собираетесь серьезно использовать RCS, то обязательно загляните в страницы руководств и источники информации, ссылки на которые приведены в конце приложения, поскольку здесь рассматривается лишь минимум возможностей.

RCS-функции сродни агентству по найму автомобилей. Только один человек может арендовать конкретную машину в настоящий момент времени. Новую машину можно арендовать только после того, как агентство добавит ее в свой парк. Покупатели могут просматривать список машин (и их возможностей) в любое время, но если два человека хотят арендовать одну и ту же машину, второй должен подождать, пока машина будет возвращена в агентство. Наконец, агенты по найму тщательно изучают каждую машину после того, как она возвращается в фирму, и записывают все изменения, произошедшие с ней за то время, пока она была арендована. Все это справедливо и для RCS.

В RCS файл – как машина. Для того чтобы следить за файлом, используя RCS (т. е. добавить его к арендуемым машинам), необходимо сначала включить его (check in) в репозиторий:

```
$ ci -u filename
```

Команда *ci* – это сокращение от «check in», а ключ *–u* указывает на то, что файл необходимо оставить там, где он находится на время добавления. Когда файл добавлен (т. е. доступен для аренды), RCS делает одно из двух, чтобы напомнить пользователю о том, что файл находится под контролем RCS:

1. Удаляет первоначальный файл, оставляя только файл в архиве RCS. Обычно этот файл носит название *filename,v* и хранится либо в

том же каталоге, что и оригинальный файл, либо в подкаталоге *RCS* (если пользователь создает его).

2. Если задан ключ *–u*, как в рассмотренном выше случае, файл возвращается обратно в рабочий каталог и доступен только для чтения.

Для редактирования файла, находящегося под контролем RCS (иными словами, взять машину в аренду), необходимо сначала извлечь его (check-out) из репозитория:

```
$ co -l filename
```

Ключ `-l` указывает RCS «жестко блокировать» (to «strictly lock») файл (т. е. другие пользователи не могут в это время извлечь его из репозитория). С командой *co* используются также ключи:

- *–r<revision number>:* для извлечения более старых версий файла.
- *–p:* для вывода последней версии на экран, при этом извлечения файла не происходит.

После того как работа с файлом закончена, его необходимо вернуть обратно в репозиторий посредством той же команды, что и ранее, когда он помещался в репозиторий первый раз (*ci –u filename*). При помещении файла в репозиторий все внесенные в него изменения сохраняются эффективным с точки зрения расходования пространства способом.

Каждый раз, когда измененный файл помещается в репозиторий, ему присваивается новый номер «ревизии». Направляя файл в репозиторий, RCS запрашивает у пользователя комментарий, который записывается в журнал изменений, автоматически поддерживаемый для каждого файла. С помощью команды *rlog filename* можно просмотреть этот журнал, а также узнать, кто в настоящее время работает с извлеченным файлом.

Если кто-то не поместит измененный файл обратно в RCS (например, уйдет домой, а у вас возникнет реальная необходимость внести изменения в данный файл), вы можете снять блокировку при помощи *rcs-u filename*. Эта команда запросит у вас сообщение, которое отошлет тому, кто устанавливал блокировку.

После снятия блокировки необходимо проверить, чем текущая версия отличается от того, что хранится в RCS. Эту информацию предоставляет команда *rcsdiff filename*. Для того чтобы сохранить эти изменения, поместите файл в репозиторий (с соответствующим комментарием), а затем извлеките его еще раз и только потом начинайте с ним работать. *rcsdiff*, как и упомянутая выше *co*, может принимать ключ *–r<revision number>* для сравнения двух предыдущих версий.

Обратите внимание на некоторые операции, выполняемые в RCS, и соответствующие им команды (табл. А.1).

Таблица А. 1. Распространенные операции в RCS

Операция	Команда
Первое помещение файла в репозиторий (файл остается активным в файловой системе)	*ci -u filename*
Извлечение с блокировкой	*co -l filename*
Помещение файла и снятие блокировки (файл остается активным в файловой системе)	*ci -u filename*
Вывести версию *x.y* файла	*co -px.y filename*
Вернуться к версии *x.y* (перезаписывает активный файл файлом указанной версии)	*co -rx.y filename*
Сравнение активного файла и последней версии	*rcsdiff filename*
Сравнение версий *x.y* и *x.z*	*rcsdiff -rx.y -rx.z filename*
Просмотр журнала	*rlog filename*
Снятие блокировки с файла, установленной другим пользователем	*rcs -u filename*

Хотите верьте, хотите нет, но это все, что вам нужно, чтобы начать работать с RCS. Как только вы начнете использовать RCS в системном администрировании, вы поймете, что это щедро окупается.

Ссылки на подробную информацию

ftp://ftp.gnu.org/pub/gnu/rcs – здесь можно найти самый последний исходный код пакета RCS.

«Applying RCS and SCCS: From Source Control to Project Control», Don Bolinger, Tan Bronson (O'Reilly, 1995).

http://www.sourcegear.com/CVS – сюда стоит обратиться, если вам нужны возможности, которых нет в RCS. Очень распространенная система CVS (Concurrent Version System) – это следующий шаг наверх. А это его основной сайт.

B

Десятиминутное руководство по LDAP

LDAP (Lightweight Directory Access Protocol, облегченный протокол доступа к сетям) – это одна из самых значительных служб каталогов, существующих в настоящее время. Вероятно, со временем системные администраторы будут работать с LDAP-серверами и клиентами в различном контексте. Это руководство представляет собой введение в номенклатуру LDAP и концепции, необходимые для использования материала из главы 6 «Службы каталогов».

Все действия в LDAP производятся над структурой данных, называемой *элементом* (*entry*). Схему, показанную на рис. B.1, имеет смысл хорошо себе представлять, когда будут рассматриваться составляющие элемента.

Элемент состоит из нескольких компонентов, называемых *атрибутами* (*attributes*), в которых хранятся данные для этого элемента. В терминах баз данных они похожи на поля записи. В главе 6 Perl используется для хранения списка машин из каталога LDAP. У каждого элемента, соответствующего машине, есть такие атрибуты, как имя, модель, местоположение, владелец и т. д.

Помимо имени атрибут состоит из *типа* (*type*) и набора *значений* (*values*), соответствующих этому типу. Если вы храните информацию о сотрудниках, то у элемента может быть атрибут `phone` (телефон) типа `telephoneNumber`. Значениями этого атрибута будут номера телефонов сотрудников. Тип имеет также *синтаксис*, определяющий, какие данные можно использовать (строки, числа и т. д.), как они сортируются и как их применять при поиске (чувствительность к регистру).

У каждого элемента есть специальный атрибут *objectClass*, содержащий несколько значений, которые вместе с настройками сервера и пользовательскими настройками определяют, какие атрибуты должны и могут существовать для этого определенного элемента.

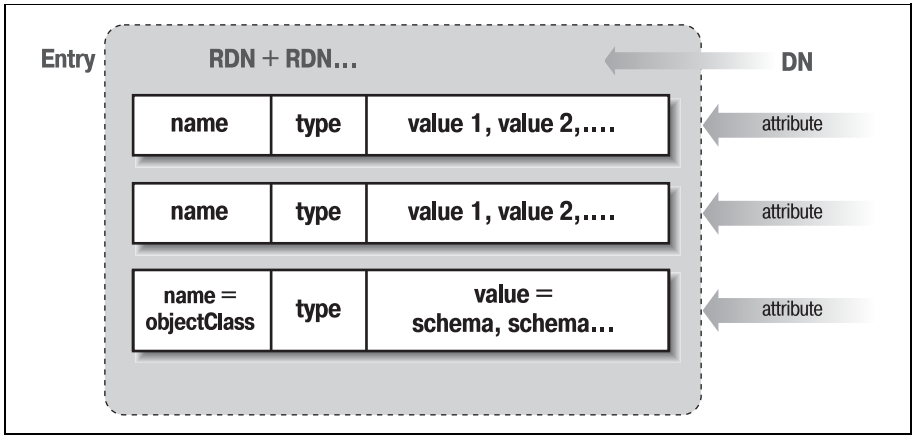

Рис. B.1. *Структура данных элемента LDAP*

Рассмотрим детальнее атрибут *objectClass*, поскольку он иллюстрирует некоторые важные качества LDAP и позволяет избавиться от жаргона, с которым мы раньше не встречались. Рассматривая атрибут *objectClass*, нужно обратить внимание на следующее:

LDAP объектно-ориентирован

Каждое значение атрибута *objectClass* является именем класса объекта. Эти классы либо определяют набор атрибутов, которые могут или обязаны быть в элементе, либо расширяют определения, унаследованные из другого класса.

Вот пример: атрибут *objectClass* элемента может содержать строку `residentialPerson`. В RFC2256 с устрашающим названием «A Summary of the X. 500(96) User Schema for use with LDAPv3» класс `residentialPerson` определяется так:

```
residentialPerson
   ( 2.5.6.10 NAME 'residentialPerson' SUP person STRUCTURAL MUST l
     MAY ( businessCategory $ x121Address $ registeredAddress $
     destinationIndicator $ preferredDeliveryMethod $ telexNumber $
     teletexTerminalIdentifier $ telephoneNumber $
     internationaliSDNNumber $
     facsimileTelephoneNumber $ preferredDeliveryMethod $ street $
     postOfficeBox $ postalCode $ postalAddress $
     physicalDeliveryOfficeName $ st $ l ) )
```

В определении сказано, что элемент класса `residentialPerson` должен иметь атрибут `l` (сокращение от locality) и может иметь целый набор других атрибутов (`registeredAddress`, `postOfficeBox` и т. д.). Ключевая часть спецификации – это строка `SUP person`. Это означает, что родительским классом (тем, от которого `residentialPerson`

наследует *свои* атрибуты) является класс person. Данное определение выглядит так:

```
person
   ( 2.5.6.6 NAME 'person' SUP top STRUCTURAL MUST ( sn $ cn )
     MAY ( userPassword $ telephoneNumber $ seeAlso $ description ) )
```

Значит, элемент класса residentialPerson должен иметь атрибуты sn (surname – фамилия), cn (common name – имя) и l (locality – местоположение) и может иметь атрибуты, перечисленные в разделах MAY из этих двух отрывков из RFC. Кроме того, известно, что person – это вершина иерархии объектов для residentialPerson, поскольку его родительским классом является специальный абстрактный класс top.

В большинстве случаев можно выйти из положения, если использовать предопределенные стандартные классы объектов. Для того чтобы создать элементы с атрибутами, которых нельзя найти в существующем классе, целесообразно найти ближайший класс и все построить на нем, как в случае с residentialPerson, построенном на person.

LDAP происходит из мира баз данных

Второе качество, которое можно увидеть в *objectClass*, – это корни LDAP, уходящие в базы данных. Набор классов объектов, определяющих атрибуты элементов, на сервере LDAP называются *схемой* (*schema*). RFC, процитированный выше, – это один пример спецификации схемы LDAP. В данной книге мы не будем касаться того, что имеет отношение к схеме. Как и в случае с проектированием базы данных, проектированию схемы можно посвятить целую книгу, но вы должны быть, по крайней мере, знакомы с термином «схема», т. к. он всплывет позже.

LDAP не ограничен хранением информации строго в структуре дерева

Последнее, что нужно сказать об *objectClass* для того, чтобы перейти от рассмотрения одного элемента к более общей картине, относится к следующему: на вершине иерархии объектов в предыдущем примере находился класс top, но существует еще один квази-суперкласс, заслуживающий упоминания, класс alias. Если alias указан, то этот элемент действительно является псевдонимом для другого элемента (определяемого атрибутом aliasedObjectName данного элемента). LDAP поощряет иерархические структуры в виде дерева, но он их не требует. Очень важно об этом помнить при написании программ, чтобы не сделать неверных предположений относительно иерархии данных на сервере.

Организация данных в LDAP

До сих пор мы говорили только об одном элементе, но спрос на каталоги, содержащие только один элемент, очень мал. Как только мы станем рассматривать каталоги, содержащие много элементов, перед нами тотчас встанет вопрос, с которого начиналось данное приложение: как найти что-либо?

Все, что обсуждалось до этого момента, подпадает под определение, именуемое в спецификации LDAP «информационной моделью». Эта та часть, которая устанавливает правила представления информации. Но для ответа на наш вопрос необходимо рассмотреть «модель имен» LDAP, определяющую, как информация организована.

Если вы посмотрите на рис. B.1, то увидите, что мы рассмотрели все составляющие элемента, кроме его имени. У каждого элемента есть имя, известное как DN, или его *отличительное имя* (*Distinguished Name*). DN состоит из строки RDN, или *относительных отличительных имен* (*Relative Distinguished Names*). К отличительным именам мы скоро вернемся, но сначала остановимся на составляющих блоках RDN.

RDN состоит из одной или нескольких пар «имя-значение атрибута» (name-value). Например, `cn=Jay Sekora` (где `cn` – это «common name», или имя) вполне может быть относительным отличительным именем. Имя атрибута – `cn`, а его значение – `Jay Sekora`.

Ни в спецификации LDAP, ни в спецификации X.500 не указано, из каких атрибутов должно состоять RDN. Единственное, что требуется, – уникальность RDN на каждом уровне в иерархии каталогов. Такое ограничение существует из-за того, что LDAP ничего не знает о «третьем элементе четвертой ветви дерева каталогов» и должен полагаться на уникальные имена на каждом уровне, чтобы различать на нем элементы. Посмотрим, как это ограничение действует на практике.

Возьмем, к примеру, еще одно значение RDN: `cn=Robert Smith`. Вероятно, это не лучший выбор для RDN, поскольку даже в средней организации, скорее всего, существует не один Роберт Смит. Если в организации работает много людей, а иерархия LDAP довольно плоская, то подобные пересечения имен вполне вероятны. Более правильный элемент состоит из двух атрибутов, например `cn=Robert Smith + l=Boston`. (Атрибуты в RDN объединяются при помощи знака «плюс».)

Но с исправленным RDN, к которому добавлен атрибут `l` (местоположение), по-прежнему будут проблемы. Вероятно, мы отложили конфликт имен, но не исключили полностью вероятность его возникновения. Более того, если Смит переедет, нам придется изменить и RDN элемента *и* местоположение (атрибут `l`) в этом элементе. Вероятно, самое лучшее относительное имя, которое можно использовать, – это уникальный и неизменный идентификатор данного человека. В частности, можно выбрать электронный адрес человека, тогда RDN изме-

нится на uid=rsmith. Этот пример должен дать читателю представление о принимаемых в схемах решениях.

Проницательные читатели заметят, что мы на самом деле не расширили сферу обзора, а по-прежнему возимся с единственным элементом. Обсуждение RDN было прелюдией, а вот и настоящий переход: элементы организованы в виде древоподобной[1] структуры, известной как *информационное дерево каталогов* (*Directory Information Tree, DIT*), или просто *дерево каталогов*. Лучше применять последний термин, поскольку в X.500 аббревиатура DIT обычно служит для обозначения одного общего дерева, похожего на глобальную иерархию DNS или MIB, о которой речь пойдет позже, при обсуждении SNMP.

А теперь снова вернемся к отличительным именам. Каждый элемент из дерева каталогов можно найти по его отличительному имени (DN), состоящему из относительного имени элемента (RDN), за которым следуют все RDN (разделяемые запятыми или точками с запятой), найденные при движении от него вверх к корневому элементу. Если перемещаться в направлении, указанном стрелками (рис. B.2), и запоминать при движении относительные имена, то можно создать DN для каждого выделенного элемента.

Для первого элемента получилось бы такое DN (отличительное имя):

```
cn=Robert Smith, l=main campus, ou=CCS, o=Hogwarts School, c=US
```

Для второго:

```
uid=rsmith, ou=systems, ou=people, dc=ccs, dc=hogwarts, dc=edu
```

ou – это сокращение от «organizational unit» (подразделение организации), o – сокращение от «organization» (организация), dc – «domain component» (компонент домена, подобный DNS), а c – сокращение от «country» (страна) (не имеет ничего общего с Улицей Сезам).

Часто проводится аналогия между DN и абсолютным именем пути файловой системы, но DN гораздо больше похож на почтовый адрес, т. к. он тоже записывается в порядке от частного к общему.[2] Почтовый адрес

Pat Hinds
288 St. Bucky Avenue
Anywhere, MA 02104
USA

---

[1] Структура называется *древоподобной*, а не просто *деревом* из-за того, что класс alias, о котором уже шла речь, позволяет создавать структуры каталогов, не являющиеся, строго говоря, деревьями (по крайней мере, с научной точки зрения).

[2] Понятно, что это не так в отношении способа записи адреса, принятого в России. – *Примеч. науч. ред.*

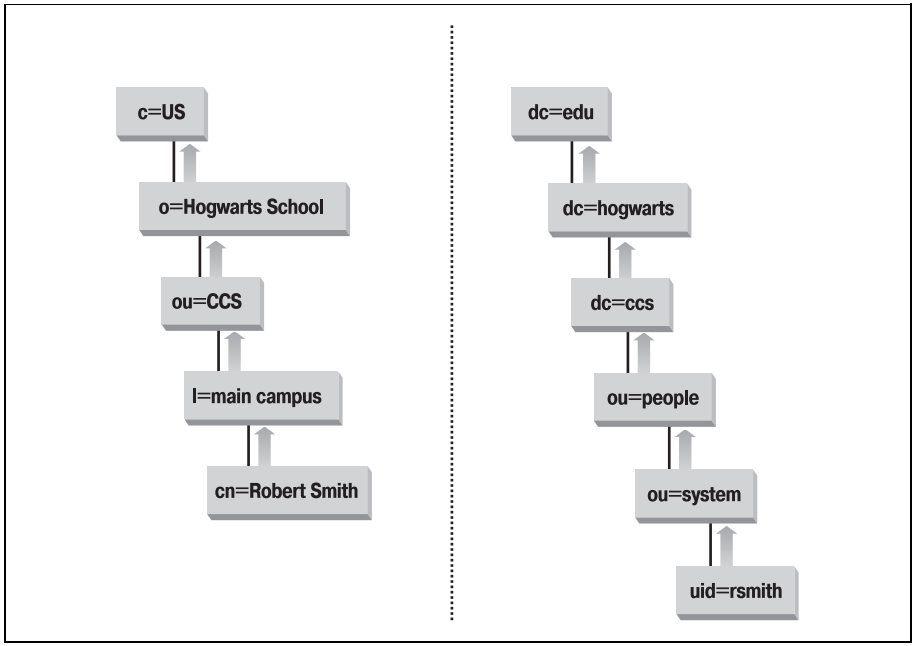

Рис. B.2. *Двигаемся вверх по дереву для создания DN*

начинается с частного (человек) и заканчивается самым общим компонентом (страна). То же имеет место и в DN. В примерах DN такой порядок хорошо заметен.

Вершина дерева каталогов называется *суффиксом* каталога, т. к. это последняя составляющая каждого отличительного имени из данного дерева каталогов. При создании иерархической инфраструктуры с использованием нескольких делегированных LDAP-серверов суффиксы очень важны. Применяя концепцию LDAPv3, известную под названием *referral*, можно добавить элемент в дерево каталогов, в котором говорится: «За всеми элементами с этим суффиксом обращайтесь к тому серверу». Направления указываются при помощи *LDAP URL*, которые очень похожи на обычные URL. Единственное их отличие – это ссылка на определенное имя или другую информацию, имеющую отношение к LDAP. Вот пример такой ссылки из RFC2255, определяющего формат LDAP URL:

```
ldap://ldap.itd.umich.edu/o=University%20of%20Michigan,c=US?postalAddress
```

C

Восьмиминутное руководство по XML

Самая впечатляющая особенность XML (eXtensible Markup Language, расширяемый язык разметки) состоит в том, что нужно знать совсем немного, чтобы начать работать. В этом приложении отражены ключевые положения. Подробные сведения можно найти в различных книгах, посвященных XML, и в источниках информации, ссылки на которые приведены в конце главы 3 «Учетные записи пользователей».

XML – это язык разметки

Понятие «язык разметки» знакомо практически каждому благодаря повсеместному распространению HTML – старшего родственника XML. Как и HTML, язык XML состоит из обычного текста с вкраплениями специальных описательных вставок или инструкций. В HTML строго определено, какие части текста, называемые *тегами*, предназначены для разметки, тогда как в XML можно создавать свои собственные теги.

XML обеспечивает гораздо более широкий спектр выражений, чем HTML. Образец такого выражения был приведен в главе 3, но вот еще один простой пример, который должен быть понятен даже тем, кто никогда раньше не сталкивался с XML:

```
<machine>
  <name> quidditch </name>
  <department> Software Sorcery </department>
  <room> 129A </room>
  <owner> Harry Potter </owner>
  <ipaddress> 192.168.1.13 </ipaddress>
</machine>
```

XML требователен

Несмотря на гибкость, XML порой более требователен, чем HTML. В нем существуют правила, которым должны подчиняться данные. Довольно сжато они описаны в спецификации XML, которую можно найти на *http://www.w3.org/TR/1998/REC-xml-19980210*. Я советую обратиться к какой-либо из аннотированных версий, подобных версии Тима Брея (Tim Bray) с *http://www.xml.com* или книге Роберта Дюшарма (Robert Ducharme) «XML: The Annotated Specification» (XML: Аннотированная спецификация) (Prentice Hall), вместо того чтобы изучать официальную спецификацию. Первая свободно доступна в «онлайне», а во второй приведено много примеров XML-кода.

Вот два правила XML, которые заставляют спотыкаться тех, кто знаком с HTML:

1. Все, что начато, должно быть закончено. В приведенном выше примере список для машины начинался с `<machine>`, а завершался с использованием `</machine>`. Без закрывающего тега это был бы пример XML, содержащий ошибку.

   В HTML такие теги, как `<img src="picture.jpg">`, вполне могут не иметь закрывающего тега. Но в XML это неверно, поэтому его нужно переписать либо так:

   ```
   <img src="picture.jpg"> </img>
   ```

   либо так:

   ```
   <img src="picture.jpg" />
   ```

   Слэш в конце последнего тега сообщает XML-анализатору, что он является одновременно и открывающим, и закрывающим тегом. Данные и окружающие их открывающий и закрывающий теги называются *элементом* (*element*).

2. Открывающие и закрывающие теги должны в точности соответствовать друг другу. Нельзя изменять их регистр. Если используется открывающий тег `<MaChINe>`, то закрывающим должен быть `</MaChINe>`, но не `</MACHine>` и не тег с любой другой комбинацией регистров. В этом отношении HTML гораздо более снисходителен.

Это два основных правила из спецификации XML. Но иногда автор определяет собственные правила, которым должен подчиняться анализатор XML. Под «подчиняться» следует подразумевать «выдавать предупреждения» или «останавливать анализ» при чтении XML-данных. Если использовать в качестве примера предыдущее определение машины в базе данных, то можно ввести дополнительное правило: «Все элементы `<machine>` должны содержать элементы `<name>` и `<ipaddress>`». Можно также ограничить содержимое элемента определенными значениями, подобными «YES» или «NO».

Эти правила определяются менее очевидным образом, чем все остальное, что будет рассмотрено, т. к. в настоящее время существует несколько конкурирующих и дополняющих друг друга предложений для определения «языка». В конце концов, XML станет самоопределяемым (т. е. структуру документа будет описывать либо сам документ, либо нечто с ним связанное).

В текущей спецификации XML используется DTD (Document Type Definition, определение типа документа), основа SGML. Вот небольшой пример кода из спецификации XML, в котором определение типа находится в начале самого документа:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
]>
<greeting>Hello, world!</greeting>
```

В первой строке примера задается версия XML и применяемая кодировка для документа (в данном случае это Unicode). Следующие три строки определяют типы данных из этого документа. В последней строке приводится сам документ (элемент `<greeting>`).

При желании определить способ, с помощью которого подтверждалась бы правильность кода `<machine>` в первом примере из начала приложения, следовало бы добавить в начало файла нечто подобное:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE machines [
  <!ELEMENT machine (name,department,room,owner,ipaddress)>
  <!ELEMENT name       (#PCDATA)>
  <!ELEMENT department (#PCDATA)>
  <!ELEMENT room       (#PCDATA)>
  <!ELEMENT owner      (#PCDATA)>
  <!ELEMENT ipaddress  (#PCDATA)>
]>
```

Это определение требует, чтобы данные, соответствующие машине, состояли бы из элементов `name`, `department`, `room`, `owner` и `ipaddress` (именно в таком порядке). Каждый из этих элементов описывается как `PCDATA` (см. раздел «Пережитки» в конце данного приложения).

В другом популярном предложении, которое пока не является спецификацией, рекомендовано в DTD-подобных целях использовать описания данных под названием *схемы* (*schemas*). Сами схемы пишутся на XML. Вот пример кода схемы, использующей реализацию от Microsoft с *http://www.w3.org/TR/1998/ NOTE-XML-data/*:

```
<?XML version='1.0' ?>
<schema id='MachineSchema'
        xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:dt="urn:schemas-microsoft-com:datatypes">
```

```
<!-определяем типы элементов (все они являются просто строками/PCDATA) -->
    <elementType id="name">
        <string/>
    </elementType>
    <elementType id="department">
        <string/>
    </elementType>
    <elementType id="room">
      <string/>
    </elementType>
    <elementType id="owner">
        <string/>
    </elementType>
    <elementType id="ipaddress">
        <string/>
    </elementType>

    <!-определяем сам элемент -->
    <elementType id="Machine" content="CLOSED">
       <element type="#name"       occurs="REQUIRED"/>
       <element type="#department" occurs="REQUIRED"/>
       <element type="#room"       occurs="REQUIRED"/>
       <element type="#owner"      occurs="REQUIRED"/>
       <element type="#ipaddress"  occurs="REQUIRED"/>
    </elementType>
</schema>
```

Технология схем XML до сих пор (на момент написания книги) находится в стадии обсуждения. XML-данные, которые использовались в приведенном выше примере, являются всего лишь одним из предложений, рассматриваемым группой Working Group. Поскольку эта технология очень быстро развивается, я советую следить за текущими стандартами (их можно найти на *http://www.w3.org*) и за тем, насколько совместимо с ними ваше программное обеспечение.

Как и вполне зрелый механизм DTD, так и новый механизм схем очень быстро могут стать запутанными, поэтому оставим дальнейшие дискуссии книгам, посвященным XML/SGML.

Два ключевых термина XML

Не изучив эти два важных термина, нельзя «далеко уйти» в XML. Говорят, что XML-данные *корректны* (*well-formed*), если они имеют верный XML-синтаксис и следуют правилам грамматики (соответствие тегов и т. д.). Часто проверка документа на корректность позволяет выявить опечатки в XML-файлах. Это уже преимущество, если данные, с которыми вы имеете дело, содержат конфигурационную инфор-

мацию, как в случае с базой данных имеющихся машин, отрывок из которой приведен выше.

Говорят, что XML-данные *действительны* (*valid*), если они удовлетворяют правилам, установленным одним из механизмов определения данных. Например, если ваши данные удовлетворяют DTD, то это действительные XML-данные.

Действительные данные по определению корректны, но обратное верно не всегда. Вполне могут существовать корректные данные без ассоциированной с ними DTD или схемы. Если такие данные верно анализируются, то они корректны, но не действительны.

Пережитки

Вот три термина, встречающиеся в литературе по XML, которые могут запутать новичка:

Attribute (атрибут)

Это описание элемента, являющееся частью открывающего тега. Если воспользоваться предыдущим примером, то в `<img src="picture.jpg" />` атрибутом будет `src="picture.jpg"`. О том, когда применять содержимое элемента, а когда атрибуты, в мире XML ведутся споры. Самые лучшие указания именно по этой теме можно найти на *http://www.oasis-open.org/cover/elementsAndAttrs.html*.

CDATA

Термин CDATA (Character Data, символьные данные) используется в двух смыслах. В первом он чаще всего относится ко всему в XML-документе, что не является разметкой (тегами и т. д.). А во втором подразумевает понятие *разделов CDATA* (*sections*). Объявление раздела CDATA указывает, что анализатор XML не должен рассматривать данный раздел, даже если он содержит текст, который может быть построен как разметка.

PCDATA

В аннотации Тима Брея к спецификации XML (упомянутой ранее) приводится следующее определение:

> Строка PCDATA является сокращением от «Parsed Character Data». Это еще одно наследие SGML; «parsed» подразумевает, что процессор XML будет просматривать текст в поисках разметки, обозначаемой символами `<` и `&`.
>
> Можно считать их данными, состоящими из CDATA и, вероятно, некоторой разметки. Большая часть XML-данных попадает под это определение.

В изучении XML имеются свои тонкости. Это маленькое руководство должно помочь вам приступить к работе.

D

Пятнадцатиминутное руководство по SQL

Реляционные базы данных могут быть отличным инструментом для системного администрирования. Доступ и управление реляционными базами осуществляется при помощи команд SQL (Structured Query Language, языка структурированных запросов). Так что системному администратору было бы неплохо выучить хотя бы основы SQL. Цель данного приложения состоит не в том, чтобы сделать читателя настоящим программистом или даже администратором баз данных, – для этого нужны годы работы и значительный опыт. Тем не менее, можно показать достаточно, чтобы начать использовать SQL. Вы не сумеете говорить на этом языке, но станете, по крайней мере, понимать смысл услышанного и будете знать достаточно, чтобы лучше разобраться в сути предмета, если это будет необходимо. В главе 7 «Администрирование баз данных SQL» такие «строительные блоки» активно используются при интеграции SQL с Perl.

SQL – это командный язык для выполнения операций над базами данных и их составляющими. Чаще всего вы будете иметь дело с такими составляющими, как таблицы. Структура столбцов и строк делает их похожими на электронные таблицы, но это сходство только внешнее. Элементы таблицы не используются для представления отношений с другими элементами, они хранят не формулы, а обычные данные. Большая часть SQL-операторов предназначена для работы с данными из этих строк и столбцов, обеспечивая пользователям возможность добавления, удаления, выборки, сортировки и связывания их между таблицами.

Рассмотрим некоторые операторы SQL. Чтобы опробовать выбранные операторы в действии, необходимо обратиться к базе данных SQL. Возможно, у читателя уже есть доступ к серверу от Oracle, Sybase, Infor-

mix, IBM, Microsoft и т. д. Если нет, отличную базу данных MySQL (с исходными кодами) можно получить на *http://www.mysql.org*.

В этом приложении будет рассмотрен обобщенный SQL, хотя каждый сервер баз данных имеет свои особенности. Мы упомянем обо всех операторах SQL, свойственных определенным реализациям баз данных.

В приведенных ниже примерах в операторах будет использоваться верхний регистр, как в большинстве книг по SQL. Согласно этому стандарту в верхнем регистре записываются все зарезервированные слова в операторах.

В большинстве примеров данного приложения будет применяться таблица, повторяющая «плоскую» базу данных из главы 5 «Службы имен TCP/IP». Напомним, как выглядят эти данные в виде таблицы (табл. D.1).

Таблица D.1. База данных машин

name	ipaddr	aliases	owner	dept	bldg	room	manuf	model
shimmer	192.168.1.11	shim shimmy shimmy-doodles	David Davis	software	main	309	Sun	Ultra60
bendir	192.168.1.3	ben bendoodles	Cindy Coltrane	IT	west	143	Apple	7500/100
sander	192.168.1.55	sandy micky micky-doo	Alex Rollins	IT	main	1101	Intergraph	TD-325
sulawesi	192.168.1.12	sula sulee	Ellen Monk	design	main	1116	Apple	G3

Создание/удаление баз данных и таблиц

В самом начале сервер пуст и в нем нет объектов, которые могут быть нам полезны. Давайте построим свою базу данных:

```
CREATE DATABASE sysadm ON userdev=10 LOG ON userlog=5
GO
```

Данная команда создает 10-мегабайтную базу данных на устройстве `userdev` с 5-мегабайтным файлом журнала на устройстве `userlog`. Эта команда специфична для серверов Sybase/Microsoft SQL Server, т. к. создание баз данных (если оно вообще выполняется) на разных серверах производится по-разному.

Команда GO применяется с интерактивными клиентами баз данных и служит указанием на то, что необходимо выполнить предыдущую команду. Но это не SQL-оператор. В следующих примерах будем считать, что команду GO необходимо выполнять после каждого SQL-оператора, если используется один из таких клиентов. Кроме того, комментарии в SQL будут обозначаться при помощи «--».

Чтобы удалить эту базу данных, необходимо выполнить команду DROP:

```
DROP DATABASE sysadm
```

Теперь создадим пустую таблицу, в которой будет храниться информация из табл. D.1.

```
USE sysadm
-- Последнее напоминание: перед тем как выполнить следующую
-- команду, необходимо набрать GO (если вы используете
-- интерактивный клиент)
CREATE TABLE hosts (
  name      character(30)     NOT NULL,
  ipaddr    character(15)     NOT NULL,
  aliases   character(50)     NULL,
  owner     character(40)     NULL,
  dept      character(15)     NULL,
  bldg      character(10)     NULL,
  room      character(4)      NULL,
  manuf     character(10)     NULL,
  model     character(10)     NULL
)
```

Сначала мы указываем, какая база данных (*sysadm*) будет использоваться. Оператор USE оказывает действие только в том случае, когда он выполняется отдельно до запуска других команд, поэтому ему нужен собственный оператор GO.

Затем мы создаем таблицу, указывая ее имя, тип и длину данных, а также настройки NULL/NOT NULL для каждого столбца. Теперь немного поговорим о типах данных.

В таблице можно хранить различные типы данных, включая числа, даты, текст и даже изображения и другие двоичные данные. Столбцы в ней создаются для хранения данных определенного типа. У нас скромные потребности, поэтому таблица состоит из столбцов, хранящих простые строки characters (символов). В SQL можно строить определяемые пользователями псевдонимы типов данных, такие как ip_address или employee_id. Определяемые пользователями типы данных применяются при создании таблиц для поддержки читаемости ее структуры, а также для форматов данных, которые не должны меняться от столбца к столбцу в разных таблицах.

Последний набор параметров в предыдущей команде определяет, обязательным ли является поле. Если параметр равен NOT NULL, то новую

строку нельзя добавить в случае, если в этом поле у нее нет данных. В нашем примере важны имя машины и ее IP-адрес, так что следует объявить эти поля как NOT NULL. Все остальные поля необязательны (хоть и желательны). Помимо NULL/NOT NULL существуют и другие ограничения, накладываемые на поля для согласованности данных. Например, чтобы убедиться, что две машины не называются одинаково, можно изменить строку

```
name      character(30)     NOT NULL,
```

на:

```
name      character(30)     NOT NULL CONSTRAINT unique_name UNIQUE,
```

Это ограничение мы называем unique_name. Введенные названия позволяют получать более осмысленные сообщения об ошибках, которые выдаются при нарушении ограничений. Изучите документацию вашего сервера, чтобы выяснить, какие еще ограничения можно применять к таблицам.

Удалить таблицу из базы данных значительно проще, чем создать ее:

```
USE sysadm
DROP TABLE hosts
```

Добавление данных в таблицу

Теперь у нас есть пустая таблица; так что рассмотрим два способа добавления в нее данных. Вот первый способ:

```
USE sysadm
INSERT hosts
   VALUES (
     'shimmer',
     '192.168.1.11',
     'shim shimmy shimmydoodles',
     'David Davis',
     'Software',
     'Main',
     '309',
     'Sun',
     'Ultra60'
   )
```

В первой строке мы сообщаем серверу, что собираемся работать с объектами из базы данных *sysadm*. Во второй строке выбирается таблица *hosts* и в нее добавляются строки – по одному полю за один раз. Такой вариант команды *INSERT* добавляет в таблицу всю строку целиком (ту, в которой определены все поля). Чтобы добавить строку

только частично, можно указать, какие поля следует дописать, например, так:

```
USE sysadm
INSERT hosts (name,ipaddr,owner)
   VALUES (
     'bendir',
     '192.168.1.3',
     'Cindy Coltrane'
    )
```

Команда INSERT завершится с ошибкой, если попытаться добавить строку, в которой определены не все обязательные (NOT NULL) поля.

INSERT можно использовать и для добавления данных из одной таблицы в другую, такое применение будет рассмотрено позже. Во всех остальных примерах будем считать, что таблица *hosts* заполнена до конца при помощи команды INSERT в первой форме.

Запрос информации

Являясь системным администратором, чаще всего вы будете применять SQL-команду SELECT, которая используется для получения информации с сервера. Перед тем как говорить об этой команде, нужно заметить, что SELECT – это пропуск в мир SQL. Мы покажем только самые простые формы данной команды. Умение создавать хорошие запросы (и умение проектировать базы данных, к которым легко такие запросы строить) – это искусство, и более подробно этот вопрос рассматривается в книгах, целиком посвященных SQL и базам данных.

В самой простой форме SELECT служит для получения информации о сервере и соединении. В этом случае не нужно определять источник данных. Вот два примера:

```
-- оба зависят от производителя базы данных
SELECT @@SERVERNAME
SELECT VERSION();
```

Первый оператор возвращает имя сервера для Sybase или MS-SQL; второй – текущую версию сервера MySQL.

Получение всех записей из таблицы

Для получения всех данных из таблицы *hosts* применяется такой SQL-код:

```
USE sysadm
SELECT * FROM hosts
```

В результате возвращаются все строки и поля, причем поля следуют в той последовательности, в которой они определялись при создании базы данных:

```
name      ipaddr        aliases                    owner            dept
bldg   room manuf      model
--------- ------------ -------------------------- ---------------- ------
-- ------ ---- ---------- ---------
shimmer   192.168.1.11  shim shimmy shimmydoodles   David Davis      Software
Main  309  Sun         Ultra60
bendir    192.168.1.3   ben bendoodles             Cindy Coltrane    IT
West  143  Apple       7500/100
sander    192.168.1.55  sandy micky mickydoo       Alex Rollins      IT
Main  1101 Intergraph TD-325
sulawesi  192.168.1.12  sula su-lee                Ellen Monk        Design
Main  1116 Apple       G3
```

Если нужно получить только определенные поля, следует явно указать их имена:

```
USE sysadm
SELECT name,ipaddr FROM hosts
```

Когда мы определяем поля по имени, они возвращаются в той последовательности, в которой указывались, независимо от порядка, используемого при создании таблицы. Например, для получения связи IP-адресов со зданиями можно применить следующую команду:

```
USE sysadm
SELECT bldg,ipaddr FROM hosts
```

В результате получим:

```
bldg       ipaddr
---------- ---------------
Main       192.168.1.11
West       192.168.1.3
Main       192.168.1.55
Main       192.168.1.12
```

Получение подмножества строк из таблицы

Базы данных не были бы такими интересными, если бы из них нельзя было получить некое подмножество данных. В SQL употребляется команда SELECT, в которую добавлено ключевое слово WHERE для определения условия:

```
USE sysadm
SELECT * FROM hosts WHERE bldg="Main"
```

В результате получаем:

```
name      ipaddr        aliases                    owner            dept
bldg   room manuf      model
--------- ------------ --------------------------- ---------------- ------
-- ------ ---- ---------- ---------
shimmer   192.168.1.11  shim shimmy shimmydoodles  David Davis      Software
Main  309  Sun        Ultra60
sander    192.168.1.55  sandy micky mickydoo       Alex Rollins     IT
Main  1101 Intergraph TD-325
sulawesi  192.168.1.12  sula su-lee                Ellen Monk       Design
Main  1116 Apple      G3
```

С ключевым словом WHERE можно использовать стандартные условные операторы, применяемые в программировании:

```
=      >      >=      <      <=      <>
```

В отличие от Perl, в SQL нет отдельных операторов для сравнения строк и чисел.

Условные операторы можно объединять посредством AND/OR и отрицать при помощи NOT. Проверить, является ли поле пустым, позволяет оператор IS NULL, а проверить обратное – IS NOT NULL. Например, этот фрагмент SQL-кода выведет список машин, для которых в таблице не указаны владельцы:

```
USE sysadm
SELECT name FROM hosts WHERE owner IS NULL
```

Если требуется найти все строки, в которых значения некоторого поля равны одному из указанных, можно использовать оператор IN для задания списка:

```
USE sysadm
SELECT name FROM hosts WHERE dept IN ('IT', 'Software')
```

Ответом будет список машин из отделов «IT» и «Software». SQL также позволяет получить строки, совпадающие с диапазоном значений (полезнее всего это применять с численными данными и датами), при помощи оператора BETWEEN. Вот пример запроса, возвращающий список машин, находящихся в основном здании на десятом этаже:

```
USE sysadm
SELECT name FROM hosts
  WHERE (bldg = 'Main') AND
        (room BETWEEN '1000' AND '1999')
```

Наконец, ключевое слово WHERE можно использовать с LIKE для выбора строк при помощи слабого механизма соответствия шаблону (слабого в сравнении с регулярными выражениями в Perl). Например, следующий запрос выбирает все машины, в псевдонимах которых встречается строка «doodles»:

```
USE sysadm
SELECT name FROM hosts WHERE aliases LIKE '%doodles%'
```

Обратите внимание, какие метасимволы поддерживаются в LIKE (табл. D.2).

Таблица D.2. Метасимволы LIKE

Метасимвол	Значение	Ближайший эквивалент из регулярных выражений Perl
%	Ноль или более символов	.*
_	Один символ	.
[]	Один символ из указанного списка или диапазона	[]

В некоторых серверах баз данных добавлены расширения к SQL, позволяющие применять регулярные выражения в операторах SELECT. Например, в MySQL существует оператор REGEXP, который можно использовать с SELECT. REGEXP не обладает всей силой регулярных выражений Perl, но он значительно увеличивает их гибкость по сравнению со стандартными метасимволами SQL.

Простая обработка данных, возвращаемых в результате запросов

У оператора SELECT существуют два полезных ключевых слова: DISTINCT и ORDER BY. Первое позволяет изъять из запроса повторяющиеся записи. При желании получить список всех различных производителей, представленных в таблице *hosts*, можно было бы использовать DISTINCT:

```
USE sysadm
SELECT DISTINCT manuf FROM hosts
```

При необходимости получить отсортированные данные, можно было бы применить ORDER BY:

```
USE sysadm
SELECT name,ipaddr,dept,owner FROM hosts ORDER BY dept
```

SQL имеет несколько операторов, преобразующих данные, возвращаемые в результате запроса. Такая возможность позволяет изменять имена полей, выполнять итоговые вычисления и вычисления внутри и между полями, изменять формат выводимых полей, осуществлять подзапросы и совершать множество иных действий. За информацией о различных ключевых словах, используемых с SELECT, обратитесь к литературе, посвященной SQL.

Добавление результатов запроса в другую таблицу

На некоторых SQL-серверах можно на лету создать новую таблицу, содержащую результаты запроса, при помощи ключевого слова INTO:

```
USE sysadm
SELECT name,ipaddr INTO itmachines FROM hosts WHERE dept = 'IT'
```

Этот оператор работает так же, как и предыдущие, за одним исключением: результаты данного запроса помещаются в таблицу под названием *itmachines*. На некоторых серверах такая таблица создается на лету в случае, если она еще не существует. Этот оператор можно считать эквивалентом оператора «>» в большинстве систем Unix и командных интерпретаторах NT.

Отдельные серверы баз данных (как MySQL) не поддерживают оператор SELECT INTO; для выполнения этого действия в них нужно применять команду *INSERT*. Другие серверы, например MS-SQL и Sybase, требуют установки специального флага на базу данных для использования SELECT INTO, иначе команда завершится с ошибкой.

Изменение информации в таблице

Знание команды SELECT может пригодиться и при работе с другими командами. Например, упомянутая ранее INSERT также способна принимать ключевое слово SELECT. Это позволяет вставлять запрашиваемую информацию в существующую таблицу. Если отдел программного обеспечения вздумает объединиться с отделом IT, можно будет добавить машины из этого отдела в таблицу *itmachines*:

```
USE sysadm
INSERT itmachines
  SELECT name,ipaddr FROM hosts
  WHERE dept = 'Software'
```

Если нужно изменить какую-либо строку в таблице, достаточно выполнить команду UPDATE. Например, если все отделы компании переведут в одно здание Central, то имя здания в каждой строке можно будет изменить так:

```
USE sysadm
UPDATE hosts
  SET bldg = 'Central'
```

Более вероятно, что нам понадобится изменить только определенные строки из таблицы. Для этого применяется полезное ключевое слово WHERE, рассмотренное при обсуждении оператора SELECT:

```
USE sysadm
UPDATE hosts
  SET dept = 'Development'
  WHERE dept = 'Software'
```

Упомянутая команда изменит название отдела `Software` на `Development`. А данная команда переведет машину *bendir* в основной корпус:

```
USE sysadm
UPDATE hosts
  SET bldg = 'Main'
  WHERE name = 'bendir'
```

При желании удалить строку или несколько строк из таблицы, вместо того чтобы обновлять их, выполните команду `DELETE`:

```
USE sysadm
DELETE hosts
  WHERE bldg = 'East'
```

Не существует способа отменить операцию `DELETE`, так что будьте осторожны.

Установление связей между таблицами

Реляционные базы данных предлагают множество способов установить связи между данными из двух или более таблиц. Этот процесс называется объединением («joining») таблиц. Объединения очень быстро могут стать сложными, учитывая количество используемых запросов и точный контроль, который программист имеет над возвращаемыми данными. Тем, кого интересуют такие детали, лучше заглянуть в книгу по SQL.

Рассмотрим один пример объединения. Здесь будет использоваться таблица под названием *contracts,* в которой содержится информация о гарантиях на каждую машину (табл. D.3).

Таблица D.3. Таблица Contracts

name	servicevendor	startdate	enddate
bendir	Dec	09-09-1995	06-01-1998
sander	Intergraph	03-14-1998	03-14-1999
shimmer	Sun	12-12-1998	12-12-2000
sulawesi	Apple	11-01-1995	11-01-1998

Вот один из способов установить отношение между таблицей *hosts* и таблицей *contracts* при помощи объединения:

```
USE sysadm
```

```
SELECT name,servicevendor,enddate
  FROM contracts, hosts
  WHERE contracts.name = hosts.name
```

Проще всего понять этот код, если начать читать его с середины. Условие `FROM contracts, hosts` говорит серверу о том, что связь устанавливается между таблицами *contracts* и *hosts*. Условие `WHERE contracts.name = hosts.name` сообщает, что мы ищем совпадения между строками из таблиц *contracts* и *hosts*, основываясь на содержимом поля `name` из каждой таблицы. Наконец, строка `SELECT...` определяет поля, которые мы хотим включить в получаемые данные.

Дополнительные аспекты SQL

Перед тем как завершить это руководство, нужно упомянуть о более «продвинутых» темах из SQL, с которыми вы можете столкнуться.

Представления

Некоторые SQL-серверы позволяют создавать различные *представления* (*views*) таблицы, которые похожи на волшебные постоянные запросы `SELECT`. Когда вы создаете представление при помощи специального запроса `SELECT`, результаты запроса не исчезают и ведут себя как отдельная таблица. Как ко всякой таблице, к ним можно посылать запросы. Изменения представлений с некоторыми ограничениями передаются оригинальной таблице или таблицам.

Заметьте, я сказал *таблицам*. Именно тут проявляется волшебство представлений: можно создать представление таблицы, состоящее из объединения этой и другой таблицы. Такое представление ведет себя как одна большая виртуальная таблица. Изменения представления передаются обратно таблицам, участвующим в объединении, создающем представление.

Можно также создать представление, новое поле которого будет состоять из результатов вычислений между другими полями этой таблицы, почти как в электронных таблицах. Представления полезны и для более обычных целей, например, для упрощения запросов (т. е. можно будет выбрать меньшее количество полей) и реструктуризации данных (т. е. представление данных остается первоначальным, даже если меняются поля в структуре таблицы).

Вот как создать представление, используемое для упрощения запросов:

```
USE sysadm
CREATE VIEW ipaddr_view AS SELECT name, ipaddr FROM hosts
```

Теперь можно применить очень простой запрос, чтобы получить только ту информацию, которая нам нужна:

```
USE sysadm
SELECT * FROM ipaddr_view
```

А вот результат запроса:

```
name                          ipaddr
----------------------------- ---------------
shimmer                       192.168.1.11
bendir                        192.168.1.3
sander                        192.168.1.55
sulawesi                      192.168.1.12
```

Как и таблицы, представления можно удалять, используя разновидность команды DROP:

```
USE sysadm
DROP VIEW ipaddr_view
```

Курсоры

До сих пор сервер возвращал все результаты запроса по его завершении. Иногда бывает предпочтительнее получать ответ построчно. Чаще всего это справедливо при встраивании SQL-запросов в другие программы. Если запрос возвращает десятки тысяч строк, очень велика вероятность, что вам захочется обработать результаты построчно, а не хранить все в памяти для дальнейшего использования. Этот метод применяется в большинстве случаев, когда необходимо обращаться к SQL из Perl. Вот маленькая программа на SQL, в которой показано употребление курсоров на сервере Sybase или MS-SQL:

```
USE sysadm
-- объявляем переменные
DECLARE @hostname character(30)
DECLARE @ip character(15)

-- объявляем курсор
DECLARE hosts_curs CURSOR FOR SELECT name,ipaddr FROM hosts

-- открываем курсор
OPEN hosts_curs

-- обходим в цикле таблицу, получая по одной строке за один раз
-- до тех пор, пока не получим ошибку
FETCH hosts_curs INTO @hostname,@ip
WHILE (@@fetch_status = 0)
  BEGIN
     PRINT "----"
     PRINT @hostname
     PRINT @ip
     FETCH hosts_curs INTO @hostname,@ip
  END
```

```
-- закрываем курсор (это не обязательно, если
-- далее следует DEALLOCATE)
CLOSE hosts_curs

-- снимаем определение (undefine) курсора
DEALLOCATE hosts_curs
```

В результате получается следующее:

```
----
shimmer
192.168.1.11
----
bendir
192.168.1.3
----
sander
192.168.1.55
----
sulawesi
192.168.1.12
```

Хранимые процедуры

Большинство систем баз данных позволяют загружать на сервер SQL код, где он хранится в оптимизированном, проанализированном виде для быстрого выполнения. Такой код называется *хранимыми процедурами* (*stored procedures*). Хранимые процедуры часто являются важным компонентом SQL для администраторов, поскольку на них основана большая доля управления сервером. Например, чтобы изменить владельца базы данных *sysadm* в Sybase, нужно выполнить следующее:

```
USE sysadm
sp_changedbowner "jay"
```

Примеры вызова хранимых процедур можно найти в главе 7. Теперь, когда вы знакомы с основами SQL, можно браться и за нее.

E

Двадцатиминутное руководство по SNMP

Простой протокол управления сетью (Simple Network Management Protocol, SNMP) повсеместно используется для управления устройствами в сети. К сожалению, как говорилось в начале главы 10 «Безопасность и наблюдение за сетью», SNMP не очень-то и простой протокол (несмотря на название). Это довольно длинное руководство предоставит вам информацию, необходимую для работы с протоколом SNMP версии 1.

SNMP основывается на предположении, что у вас есть управляемая станция (management station), получающая информацию от SNMP-агента, запущенного на удаленном устройстве. Можно настроить агент так, чтобы он сообщал управляемой станции о наступлении важных условий (например, значения счетчика превышают допустимую границу). При программировании на Perl в главе 10 «Безопасность и наблюдение за сетью» мы, по существу, сами были управляемой станцией, посылающей SNMP-агентов на другие сетевые устройства.

Поговорим о SNMP версии 1. Было предложено семь версий протокола: (SNMPv1, SNMPsec, SNMPv2p, SNMPv2c, SNMPv2u, SNMPv2* и SNMPv3). Версия v1 – единственная версия, которая широко используется, хотя ожидается, что популярность v3 будет заметно выше благодаря лучшей архитектуре безопасности.

И Perl и SNMP имеют простые типы данных. В качестве основного типа в Perl используется скалярный тип данных. В Perl списки и хэши – это всего лишь набор скаляров. В SNMP вы также будете иметь дело со скалярными переменными (variables). Переменные SNMP могут иметь значения одного из четырех примитивных типов: целые, строки, объекты, идентификаторы объектов (об этом позже) или нулевые значения. Как и в Perl, в SNMP можно объединять набор связанных

переменных в более крупные структуры (чаще всего это *таблицы*). На этом их сходство заканчивается.

Когда дело касается имен переменных, Perl и SNMP радикально отличаются. В Perl можно давать переменным практически любые имена (принимая во внимание некоторые ограничения). Подход к именованию переменных в SNMP гораздо более строгий. Все SNMP-переменные существуют внутри иерархической структуры, известной как база управляющей информации (Management Information Base, MIB). В ней определены все допустимые имена переменных. MIB, теперь уже версии MIB-II, определяет древовидную структуру для всех объектов (и их имен), которыми можно управлять через SNMP.

В некотором роде, MIB похожа на файловую систему. Вместо организации файлов MIB логически организует управляющую информацию в виде иерархической древовидной структуры. Каждый узел в этом дереве имеет короткую строку текста, называемую *меткой*, и номер, соответствующий позиции узла на этом уровне дерева. Чтобы понять, как все это работает, найдем SNMP-переменную в MIB, применяемую для хранения собственного системного описания. Потерпите; нам придется немного погулять по дереву (восьми уровней будет достаточно), чтобы добраться до нее.

Посмотрите на вершину дерева MIB (рис. E.1).

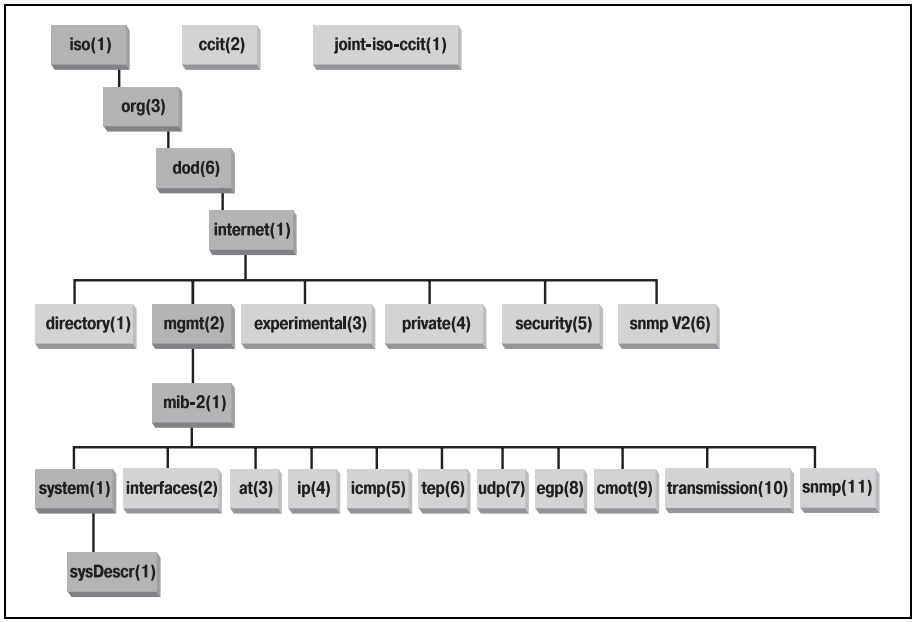

Рис. E.1. *Поиск sysDescr(1) в MIB*

Вершина дерева состоит из стандартных организаций: `iso(1)`, `ccitt(2)`, `joint-iso-ccitt(3)`. Под узлом `iso(1)` расположен узел `org(3)` для дру-

гих организаций. Под этим узлом находится dod(6), то есть Министерство обороны (Department of Defense). Ниже расположен internet(1), поддерево для сообщества Интернета.

Именно здесь начинается кое-что интересное. Комитет IAB (Internet Activities Board) поместил ряд поддеревьев под internet(1) (табл. E.1).

Таблица E.1. Поддеревья узла internet(1)

Поддерево	Описание
Directory(1)	Каталог OSI
mgmt(2)	Стандартные объекты RFC
Experimental(3)	Эксперименты с Internet
Private(4)	Зависит от производителя
Security(5)	Безопасность
snmpV2(6)	Описание работы SNMP

Поскольку нас интересует применение SNMP для управления устройствами, мы обратимся к ветви mgmt(2). Первый узел под mgmt(2) – сам MIB (это почти рекурсия). А раз существует только один MIB, то единственный узел под mgmt(2) – это mib-2(1).

По-настоящему MIB начинается с этого уровня в дереве. Мы ищем первую группу ветвей, называемых группами объектов, где хранятся переменные, к которым хотим обратиться:

```
system(1)
interfaces(2)
at(3)
ip(4)
icmp(5)
tcp(6)
udp(7)
egp(8)
cmot(9)
transmission(10)
snmp(11)
```

Раз мы ищем «системное описание» SNMP-переменной, будет логично посмотреть в группу system(1). Первый узел в этом дереве – sysDescr(1). Мы нашли то, что искали.

Зачем связываться с обходом деревьев? В результате такого путешествия можно получить идентификатор объекта sysDescr(1). Идентификатор объекта, или OID, – это всего лишь набор чисел, разделенных точками, с каждого уровня дерева, которые встречались на пути к объекту. Вот как это выглядит в графическом виде (рис. E.2).

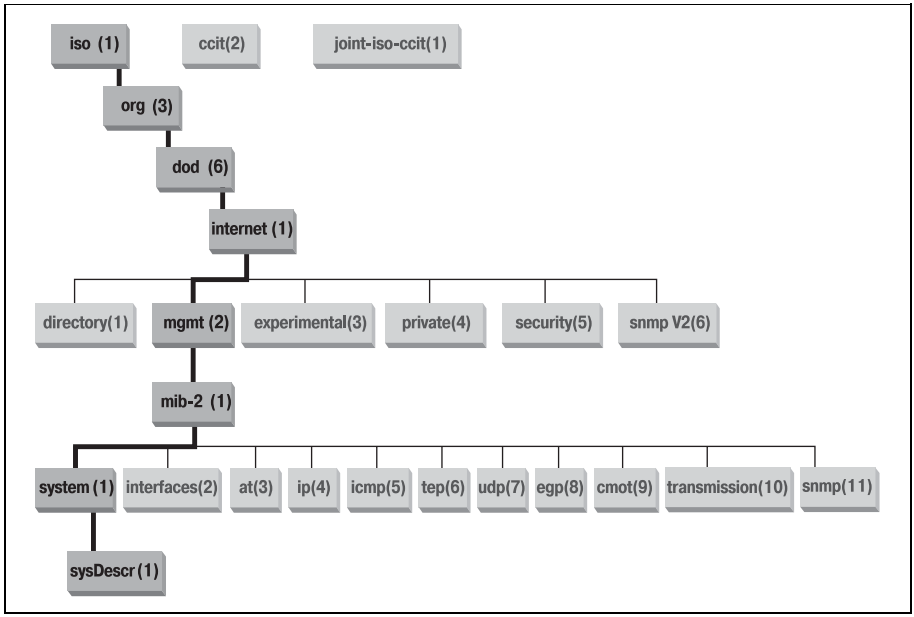

Рис. E.2. *Поиск OID для нужного объекта*

Поскольку OID для дерева Интернета – это `1.3.6.1`, то OID для системной группы объектов – `1.3.6.1.2.1.1`, а OID для объекта `sysDescr` – `1.3.6.1.2.1.1.1`.

Если нужно воспользоваться таким идентификатором на практике, то для получения значения этой переменной следует добавить к идентификатору еще одно число. Нам нужно добавить `.0`, что соответствует первому (и единственному, т. к. устройство не может иметь более одного описания) *экземпляру* этого объекта.

Что ж, давайте так и поступим; воспользуемся этим идентификатором, рассматривая SNMP в действии. В этом приложении будут применяться в демонстрационных целях инструменты из пакета UCD-SNMP. Пакет UCD-SNMP, который можно найти на *http://ucd-snmp.ucdavis.edu/*, является отличной бесплатной реализацией SNMPv1 и SNMPv3. Мы используем именно эту реализацию SNMP, т. к. один из модулей Perl связан с его библиотекой, но и все остальные клиенты, которые могут посылать SNMP-запросы, действуют практически так же. После знакомства с SNMP-утилитами командной строки читателю будет легко переходить на эквиваленты в Perl.

Инструменты командной строки из UCD-SNMP требуют ставить первой точку в OID/имени переменной, если оно начинается от вершины дерева. В противном случае считается, что OID/имя переменной начинается с вершины дерева `mib-2`. Вот два способа получить системное описание для машины *solarisbox*:

```
$ snmpget solarisbox public .1.3.6.1.2.1.1.1.0
$ snmpget solarisbox public .iso.org.dod.internet.mgmt.mib-2.sys-
                                        tem.sysDescr.0
```

Обе эти строки позволяют получить одно и то же:

```
system.sysDescr.0 = Sun SNMP Agent, Ultra-1
```

Но вернемся к теории. Очень важно помнить, что «P» в SNMP означает *протокол* (Protocol). Сам SNMP – это всего лишь протокол для взаимодействия между элементами в управляющей инфраструктуре. Операции, или единицы данных протокола (Protocol Data Units, PDU), считаются *простыми* (*simple*). Вот какие единицы PDU встречаются чаще всего, особенно при программировании на Perl[1]:

get-request

`get-request` – «рабочая лошадка» в семействе PDU. Посредством `get-request` опрашивают SNMP-объект, для того чтобы получить значение SNMP-переменной. Очень многие при работе с SNMP никогда не используют ничего, кроме этой операции.

get-next-request

`get-next-request` очень похож на `get-request`, только он возвращает элемент из MIB, который находится *после* указанного («первый лексикографический наследник» (first lexicographic successor) в терминах RFC). Эта операция бывает полезна, если необходимо найти все элементы из логической таблицы. Например, можно послать несколько последовательных запросов `get-next-requests` для обращения к каждой строке из ARP-таблицы рабочей станции. Очень скоро будет приведен пример этой операции в действии.

set-request

`set-request` делает именно то, что можно ожидать; он пытается изменить значение SNMP-переменной. Эта операция служит для изменения конфигурации SNMP-совместимых устройств.

trap/snmpV2-trap

`trap` – это имя SNMPv1, а `snmpV2-trap` – имя из SNMPv2/3. Обсуждение прерываний лежит за рамками данной книги, но, коротко говоря, они позволяют получить информацию о событиях (перезагрузка или достижение границы счетчиком) с SNMP-совместимой машины без явного запроса подобной информации.

response

`response` – это операция, применяемая для передачи ответа любой из операций. Ее можно использовать для ответа на `get-request`, для со-

[1] Канонический список PDU можно найти в RFC1905 для SNMPv2 и SNMPv3, который основывается на списке из RFC1157 для SNMPv1. Список из RFC незначительно длиннее приведенного здесь, так что вы не много теряете.

общения о том, успешно ли завершилась операция `set-request` и т. д. Вам редко придется явно применять эту операцию при программировании, т. к. большая часть SNMP-библиотек, программ и Perl-модулей обрабатывают ответы автоматически. Тем не менее, очень важно понимать не только правила построения запросов, но и особенности генерирования ответов.

Если читатель никогда раньше не имел дела с SNMP, естественная реакция на приведенный выше список была бы такой: «Что это? Получить, установить, сказать о том, что что-то произошло, и это все?» Но *простой* по замыслу создателей SNMP отнюдь не противоположность *мощному*. Если производитель SNMP-устройства правильно выберет переменные, при помощи протокола можно будет сделать практически все. Классический пример из RFC – это перезагрузка SNMP-совместимого устройства. Специальных операций для «запроса на перезагрузку» может и не существовать, но производитель сумеет легко реализовать эту операцию, используя триггерную переменную SNMP для хранения количества секунд, прошедших до загрузки. Когда эта переменная изменяется через `set-request`, перезагрузку устройства можно выполнить в указанное время.

Но если доступны такие возможности, то что предпринимается в плане безопасности, чтобы любой, у кого есть SNMP-клиент, не смог бы перегрузить вашу машину? В более ранних версиях протокола механизм защиты был очень слаб. На самом деле, некоторые даже расшифровывали аббревиатуру SNMP как «Security Not My Problem» (Безопасность – не моя проблема) из-за слабого механизма аутентификации в SNMPv1. Для того чтобы разобраться в тонкостях механизма защиты, т. е. понять, *кто*, *что* и *как*, нужно разобраться с терминологией, так что потерпите.

SNMPv1 и SNMPv2C позволяют определить административные отношения между SNMP-объектами, известными как *сообщества* (*communities*). Сообщества – это способ группировки SNMP-агентов со сходными ограничениями на доступ с управляющими объектами, которые удовлетворяют этим ограничениям. Все объекты из сообщества имеют одно и то же *имя сообщества*. Для доказательства того, что вы являетесь членом сообщества, необходимо знать только имя сообщества. Это и есть составляющая «*кто имеет доступ*?»

Что касается составляющей «к чему у них есть доступ?», в RFC1157 части MIB, имеющие отношение к определенному элементу сети, называются *SNMP MIB view*. Например, SNMP-совместимый тостер[1] не будет поддерживать те же конфигурационные переменные SNMP, что и SNMP-совместимый маршрутизатор.

[1] Существует SNMP-совместимая машинка для производства кока-колы (информацию об этом можно найти на *http://www.nixu.fi/limu*), так что все это не совсем притянуто за уши.

Каждый объект в MIB определен как `read-only` (доступный только для чтения), `read-write` (доступный для чтения/записи) или `none` (нет доступа). Это называется *режимом SNMP-доступа* (*SNMP access mode*) для объекта. Если объединить SNMP MIB view и режим SNMP-доступа, будет получен *профиль сообщества SNMP* (*SNMP community profile*), описывающий тип доступа, предоставленный переменным в MIB определенным сообществом.

Теперь, объединив части *кто* и *что*, мы получим *политику доступа SNMP* (*SNMP access policy*), описывающую, какие типы доступа предлагают друг другу члены определенного сообщества.

Как все это работает в реальной жизни? Вы настраиваете маршрутизатор или рабочую станцию так, чтобы они входили как минимум в два сообщества, одно из которых контролирует доступ для чтения, а другое – доступ для чтения и записи. Часто их называют сообществами `public` и `private`, в соответствии с популярными именами по умолчанию для этих сообществ. Например, можно добавить это как часть конфигурационной информации для маршрутизатора Cisco:

```
! устанавливаем имя доступного только для чтения сообщества в
MyPublicCommunityName
snmp-server community MyPublicCommunityName RO

! устанавливаем имя доступного для чтения и записи сообщества в
MyPrivateCommunityName
snmp-server community MyPrivateCommunityName RW
```

В Solaris можно было бы добавить эти строчки в файл */etc/snmp/conf/snmpd.conf*:

```
read-community  MyPublicCommunityName
write-community MyPrivateCommunityName
```

SNMP-запросы к любому из этих устройств будут использовать имя сообщества `MyPublicCommunityName` для получения доступа к переменным, доступным только для чтения, и `MyPrivateCommunityName` для изменения переменных, доступных для чтения и записи, на этих устройствах. Имя сообщества затем выступает в качестве псевдо-пароля для получения SNMP-доступа к устройству. Это довольно убогая схема безопасности. Имя сообщества не только передается открытым текстом в каждом SNMP-пакете, но оно также пытается защитить доступ посредством механизма «безопасность через скрытность».

В более поздних версиях SNMP, в частности в версии 3, безопасность протокола была значительно улучшена. В RFC2274 и RFC2275 определяется пользовательская модель безопасности (User Security Model, USM) и модель управления доступом «View-Based» (View-Based Access Control Model, VACM). USM предоставляет криптографическую защиту аутентификации и шифрует сообщения. VACM предлагает исчер-

пывающий механизм управления доступом для объектов MIB. Эти механизмы пока еще являются относительно новыми и нереализованными (например, только один из Perl-модулей поддерживает их, но и такая поддержка появилась совсем недавно). Мы не будем обсуждать здесь эти механизмы, но вам, вероятно, стоит внимательно изучить RFC, т. к. популярность SNMPv3 увеличивается.

SNMP на практике

Теперь, когда вы получили изрядную дозу теории SNMP, применим эти знания на практике. Вы уже знаете, как запросить системное описание для машины (помните краткое введение, приведенное ранее). Рассмотрим еще два примера: запрос времени непрерывной работы системы и таблицы маршрутизации IP.

До сих пор вы полагались только на мои слова при поиске местоположения и имени SNMP-переменной в MIB. Это нужно изменить, поэтому первый шаг при запросе информации через SNMP – это процесс, который я называю «MIB groveling» («раболепство перед MIB»):

Шаг 1

Найти нужный документ MIB. Если вы ищете независимые от устройства настройки, которые можно встретить на любом SNMP-устройстве, скорее всего, они найдутся в RFC1213.[1] Если вам нужны имена переменных, зависящих от производителя, например, переменной, в которой хранится «цвет мерцающей лампочки на передней панели конкретного ATM-коммутатора», то следует обратиться к производителю коммутатора и запросить копию их *модуля MIB* (*MIB module*). Я педантично отношусь к применяемым терминам, потому что нередко можно услышать, как люди неверно говорят: «Мне нужна база MIB для этого устройства». В мире существует только одна база MIB; все остальное находится где-то в ее структуре (обычно где-то под ветвью `private(4)`).

Шаг 2

Найти в описаниях MIB нужную вам SNMP-переменную.

Чтобы упростить второй шаг, разберемся с форматом.

Описания MIB совсем не так страшны, когда вы привыкаете к ним. Они выглядят как один длинный набор объявлений переменных, похожий на то, что можно найти в исходном коде. Это не совпадение, потому что они и *есть* объявления переменных. Если производитель ответственно подойдет к созданию своего модуля, то в нем будет достаточно комментариев, как и в любом хорошем исходном тексте.

[1] RFC1213 в некоторой степени был обновлен RFC2011, RFC2012 и RFC2013. В RFC1907 к MIB добавлены элементы SNMPv2.

Информация MIB записывается в соответствии со стандартным соглашением OSI (Open Systems Interconnection, взаимодействие открытых систем) подмножества ASN.1 (Abstract Syntax Notation One, язык для описания абстрактного синтаксиса данных). Описание этого подмножества и другие сведения об описании данных для SNMP можно найти в RFC под названием «Structure for Management Information» (SMI). Они дополняют RFC, определяющие протокол SNMP и текущую базу MIB. Например, самое последнее (на момент готовности рукописи этой книги) описание протокола SNMP можно найти в RFC1905, описание самой последней базы MIB, с которой работает протокол, – в RFC1907, а SMI для этой базы MIB – в RFC2578. Я обращаю на это ваше внимание, поскольку нередко приходится обращаться к разным документам при поиске определенного SNMP-объекта.

Воспользуемся этими знаниями, чтобы решить первую задачу: выясним, используя SNMP, время непрерывной работы машины. Вопрос довольно общий, так что вероятность найти нужную SNMP-переменную в RFC1213 очень велика. Быстрый поиск «uptime» в RFC1213 позволяет получить такой отрывок из ASN.1:

```
sysUpTime OBJECT-TYPE
              SYNTAX  TimeTicks
              ACCESS  read-only
              STATUS  mandatory
              DESCRIPTION
                      "The time (in hundredths of a second) since the
                      network management portion of the system was last
                      re-initialized."
              ::= { system 3 }
```

Внимательно разберемся в этом определении:

`sysUpTime OBJECT-TYPE`

Эта строка определяет объект под названием `sysUpTime`.

`SYNTAX TimeTicks`

Это объект типа `TimeTicks`. Типы объектов определяются в SMI, о них упоминалось совсем недавно.

`ACCESS read-only`

Этот объект доступен только для чтения через SNMP (т. е. `get-request`); его нельзя изменить (т. е. `set-request`).

`STATUS mandatory`

Данный объект должен быть реализован в любом SNMP-агенте.

`DESCRIPTION...`

Это текстовое описание объекта. Всегда внимательно читайте все, что написано в подобном поле. В этом определении нас ждет сюрприз. `sysUpTime` показывает только то время, которое прошло с мо-

мента последней инициализации части системы, имеющей отношение к управлению сетью («the network management portion of the system was last re-initialized»). Это означает, что мы можем узнать только время непрерывной работы с момента последнего запуска SNMP-агента. Почти всегда это время совпадает с временем непрерывной работы самой системы, так что если вы заметите отклонение, то причина, скорее всего, будет в этом.

```
::= { system 3 }
```

Эта строка определяет, где именно в дереве MIB хранится объект. Объект `sysUpTime` – это третья ветвь дерева системной группы объектов. Подобная информация также позволяет получить часть идентификатора объекта, которая понадобится позже.

При желании запросить эту переменную с машины *solarisbox* в сообществе, доступном только для чтения, можно было бы использовать такую командную строку из UCD-SNMP:

```
$ snmpget solarisbox MyPublicCommunityName system.sysUpTime.0
```

В результате получим:

```
system.sysUpTime.0 = Timeticks: (5126167) 14:14:21.67
```

То есть агент был последний раз инициализирован 14 часов назад.

В примерах из этого приложения подразумевается, что SNMP-агенты настроены таким образом, что принимают запросы от запрашивающих узлов. В целом, если можно разрешить SNMP-доступ только с небольшого подмножества узлов, это следует сделать.

«Нужно знать» – это отличный принцип безопасности, которому надо следовать. Хорошо было бы ограничить сетевые службы, предоставляемые каждой машиной или устройством. Если не нужно предоставлять сетевую службу, ее лучше отключить. Если служба необходима, ограничьте доступ к ней только теми устройствами, которым это «нужно знать».

Пришло время привести второй, более совершенный SNMP-пример: сброс содержимого таблицы маршрутизации устройства. Сложность этого примера состоит в том, что надо обрабатывать группу скалярных данных как одну логическую таблицу. Для получения этих данных выполним операцию `get-next-request`. Наш первый шаг на пути к цели – найти описание MIB таблицы маршрутизации. Поиск по «route» в RFC1213, в конечном счете, выдает следующее определение:

```
-- The IP routing table contains an entry for each route
-- presently known to this entity.
-- (Таблица маршрутизации содержит запись для каждого маршрута,
```

```
-- известного в настоящий момент этому элементу.)
ipRouteTable OBJECT-TYPE
    SYNTAX  SEQUENCE OF IpRouteEntry
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
              "This entity's IP Routing table."
    ::= { ip 21 }
```

Это определение не слишком отличается от того, которое мы видели минутой раньше. Отличия есть только в строках ACCESS и SYNTAX. Строка ACCESS предупреждает, что этот объект является всего лишь структурной единицей, представляющей целую таблицу, а не настоящей переменной, к которой можно направить запрос. Строка SYNTAX говорит о том, что таблица состоит из целого ряда объектов IpRouteEntry. Посмотрим на начало определения IpRouteEntry:

```
ipRouteEntry OBJECT-TYPE
    SYNTAX  IpRouteEntry
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
            "A route to a particular destination."
    INDEX   { ipRouteDest }
    ::= { ipRouteTable 1 }
```

Строка ACCESS говорит, что мы нашли еще один «заполнитель» (placeholder), на этот раз для каждой строки из таблицы. Но этот заполнитель может нам кое-что поведать. Можно обратиться к каждой строке, используя индексацию, т. е. объект ipRouteDest для каждой строки.

Если эти несколько уровней определений вас утомляют, обратитесь за помощью к Perl. Будем считать, что мы имеем дело с хэшем списков в Perl. Ключом хэша для строки будет переменная ipRouteDest. Значением этого хэша будет ссылка на список, содержащий другие элементы этой строки (т. е. остальную информацию о маршруте).

Определение ipRouteEntry выглядит так:

```
ipRouteEntry ::=
    SEQUENCE {
        ipRouteDest
            IpAddress,
        ipRouteIfIndex
            INTEGER,
        ipRouteMetric1
            INTEGER,
        ipRouteMetric2
            INTEGER,
        ipRouteMetric3
            INTEGER,
        ipRouteMetric4
```

```
            INTEGER,
        ipRouteNextHop
            IpAddress,
        ipRouteType
            INTEGER,
        ipRouteProto
            INTEGER,
        ipRouteAge
            INTEGER,
        ipRouteMask
            IpAddress,
        ipRouteMetric5
            INTEGER,
        ipRouteInfo
            OBJECT IDENTIFIER
    }
```

Теперь вы видите элементы, составляющие каждую строку в таблице. MIB продолжается описанием этих элементов. Вот два первых определения:

```
ipRouteDest OBJECT-TYPE
    SYNTAX  IpAddress
    ACCESS  read-write
    STATUS  mandatory
    DESCRIPTION
            "The destination IP address of this route. An
            entry with a value of 0.0.0.0 is considered a
            default route. Multiple routes to a single
            destination can appear in the table, but access to
            such multiple entries is dependent on the table-
            access mechanisms defined by the network
            management protocol in use."
     ::= { ipRouteEntry 1 }

 ipRouteIfIndex OBJECT-TYPE
     SYNTAX  INTEGER
     ACCESS  read-write
     STATUS  mandatory
     DESCRIPTION
             "The index value which uniquely identifies the
             local interface through which the next hop of this
             route should be reached. The interface identified
             by a particular value of this index is the same
             interface as identified by the same value of
             ifIndex."
     ::= { ipRouteEntry 2 }
```

Графическое представление ipRouteTable поможет вам разобраться с этой информацией (рис. E.3).

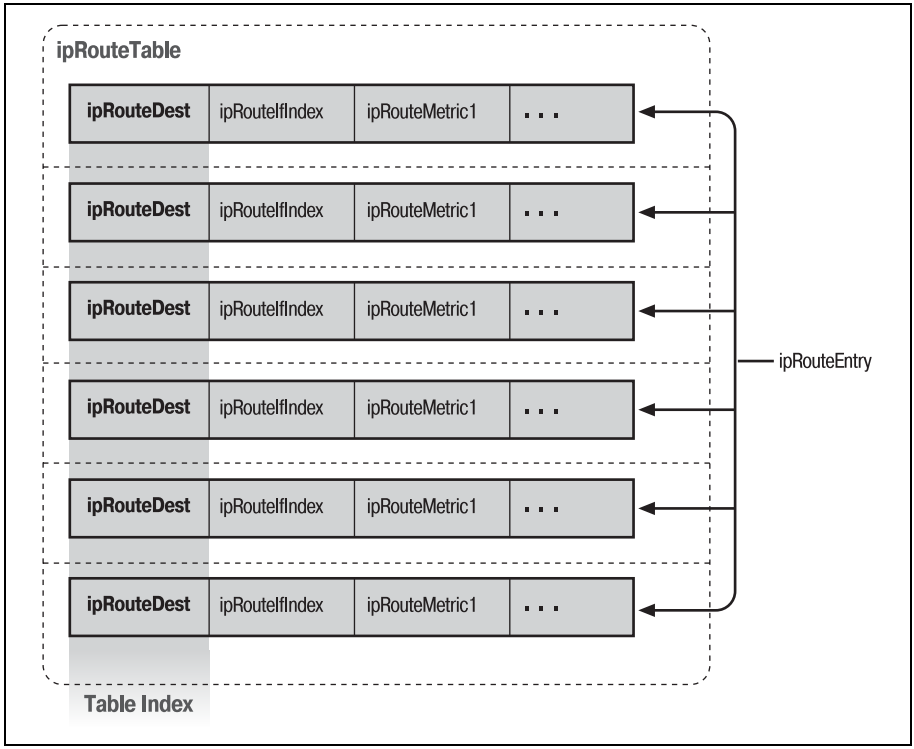

Рис. E.3. Структура ipRouteTable и ее индекс

Если вы разобрались с этой частью MIB, то на следующем шаге следует запросить информацию. Данный процесс называется «обходом таблицы». В большинство SNMP-пакетов входит утилита командной строки, называемая либо *snmptable*, либо *snmp-tbl*, выполняющая такой обход, но утилиты могут и не предоставить нужной вам степени контроля. Например, возможно, нужна не вся таблица маршрутизации, а лишь список ipRouteNextHops. Кроме того, в некоторых SNMP-пакетах Perl просто нет подпрограмм для обхода деревьев. Так что имеет смысл знать, как выполнять все эти действия вручную.

Чтобы было проще понять, как это делается, я приведу информацию, которую мы в конечном итоге собираемся получать от устройства. Это позволит вам увидеть, как каждый шаг добавляет в таблицу еще одну строку собираемых данных. Если зарегистрироваться на удаленной машине (вместо того чтобы использовать SNMP для удаленного запроса) и набрать *netstat –nr* для сброса таблицы маршрутизации, вывод может выглядеть так:

```
default          192.168.1.1       UGS         0     215345  tu0
```

```
127.0.0.1        127.0.0.1        UH          8    5404381  lo0
192.168.1/24     192.168.1.189    U          15    9222638  tu0
```

Это соответствует внутреннему интерфейсу обратной петли по умолчанию и локальным сетевым маршрутам.

Теперь посмотрим, как получить часть этой информации через утилиты UCD-SNMP. В данном примере нас интересуют только первые два столбца данных (направление маршрута и адрес следующего узла). Мы выполняем первоначальный запрос к первому экземпляру этих двух переменных в таблице. Все, что выделено жирным шрифтом, – это одна длинная команда, и здесь она разбита на две строчки только для разборчивости:

```
$ snmpgetnext computer public ip.ipRouteTable.ipRouteEntry.ipRouteDest
ip.ipRouteTable.ipRouteEntry.ipRouteNextHop
ip.ipRouteTable.ipRouteEntry.ipRouteDest.0.0.0.0 = IpAddress: 0.0.0.0
ip.ipRouteTable.ipRouteEntry.ipRouteNextHop.0.0.0.0 = IpAddress: 192.168.1.1
```

Следует обратить внимание на две части подобного ответа. Первая – это данные, возвращаемые после знака равенства. `0.0.0.0` означает «маршрут по умолчанию», так что подобная информация соответствует первой строке таблицы маршрутизации из приведенного выше примера. Вторая важная часть ответа – это добавленные к именам переменных `.0.0.0.0`. Это индекс для элемента `ipRouteEntry`, представляющего строку таблицы.

Получив первую строку, можно выполнить еще один вызов `get-next-request`, на этот раз, используя индекс. Запрос `get-next-request` всегда возвращает *следующий* элемент из MIB, так что достаточно передать ему индекс строки, которую мы только что получили, для перехода к следующей после нее строке:

```
$ snmpgetnext gold public ip.ipRouteTable.ipRouteEntry.ipRouteDest.0.0.0.0
ip.ipRouteTable.ipRouteEntry.ipRouteNextHop.0.0.0.0
ip.ipRouteTable.ipRouteEntry.ipRouteDest.127.0.0.1 = IpAddress: 127.0.0.1
ip.ipRouteTable.ipRouteEntry.ipRouteNextHop.127.0.0.1 = IpAddress: 127.0.0.1
```

Вероятно, читатель уже догадался, каким будет следующий шаг. Мы используем еще один запрос `get-next-request` при помощи фрагмента `127.0.0.1` (индекс) ответа `ip.ipRouteTable.ipRouteEntry.ipRouteDest.127.0.0.1`:

```
$ snmpgetnext gold public ip.ipRouteTable.ipRouteEntry.ipRouteDest.127.0.0.1
ip.ipRouteTable.ipRouteEntry.ipRouteNextHop.127.0.0.1
ip.ipRouteTable.ipRouteEntry.ipRouteDest.192.168.1 = IpAddress: 192.168.1.0
ip.ipRouteTable.ipRouteEntry.ipRouteNextHop.192.168.11.0 = IpAddress:
192.168.1.189
```

Если вы посмотрите на вывод команды *netstat*, приведенный выше, то увидите, что мы достигли своей цели и получили все строки из таблицы маршрутизации. Как бы мы узнали об этом, если бы по иронии

судьбы у нас не было полученного ранее вывода команды *netstat*? В обычных обстоятельствах следовало действовать как обычно и послать запрос:

```
$ snmpgetnext gold public
ip.ipRouteTable.ipRouteEntry.ipRouteDest.192.168.1.0
ip.ipRouteTable.ipRouteEntry.ipRouteNextHop.192.168.1.0
ip.ipRouteTable.ipRouteEntry.ipRouteIfIndex.0.0.0.0 = 1
ip.ipRouteTable.ipRouteEntry.ipRouteType.0.0.0.0 = indirect(4)
```

Стоп, ответ не совпадает с запросом! Ведь запрашивали `ipRouteDest` и `ipRouteNextHop`, а получили `ipRouteIfIndex` и `ipRouteType`. Мы вышли за пределы таблицы `ipRouteTable`. SNMP-запрос `get-next-request` честно выполнил свой долг и вернул «первого лексикографического последователя» (first lexicographic successor) из MIB для каждого объекта из запроса. Если посмотреть в определение `ipRouteEntry` из RFC1213, то можно увидеть, что `ipRouteIfIndex(2)` следует за `ipRouteDest(1)`, а `ipRouteType(8)` действительно следует за `ipRouteNextHop(7)`.

Так что ответ на вопрос, заданный минуту назад: «Как узнать, что мы получили все данные из таблицы?», будет таким: «Когда вы заметите, что вышли за пределы таблицы». С точки зрения программирования это равнозначно тому, чтобы проверить, возвращается ли в ответ на ваш запрос одна и та же строка или префикс OID. Например, можно убедиться, что все ответы на запрос о `ipRouteDest` содержали либо `ip.ipRouteTable.ipRouteEntry.ipRouteDest`, либо `1.3.6.1.2.1.4.21.1.1`.

Познакомившись с основами SNMP, можно возвращаться к главе 10 и узнать, как применять SNMP из Perl. Также стоит проверить ссылки в конце главы 10 для получения более подробной информации по SNMP.

Алфавитный указатель

M

N

O

P

Q

R

S

T

U

Y

А

Б

В

Г

Д

Ж

З

И

К

М

Н

О

П

Р

С

Т

У

Ф